Neural Network Reconstruction via Graph Locality-Driven Machine Learning

Bard College

Hayden Sartoris

Contents

1 Background **2**

1.1 Biological Neural Networks 2

1.2 Graph Structures 2

1.2.1 Graph Locality 2

1.3 Convolutional Neural Networks 2

1.3.1 Adaptation to Graph Locality 2

1.4 Concepts and Terms 2

1.4.1 Adjacency Matrices 3

2 Model **4**

2.1 Data 4

2.1.1 Generation 5

2.1.2 Restructuring 7

2.1.3 Generalizability 8

2.2 Architecture 9

2.2.1 Structure & Computation Details 9

2.2.2 Conceptual Model 10

2.2.3 Matrix Model 13

2.2.4 Benchmark Model 16

2.2.5 n-independence 17

3 Training **19**

3.1 Activation Functions 19

3.1.1 Initial & Convolutional Layers 19

3.1.2 Final Layer 20

3.2 Loss & Optimization 21

3.2.1 Loss Function 21

3.2.2 Optimizer Function 23

3.3 Datasets 24

3.4 Matrices 24

3.4.1 Initialization 24

3.4.2 Locality Layer Operations 24

3.5 Hyperparameter Optimization 26

3.5.1 Batch Size 26

4 Results **27**

4.1 Overfitting 27

4.1.1 Empty Data 27

4.1.2 Random Data 28

4.1.3 Analysis 28

4.2 3-neuron generator 29

4.2.1 Example Model 30

4.2.2 Trained Network Operation 31

4.3 Higher-order Datasets 32

4.4 Applicability Beyond Training Data 34

4.4.1 Inverted Network 35

4.4.2 Cyclical Network 35

5 Discussion **37**

5.1 Data 37

5.1.1 Complex Neurons 38

5.1.2 Larger, Structured Networks 38

5.2 Potential Improvements to Locality 39

5.2.1 Layering 39

5.2.2 Algorithm 39

5.2.3 Loss 39

5.2.4 Optimizer 40

5.3 Potential Applications/Further Development 40

A Parameter Optimization Miscellanea **41**

A.1 Data 41

A.1.1 Spike Rate Determination 41

B Model **42**

B.1 Batched Architecture Calculations 42

List of Figures

2.1 Example of 3-neuron network and adjacency matrix. 5

2.2 Example output matrix for a 3-neuron network simulated for five steps. 7

2.3 Transposed and truncated matrix and associated visualization. 8

2.4 Relationship between $\mathbb{D}'$ and $\mathbb{D}'_N$. 11

3.1 ReLU function definition and graph 19

3.2 Sigmoid function definition and graph 20

3.3 Graph of $y = \tanh(x)$ 20

3.4 Example adjacency matrix 22

3.5 Adam decay function over 100 steps. Converges asymptotically to 1. 23

4.1 Training parameters for null hypothesis networks 27

4.2 Predictions and losses when training on an empty dataset . . . 28

4.3 Average prediction for random data. loss: 0.5 28

4.4 Network structure and adjacency matrix of the generator. (Reproduced from Figure 2.1) 29

4.5 Path of data through network. Transparency for each value is scaled relative to the maximum value found in the matrix. . . 31

4.6 Final weights (max: 7.31) 32
4.7 Ten neuron generator and adjacency matrix. For purposes of clarity, all zero values in the matrix have been omitted. 33
4.8 Loss & parameters for model trained on data from generator given in Figure 4.7 33
4.9 Example of data from generator defined in Figure 4.7, passed through the locality-based and benchmarks models. 34
4.10 Inverted version of Figure 4.4 35
4.11 Cyclical 3-neuron network 35

Abstract

A ubiquitous problem within the field of computational neuroscience is the determination of biological neural network structure and connectivity from imaging of stochastic, large-scale network activity. We propose a machine learning algorithm inspired by convolutional approaches to image processing, adapted to the graph structure of neural networks. To achieve this, we redefine locality in terms of graph adjacency, and create a scale-independent algorithm facilitated by modern machine learning techniques to incorporate this locality data into individual connection prediction.

1 Background

1.1 Biological Neural Networks

1.2 Graph Structures

1.2.1 Graph Locality

1.3 Convolutional Neural Networks

Convolutional neural networks as we know them today were first put forth by LeCun et al. in

1.3.1 Adaptation to Graph Locality

1.4 Concepts and Terms

Before diving into the specifics of data production, model architecture, and training, it's important to establish several important concepts.

1.4.1 Adjacency Matrices

The representation of neural network connectivity that we will focus on is the adjacency matrix. For n neurons, an adjacency matrix $\mathbb{M}$ will be of dimensions $(n \times n)$. A simplistic method of predicting network activity, and one that we will use to produce our data, is to multiply this matrix by an n-vector representing current activity at each neuron. Such an operation appears as follows for $n = 3$:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Thus the activity for a given neuron is defined entirely in terms of network activity at the previous timestep and the weights in the adjacency matrix in the row corresponding to that neuron. We thereby arrive at a simple expression of the mechanics of adjacency matrices:

1. Weights in some row i define inputs to neuron i
2. Weights in some column j define outputs from neuron j
3. The singular weight at $\mathbb{M}_{ij}$ defines the connection from neuron j to neuron i.

Keeping this inverse relationship in mind will help prevent confusion in later chapters.

2 Model

The model trained and tested here represents ... stuff

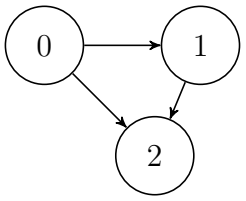
2.1 Data

Insofar as we treat ANNs as providing arbitrary function approximation, training a network requires input data representing the known data about the system we wish to model, as well as output data we wish the network to produce from the inputs. More generally, input data usually entails information that is easy to acquire about the process being modeled, while output data, or labels, correspond to a dataset that is difficult to acquire generally. Of course, this means that the first step in training a neural network is to assemble a sufficiently large set of inputs and outputs in order to fully, or at least approximately, characterize the problem at hand.

In our case, we wish to map from (relatively) easily available data about biological networks, individual neuron spike times, to network structure. While such data exist, generating our own allows us to better analyze the results of the algorithm.

2.1.1 Generation

In order to demonstrate the validity of our algorithm for graph convolution, we opt for a simplified form of the kind of data that would be used in a real-world setting. To this end, we create adjacency matrices representing simple, small-n toy networks.



	0	1	2
0	0	0	0
1	1	0	0
2	1	1	0

Figure 2.1: Example of 3-neuron network and adjacency matrix.

Binary values are used throughout these toy networks: either a connection exists or it doesn't; either a 'neuron' is spiking or it isn't. To produce spiking data, we create an n-vector $\mathbb{S}$ representing the current state of the toy network, with random neurons already spiking based on a chosen spike rate. From here, the process is as in 1.4.1, where $\mathbb{M}$ is the adjacency matrix:

$$\underset{n\times n}{\mathbb{M}} \times \underset{n\times 1}{\mathbb{S}^t} = \underset{n\times 1}{\mathbb{S}^{t+1}}$$

Additonally, $\mathbb{S}^{t+1}$ may have one or more neurons spike randomly, as determined by the spike rate of the simulation.[1] All values are clipped to the range $[0, 1]$, to avoid double spiking. At each step, $\mathbb{S}$ is appended to an output matrix, which is saved after simulation is complete. For t simulation steps, the completed output has shape $(n \times t)$.

Generally, we ran simulations as described for 50 steps[2], then saved the resulting output matrix. As many as fifty thousand simulations were run for

[1] SEE APPENDIX

[2] See 2.1.2

each generator network. As well as saving the simulated spike trains, we save the adjacency matrix describing the generator, in order to provide a target for the model to train on.

Example Data Generation

Consider the network defined in Figure 2.1. Supposing that we randomly spike neuron 0 at the first step, our initial state appears as such, where $\mathbb{O}$ is the output matrix and $\mathbb{R}^0$ is an n-vector wherein each element has been randomly assigned 0 or 1, based on the spike rate of the simulation:

$$\mathbb{M} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \qquad \mathbb{S}^0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbb{O} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbb{R}^0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

We now compute $\mathbb{S}^1$ as above:

$$\mathbb{S}^1 = (\mathbb{M} \times \mathbb{S}^0) + \mathbb{R}^0 = \left(\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

In this case, neuron 1 was spiked randomly, but was also spiked by virtue of its connection from 0. Since in this simple model we only consider neurons to be either spiking or not, binary values, we clip the values in $\mathbb{S}^1$ to a maximum of 1, in order to prevent cases such as this one from causing spikes of greater magnitude to propagate through the network. This also prevents neurons from double spiking due to multiple inputs being active in the same timestep. Thus we have our final value for $\mathbb{S}^1$, and append it to $\mathbb{O}$.

$$\mathbb{S}^1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \qquad \mathbb{O} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

If we were to repeat this process several more times, we might end up with an output matrix such as in Figure 2.2.

$$\mathbb{O} = \left[\mathbb{S}^0 \mid \mathbb{S}^1 \mid \mathbb{S}^2 \mid \mathbb{S}^3 \mid \mathbb{S}^4\right] = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 2.2: Example output matrix for a 3-neuron network simulated for five steps.

We can clearly see the effects of neuron 2 having inputs from both other neurons. Practically, the number of iterations was usually set to 50.

2.1.2 Restructuring

Input Data

The model accepts data in the form of a spike-time raster plot of dimensions $(n \times t)$, where n is the number of neurons and t is the number of timesteps being considered. The axes are reversed in comparison to the data created by the generator, and thus in the process of loading in the spike trains we transpose the matrices to the expected dimensionality. Additionally, it is not always necessary to use the full number of steps generated, depending on the size of the generator network in question, as well as its spike rate. In such a scenario, we truncate the time dimension appropriately.

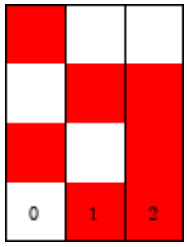
For a network accepting t timesteps of data from n neurons, the data fed into the network takes the following form:

$$\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{t1} & x_{t2} & \dots & x_{tn} \end{bmatrix}$$

Applying this process to the data in Figure 2.2, including truncating the time dimension to four, produces the data in Figure 2.3.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

(a) Transposed output matrix



(b) Graphical representation

Figure 2.3: Transposed and truncated matrix and associated visualization.

The representation of the matrix in 2.3b is an example of the method we will use to depict matrices containing real values.

Target Data

As described in 2.1.1, we save the adjacency matrix corresponding to the generator along with the simulated spiking files. When an adjacency matrix is loaded into the target dataset for training a model, we flatten it, from $(n \times n)$ to $(1 \times n^2)$. This allows us to directly compare our targets to the outputs of the model, which will be of the same dimensionality.

2.1.3 Generalizability

In most ANN implementations, feeding various data with the same label attached to it results in the network learning to ignore the input data and always return the desired label, rendering it useless. However, due to the unique structure of our model, this sort of overfitting is impossible.[3] Therefore, we must merely construct a suitably representative generator network, meaning that it contains all of the inter-neuron relationships we expect to see in the data we

[3] See **??**

ultimately feed in to test.

2.2 Architecture

We will first describe the architecture in terms that, while accurate on the macro level, do not fully reflect the actual transformations occuring in the implemented model. We will then proceed to a mathematically representative version, leaving explanation of the batched version of the model to B.1. Additionally, we describe a benchmark model not involving locality calculations, in order to provide a point of reference for the efficacy of our implementation.

2.2.1 Structure & Computation Details

Dimensionality-defining Variables

Only two values characterize the matrices and transitions involved in the model. They are as follows:

b: The number of steps of input data the model considers in a given segment of data.

d: The length of the vectors characterizing each potential connection ij. This restricts the maximum information about each potential neuron pair that the model can maintain across layer transitions.

We determined effective values for these parameters through experimentation.

While we use the number of nodes in the generator graph, n, to calculate summations and averages, the structure of our calculations is such that no aspects of the model are defined in terms of n.

Omitted Details

An elementwise activation function[4] is applied to the matrix outputs from each layer. While this is crucial to network function, our primary focus in this section is the underlying principles and mathematical expressions thereof, and activation is somewhat trivial in comparison. For details on the activation functions used, see 3.1.

2.2.2 Conceptual Model

The operations we describe here represent a per-edge approach to our architecture; i.e., the layer transitions are defined in terms of calculations applied to single pairs of nodes, as opposed to the whole-matrix operations that the architecture as implemented relies on.

First Transition

To generate the first layer of the network, we inspect every pair of neurons in the input data. Since no pair of neurons is distinguishable from another, the comparison applied is the same in all cases: we apply the same convolutional filter to all pairs. We achieve this by concatenating the spike train of each neuron i individually with every other neuron j, then multiplying by a matrix $\mathbb{W}$ of dimensionality $(d \times 2b)$. To this product we add a bias vector, $\mathbb{B}$, of dimensionality $(d \times 1)$.

$\mathbb{W}$ is trained on, and thus the comparison of each pair of spike trains is left up to the network. The transition appears as follows, where $\underset{b\times 1}{\mathbb{I}_x}$ is the input

[4]SEE NN PRINCIPLES

column at x:

$$\forall i, j \mid 0 \leq i, j < n : \underset{d \times 1}{d'_{ij}} = \underset{d \times 2b}{\mathbb{W}} \times \left(\frac{\mathbb{I}_i}{\mathbb{I}_j} \right) + \underset{d \times 1}{\mathbb{B}}$$

This leaves us with n^2 d-vectors, each characterizing one potential edge ij.

Locality Layer

In this layer, we incorporate information from all nodes potentially adjacent to each edge ij. From our previous layer, we have a matrix of shape $(d \times n^2)$ that we will refer to as $\mathbb{D}'$, but it will be useful to keep in mind an alternate representation of that matrix, one in three dimensions, which we shall refer to as $\mathbb{D}'_N$. This transformation is demonstrated in Figure 2.4.

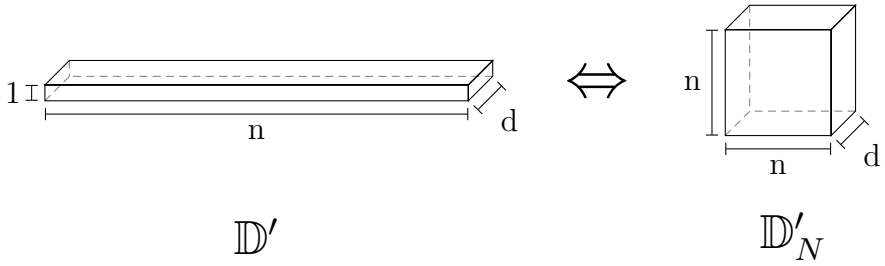


Figure 2.4: Relationship between $\mathbb{D}'$ and $\mathbb{D}'_N$.

Consider some d'_{ij} in $\mathbb{D}'_N$. Then we can say the following:

1. d'_{ij} represents the connection from j to i as it may or may not exist in this network, in the form of d values of indeterminate meaning
2. $\forall k \mid 0 \leq k < n$, d'_{jk} represents a potential input to j
3. $\forall k \mid 0 \leq k < n$, d'_{ki} represents a potential output from i

In our determination of the presence or absence of a connection from j to i, we wish to incorporate information from these potentially connected nodes;

i.e., these inputs and outputs represent potential neighbors in terms of graph locality. To achieve this, we perform the following computations[5] for each $\underset{d\times 1}{d_{ij}}$:

$$\underset{d\times 1}{\mathbb{I}} = \frac{1}{n}\sum_{k=0}^{n-1} d'_{jk} \qquad\qquad \underset{d\times 1}{\mathbb{O}} = \frac{1}{n}\sum_{k=0}^{n-1} d'_{ki} \tag{2.1a}$$

$$\underset{d\times 1}{\mathbb{I}_{\mathbb{D}}} = \underset{d\times d}{\mathbb{W}'_{in}} \times \left(\mathbb{I} \odot d'_{ij}\right) \qquad\qquad \underset{d\times 1}{\mathbb{O}_{\mathbb{D}}} = \underset{d\times d}{\mathbb{W}'_{out}} \times \left(\mathbb{O} \odot d'_{ij}\right) \tag{2.1b}$$

Here we arrive at the output, d''_{ij}:

$$\underset{d\times 1}{d''_{ij}} = \underset{d\times 2d}{\mathbb{W}'_{tot}} \times \left(\frac{\mathbb{I}_{\mathbb{D}}}{\mathbb{O}_{\mathbb{D}}}\right) + \underset{d\times 1}{\mathbb{B}'} \tag{2.1c}$$

Conceptually, in **(2.1a)** we first average all potential inputs to and outputs from potential edge ij. Then, we compute an entrywise product ($\odot$) of these vectors with the vector describing the edge in question, d'_{ij}. While we have integrated locality data into the results thus far, the network has not been allowed any processing over the resultant data, which we rectify by multiplying the input and output vectors with separate dimensionality-preserving $(d \times d)$ matrices. We thus arrive at **(2.1b)**, with vectors $\mathbb{I}_{\mathbb{D}}$ and $\mathbb{O}_{\mathbb{D}}$ representing edge ij with inputs and outputs, respectively, taken into consideration. In **(2.1c)**, we arrive at d''_{ij} by multiplying a third weight matrix by the vertical concatenation of $\mathbb{I}_{\mathbb{D}}$ and $\mathbb{O}_{\mathbb{D}}$. This matrix, $\mathbb{W}'_{tot}$, allows the network to optimize for whichever elements in $\mathbb{I}_{\mathbb{D}}$ and $\mathbb{O}_{\mathbb{D}}$ are most important in the prediction of ij. Additionally, a bias vector, $\mathbb{B}'$, is added to this product, and at this point we have d''_{ij} as it will be seen by the next layer of the network.[6]

Our concatenation approach in **(2.1c)** stands in contrast to the strategy taken in **(2.1b)**, where integration of the input and output data is forced via

[5] Actually, it's much more elegant.

[6] Disregarding the activation function

entrywise product computation. For discussion of this attribute, see 3.4.2.

Note again that none of the computations involved in this layer are dependent on n; as the summations are averaged, the values contained in their resultant vectors will be of similar magnitude for any number of neurons under consideration. After executing this algorithm for each d'_{ij}, we are left with another $(d \times n^2)$ output matrix, $\mathbb{D}''$.

Final Transition

The shift from $(d \times n^2)$ is comparatively simple, being only a dimensionality reduction:

$$\forall d''_{ij} \in \mathbb{D}'' : \underset{1\times1}{d^f_{ij}} = \underset{1\times d}{\mathbb{W}^f} \times \underset{d\times1}{d''_{ij}} \tag{2.2}$$

This leaves us with a $(1 \times n^2)$ matrix, which, following application of an activation function as defined in 3.1.2 and transposition to $(n \times n)$, we treat as the adjacency matrix of the generator associated with the input data.

2.2.3 Matrix Model

While the processes defined in 2.2.2 are accurate representations of the operations undertaken in our model, they are generally defined in terms of individual vectors, with iteration over all vectors necessarily implied. This does not take advantage of the computational abilities of modern GPU computing, and, if implemented as such, would render training times astronomical. Therefore, we create a version of our model executed entirely in terms of matrix operations, ideal for GPU execution.

First Layer

In the first layer, we wish to compare each input vector against every input vector by way of concatenation and matrix multiplication to reduce dimensionality. To achieve this via matrix operations is fairly simple. We first define two helper matrices:

$$\underset{n\times n^2}{\mathbb{E}} = \begin{bmatrix} \underset{1\times n}{\mathbb{1}} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \underset{1\times n}{\mathbb{1}} \end{bmatrix}$$

$$\underset{n\times n^2}{\mathbb{T}} = \begin{bmatrix} I_n & | & \dots & | & I_n \end{bmatrix}$$

With $\underset{b\times n}{\mathbb{I}}$ as our input data, the first layer transition is as follows:

$$\underset{d\times n^2}{\mathbb{D}'} = \underset{d\times 2b}{\mathbb{W}} \left(\frac{\mathbb{I}\times\mathbb{E}}{\mathbb{I}\times\mathbb{T}} \right) + \left(\underset{d\times 1}{\mathbb{B}} \times \underset{1\times n^2}{\mathbb{1}} \right) \tag{2.3}$$

Example: Consider a model for which $b = 3$ and $n = 2$. Suppose that we have the following input matrix:

$$\mathbb{I} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Then our helper matrices would appear as such:

$$\mathbb{E} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \mathbb{T} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

And our matrix stack:

$$\mathbb{I} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \frac{\mathbb{I}\times\mathbb{E}}{\mathbb{I}\times\mathbb{T}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Thus, over all of the columns in the resulting stack, every vector in $\mathbb{I}$ is paired with all such vectors, including itself.

Locality Layer

In the conceptual model, there are two averages of sums involved in processing each vector in $\mathbb{D}'$; one over the horizontal axis of $\mathbb{D}'_N$, and the other over the vertical axis. These can be found in (**2.1a**). Consider two vectors $d_{ij}, d_{il} \in \mathbb{D}'_N$. For both of these vectors, the average input vector is the same, its calculation being only dependent on the first coordinate, i. The inverse holds for vectors with the same second coordinate. Thus we see that these calculations need only be performed once for each $k \in [0, n)$. Considering the $(d \times n^2)$ representation of the current data matrix $\mathbb{D}'$, the 'vertical' summation of column i appears as such:

$$\mathbb{O} = \sum_k d'_{ki} = \mathbb{D}'_{0+i} + \mathbb{D}'_n + \mathbb{D}'_{2n+i} + \cdots + \mathbb{D}'_{(n-1)n+i}$$

This is the inverse of the tile operation executed by $\mathbb{T}$ in the first layer, and that same matrix allows us to compute all outputs to all edges simultaneously:

$$\underset{d \times n}{\mathbb{O}} = \frac{1}{n} \left(\mathbb{D}' \times \mathbb{T}^\top \right) \tag{2.4}$$

Similarly, to calculate the sum of row j in $\mathbb{D}'_N$:

$$\mathbb{I} = \sum_k d'_{jk} = \sum_{l=j}^{j+n-1} \mathbb{D}'_l$$

This is the inverse of the expand operation executed by $\mathbb{E}$, and once again we can use that same matrix to compute all edge inputs simultaneously:

$$\underset{d \times n}{\mathbb{I}} = \frac{1}{n} \left(\mathbb{D}' \times \mathbb{E}^\top \right) \tag{2.5}$$

These operations allow us to avoid ever transposing $\mathbb{D}'$, instead allowing us to work directly on it.

For both $\mathbb{I}$ and $\mathbb{O}$, we still need to pair the vectors within with the appropriate vector in $\mathbb{D}'$. To accomplish this, we must expand both matrices to $(d \times n^2)$.

For some vector $\mathbb{I}_x$, we wish to pair it with all vectors $d_{kx} \in \mathbb{D}'_N \mid k \in [0, n)$. In terms of $\mathbb{D}'$, these vectors map to $\mathbb{D}'_{kn+x}$; i.e., we wish to create a matrix into which we distribute a given vector in $\mathbb{I}$ n times, n columns apart. Once again, we already have a matrix specifically capable of this operation: $\mathbb{T}$. Similarly, we wish to pair any given vector $\mathbb{O}_x$ with all vectors $d_{xk} \in \mathbb{D}'_N \mid k \in [0, n)$, which correspond with $\mathbb{D}'_{xn+k}$: for each vector in $\mathbb{O}$, we broadcast it into a $(d \times n^2)$ matrix such that it repeats n times. Yet again, an established matrix will complete this task: $\mathbb{E}$. Thus our intermediary steps for this layer are quite similar to **(2.1b)**:

$$\underset{d\times n^2}{\mathbb{I}_{\mathbb{D}}} = \underset{d\times d}{\mathbb{W}'_{in}} \times ((\mathbb{I} \times \mathbb{T}) \odot \mathbb{D}') \qquad \underset{d\times n^2}{\mathbb{O}_{\mathbb{D}}} = \underset{d\times d}{\mathbb{W}'_{out}} \times ((\mathbb{O} \times \mathbb{E}) \odot \mathbb{D}') \tag{2.6a}$$

And we arrive at the matrix expression of the locality layer:

$$\underset{d\times n^2}{\mathbb{D}''} = \underset{d\times 2d}{\mathbb{W}'_{tot}} \times \begin{pmatrix} \mathbb{I}_{\mathbb{D}} \\ \hline \mathbb{O}_{\mathbb{D}} \end{pmatrix} + \left(\underset{d\times 1}{\mathbb{B}'} \times \underset{1\times n^2}{\mathbb{1}} \right) \tag{2.6b}$$

Final Layer

The operation for the matrix version of the final layer is effectively the same as **(2.2)**:

$$\underset{1\times n^2}{\mathbb{D}^f} = \underset{1\times d}{\mathbb{W}^f} \times \underset{d\times n^2}{\mathbb{D}''} \tag{2.7}$$

2.2.4 Benchmark Model

The model we provide as a benchmark mimics our model in its first **(2.3)** and final **(2.7)** layers. The difference lies in the second layer: where in **(2.6)** we

perform a variety of transforms to incorporate locality data, here this layer is entirely defined by the following equation:

$$\underset{d\times n^2}{\mathbb{D}''} = \underset{d\times d}{\mathbb{W}'} \times \underset{d\times n^2}{\mathbb{D}'} + \underset{d\times 1}{\mathbb{B}'} \times \underset{1\times n^2}{\mathbb{1}} \tag{2.8}$$

2.2.5 n-independence

Trainable Values

Between all of the operations defined in 2.2.3 (and equivalently in 2.2.2), the following matrices are the only values over which the optimizer can perform gradient descent:

First Layer

$\underset{d\times 2b}{\mathbb{W}}$: weight matrix used to merge columns of input data

$\underset{d\times 1}{\mathbb{B}}$: bias vector added to every $\mathbb{D}'_k \mid k \in [0, n)$.

Locality Layer

$\underset{d\times d}{\mathbb{W}'_{in}}$: weight matrix used to process data entering an edge

$\underset{d\times d}{\mathbb{W}'_{out}}$: weight matrix used to process data exiting an edge

$\underset{d\times 2d}{\mathbb{W}'_{tot}}$: weight matrix used to merge the data produced by $\underset{d\times d}{\mathbb{W}'_{out}}$ and $\underset{d\times d}{\mathbb{W}'_{in}}$

$\underset{d\times 1}{\mathbb{B}'}$: bias vector added to every $\mathbb{D}''_k \mid k \in [0, n)$.

Final Layer

$\underset{1\times d}{\mathbb{W}^f}$: weight matrix used to collapse all n^2 vectors into n^2 scalars.

Benchmark Model Our benchmark model shares first and final layer structures with the overall model, leading to its having the same optimizable parameters for those layers. Its second layer retains the bias vector $\mathbb{B}'$, but that and a single $(d \times d)$ matrix $\mathbb{W}'$ are the only optimizable values.

Implications

As noted previously, none of these matrices are dependent on n. Furthermore, even in the matrix model (2.2.3), the weight matrices operate individually on each ij vector, and the same bias is added to each vector. Because the network is not provided any trainable n-scale values, all calculation and training is done per node pair. This obviates the typical neural network problem of overfitting to its training dataset to the point it simply memorizes appropriate outputs.[7] Additionally, this allows for application of a trained model to data produced by generators of a different size than those used to train the model. Because our model operates entirely on local graph features, the only requirement for such an application is that the training data contain a set of features also representative of the new data.

[7] See 4.1

3 Training

3.1 Activation Functions

3.1.1 Initial & Convolutional Layers

At the end of each transition, an elementwise activation function is applied following completion of all computations. For all but the final layer, that function is ReLU[1], defined in figure 3.1.

$$relu(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

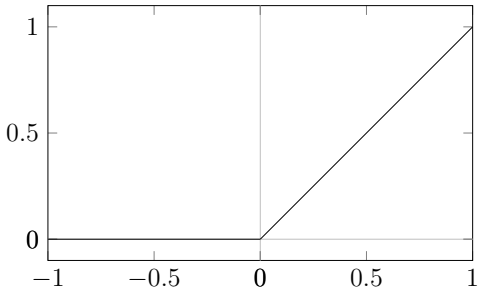


Figure 3.1: ReLU function definition and graph

Alternative Activations

In addition to ReLU, we considered a sigmoid activation function, as in Figure 3.2.

[1]NEEDS CITATION

$$S(x) = \frac{1}{1 + exp(-x)}$$

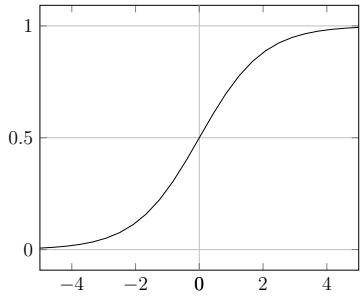


Figure 3.2: Sigmoid function definition and graph

However, this function requires that the network have extremely well-tuned matrices in order to produce values near zero, and, given our binary generator networks, it presents unecessary training difficulty, leading us to use ReLU.

3.1.2 Final Layer

Additionally, ReLU's preservation of positive values and elimination of negative work in concert with the activation function of the final layer, hyperbolic tangent (Figure 3.3). The clipping of negative values to 0 in previous layers of the network allows greater imprecision in the penultimate layers in order to predict a 0 in the output adjacency matrix: rather than needing to fine tune the filters to produce exactly 0 for nonexistent connections, the model need only drive the values for such neuron pairs into the negatives, and let the application of ReLU correct.

Similarly, the final layer *tanh* allows the network to drive weights for probable connections far into the positives, with the activation function ultimately truncating them to 1.

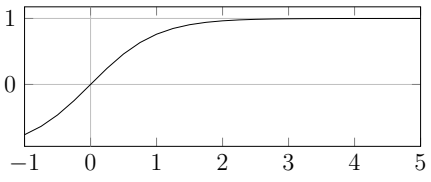


Figure 3.3: Graph of $y = \tanh(x)$

3.2 Loss & Optimization

In a nutshell, backpropagation via gradient descent is a method for training neural networks by calculating the extent to which each value in a particular layer is responsible for the overall network error on a single data point or batch, then correcting that value by an amount commensurate to its error and overall learning rate. This process operates from the final layer back to the first, hence 'backpropagation'.

In order to effectively descend the gradient, a network needs a function defining error from the desired output and an algorithm for applying gradient descent based on that error and a specified learning rate.

The loss function must provide useful values to the optimizer in order to allow effective gradient descent towards the goal, and the optimizer must adjust the network fast enough to converge to the target while avoiding converging to a suboptimal solution. As the network gets closer to an optimal state, adjusting at the same rate as at the start of training will almost invariably overshoot the desired configuration. Due to this, the optimizer must dynamically modify the extent to which it adjusts the network as training goes on.

3.2.1 Loss Function

We define a basic custom loss function in order to better fit the outputs we expect to see.

For final model output $\mathbb{O}$ and target $\mathbb{T}$, we take the sum squared difference, S, of the two vectors and the sum over $\mathbb{T}$, S_T, **(3.1a)**, and divide these two values to achieve loss L.[2]

$$S = \sum_i (\mathbb{O}_i - \mathbb{T}_i)^2 = \sum_i [(\mathbb{O} - \mathbb{T})_i)]^2 \qquad S_T = \sum_i \mathbb{T}_i \qquad \textbf{(3.1a)}$$

$$L = \frac{S}{S_T} \tag{3.1b}$$

Thus, rather than scale loss with the number of total possible connections (n^2) as with a mean squared error, we scale our loss with the number of actual connections in the true adjacency matrix, keeping the loss values somewhat higher in the early stages of training, yet still falling to levels comparable to that of MSE as the model learns to predict appropriately.

Effects

Consider a model analyzing data from a 3-neuron generator with an adjacency matrix as given in Figure 3.4, and suppose that its output is a vector containing two correct values and one wrong value. Then our parameters for determining loss by way of **(3.1)** are as follows:

	0	1	2
0	0	0	0
1	1	0	0
2	1	1	0

Figure 3.4: Example adjacency matrix

$$\mathbb{O} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{bmatrix}$$
$$\mathbb{T} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \end{bmatrix}$$
$$(\mathbb{O} - \mathbb{T})^2 = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 \end{bmatrix}$$
$$S = \sum_i (\mathbb{O}_i - \mathbb{T}_i)^2 = 2.0$$
$$S_T = \sum_i \mathbb{T}_i = 3.0$$

And our loss is finally determined:

$$L = \frac{S}{S_T} = \frac{2.0}{3.0} = .\overline{6}$$

[2]Recall from 2.1.2 that the targets $\mathbb{T}$ given to the model are the flattend generator adjacency matrix; dimensionality $(1 \times n^2)$.

Thus, our loss function 'punishes' the network equally for false positives and false negatives: due to the squared difference, a 1 where there should be a 0 adds the same loss as a 0 where there should be a one. This is perhaps not the ideal method; see The value produced for each input/target pair is then passed to the optimizer.

3.2.2 Optimizer Function

We used the Adam optimizer[3] as provided by TensorFlow, providing different initial learning rates per dataset. Those values were arrived at via experimentation. After initializing the optimizer, it is passed the loss at each step and performs gradient descent on the trainable matrices.

Adam adjusts its learning rate as time goes on, according to the following equation, where β_n^t indicates exponentiation by t and lr denotes learning rate:

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$lr_t = lr_{init} \times \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$$

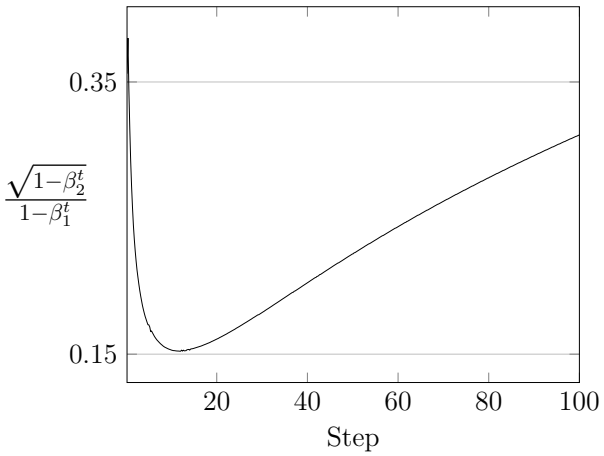


Figure 3.5: Adam decay function over 100 steps. Converges asymptotically to 1.

[3]Citation

3.3 Datasets

3.4 Matrices

3.4.1 Initialization

Initially, we seeded our matrices with random values from a normal distribution of standard deviation 1.0 and mean 0, using the TensorFlow implementation of `tf.random_normal(<dimensions>)`. Due, however, to the cumulative nature of our matrix operations (in the locality layer, for instance, there are three separate multiplications (2.2.3)), we found that the values of the outputs were so high or low as to render the model somewhat random in its convergence, or lack thereof.

We found that by reducing the standard deviation of our distributions to 0.25, we can ensure that most, if not all, training runs of our model converge. If raised higher, models trained on complex generator networks may consistently fail to converge, and a lower value tends to lead to convergence on non-optimal solutions, such as prediciting all zeroes.

3.4.2 Locality Layer Operations

As discussed in 2.2.2, a different method for integrating inputs and outputs to a given edge was originally considered. If $\mathbb{I}$ is the average input vector for some edge d_{ij}, and $\mathbb{O}$ is the average output vector, then the original operations went as follows:

$$\underset{d\times 1}{\mathbb{I}_{\mathbb{D}}} = \underset{d\times 2d}{\mathbb{W}'_{in}} \times \begin{pmatrix} \mathbb{I} \\ d'_{ij} \end{pmatrix} \qquad \underset{d\times 1}{\mathbb{O}_{\mathbb{D}}} = \underset{d\times 2d}{\mathbb{W}'_{out}} \times \begin{pmatrix} d'_{ij} \\ \mathbb{O} \end{pmatrix} \tag{3.2a}$$

$$d''_{ij} = \mathbb{I}_{\mathbb{D}} + \mathbb{O}_{\mathbb{D}} \tag{3.2b}$$

This was suboptimal for a variety of reasons. The addition of the two vectors in the final step implied that both inputs and outputs were of exactly equal value in determining the existence of an edge, and even further that, for any index into those two vectors, the values at that index would be usefully comparable in some way.

Beyond this, the integration of locality data in this format seemed to require careful tuning, and gradient descent did not work well with this setup. Specifically, at an initial network state, the first layer has not been optimized to provide useful data targeted at the second layer summations that produce $\mathbb{I}$ and $\mathbb{O}$. However, the only reason for the network to trend towards this type of data shaping in the first layer would be an observed decrease in loss. While doubtless possible, it seems a more attractive (loss optimizing) option appears: zero the left side of $\mathbb{W}'_{in}$, and the right of $\mathbb{W}'_{out}$. As the noisy locality data is removed from the system, the loss decreases, and eventually the network arrives at a somewhat remarkable state: the halves of the weight matrices that remain in use converge to the same values, operating as they are on the same data.

For these reasons, we opted for the implementation described in 2.2.2, in which we force the integration of locality data into the model's calculations via entrywise multiplication, and provide an optimizable matrix for combining input and output data, allowing the network to learn which parts are most important.

3.5 Hyperparameter Optimization

3.5.1 Batch Size

'Batching' refers to the process of assembling a set of items from the training data and passing them through the network in parallel, then optimizing over the resulting losses simultaneously. This greatly speeds computation speed by removing the costly optimization operation from each step. We found that 32 units per batch was an effective number, offering high training speeds with relatively stable loss curves.

4 Results

4.1 Overfitting

As discussed in 2.1.3 and 2.2.5, the unique structure of our model prevents it from overfitting to a particular generator topology, allowing us to create a single generator containing connections representative of the types of data we expect to analyze with the trained model. We demonstrate this aspect of our architecture in two test cases: by training models on an empty dataset paired with one adjacency matrix throughout, and training with a random dataset paired with that same adjacency matrix.

b (timesteps)	8
d	5
Batch size	32
Training steps	20000
Learning rate	.0005
Training samples	18000
Validation samples	4500

Figure 4.1: Training parameters for null hypothesis networks

4.1.1 Empty Data

We ran a combined 100 training sessions of the benchmark model and our convolutional model, with parameters as defined in Figure 4.1, on a dataset whose inputs contained only zeroes and whose target was the adjacency matrix in Figure 4.4. For both models, exactly two losses and corresponding

outputs repeatedly occurred (Figure 4.2), with the models demonstrating a total inability to memorize the target data.

	0	1	2
0	.3	.3	.3
1	.3	.3	.3
2	.3	.3	.3

(a) loss: $0.\overline{6}$

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0

(b) loss: 1.0

Figure 4.2: Predictions and losses when training on an empty dataset

4.1.2 Random Data

For this trial, all model parameters were identical to those in 4.1.1. In this case, however, the data fed into the network consisted of raster plots whose items had been randomly assigned to 0 or 1. While the results were somewhat less consistent, over the course of 100 training sessions, the models that were able to converge to a minimum loss predicted the matrix in Figure 4.3 the overwhelming majority of the time.

	0	1	2
0	0	.5	.5
1	.5	0	.5
2	.5	.5	0

Figure 4.3: Average prediction for random data. loss: 0.5

4.1.3 Analysis

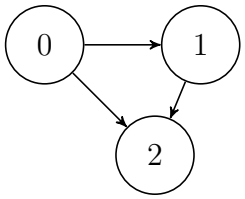
While the results of 4.1.2 are at first confusing, given the per edge architecture of our model, this result is not particularly surprising: in the first layer transition, every spike vector is compared against every other spike vector, including itself. Thus the model was in fact able to learn one feature, self loops, and, since loss decreased for driving such connections to 0, that they do not exist.

For the remainder of the potential connections, the model, lacking any sort of way to distinguish between them, found an equilibrium value that, when applied to the remaining connectinos, minimized loss. Note that both uniformly increasing or decreasing the nonzero weights in Figure 4.3 increases loss.

The same is true of the results in 4.1.1, with the output in Figure 4.2b particularly illustrative of the problem of entropy traps in neural networks. For models that converged to this output, the initial seeding of the weight and bias matrices was such that the fastest decreases in loss were found by adjusting trainable values to produce an empty matrix. Once there, uniformly increasing the output values would initially increase the loss, preventing the network from pushing upward and eventually reaching the lower loss state of Figure 4.2a.

4.2 3-neuron generator

We now consider a generator network consisting of three nodes connected as in Figure 4.4. All weights are binary, and a spike rate of .25 was used.[1]



	0	1	2
0	0	0	0
1	1	0	0
2	1	1	0

Figure 4.4: Network structure and adjacency matrix of the generator. (Reproduced from Figure 2.1)

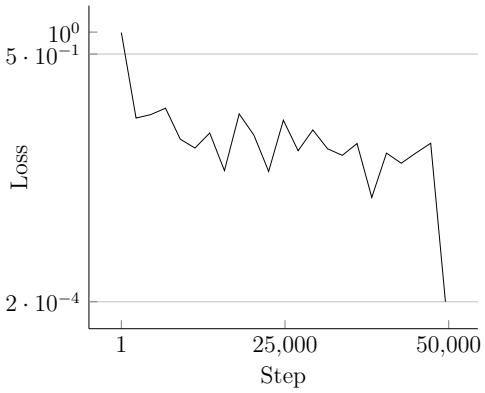
Reconstructing this simplified graph allows us to demonstrate that our con-

[1]SEE APPENDIX for information on spike rates

volutional approach is capable of reconstruction. Furthermore, the small generator size requires few timesteps and a small interlayer featurespace; i.e., $b, d < 10$. This results in a relatively simple set of transitions, allowing us to explore and understand the inner workings of the network.

4.2.1 Example Model

In order to demonstrate the internal mechanics of our model, we trained on data produced by the generator given in Figure 4.4, with parameters as given in 4.1. In this example, small values of b and d were used in order to allow for better comprehension and visualization of the internal mechanics; the practical effect of this is that relatively small matrices were available for the model to optimize, making each value adjustment more impactful on output, and thus each training step more dramatic. These are acceptable limitations, however, insofar as they provide a more comprehensible model struture.



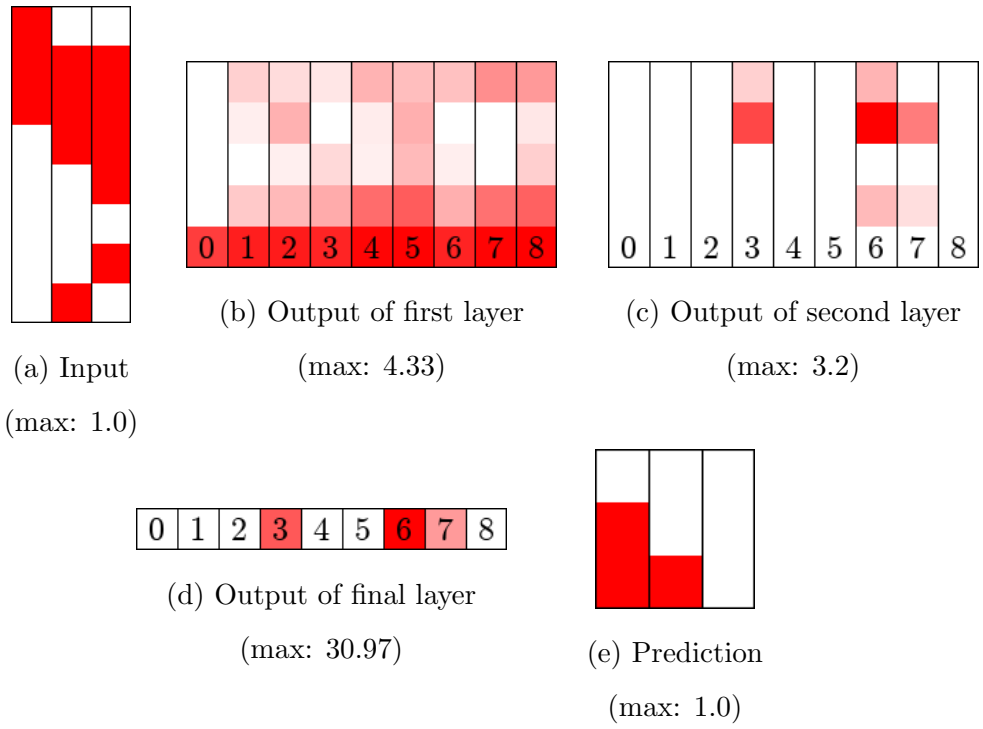
b (timesteps)	8
d	5
Batch size	32
Learning rate	.001
Training samples	36000
Validation samples	9000

Table 4.1: Loss & parameters for model 4.2.1. The loss here is choppy, due perhaps to aggressive optimization by Adam. [2]

[2] See 3.2.2

4.2.2 Trained Network Operation

Here, we will consider a single item of data as it travels through the model trained in 4.2.1.



(a) Input (max: 1.0)

(b) Output of first layer (max: 4.33)

(c) Output of second layer (max: 3.2)

(d) Output of final layer (max: 30.97)

(e) Prediction (max: 1.0)

Figure 4.5: Path of data through network. Transparency for each value is scaled relative to the maximum value found in the matrix.

In Figure 4.5, we demonstrate the progression of 4.5a through the trained model. The final layer, including activation[3], produces an n^2-vector which, when reshaped into an $(n \times n)$ matrix, is an exact match for the target, with all connections located and weighted appropriately.[4]

[3] See 3.1.2

[4] While $[1, 0]$ and $[2, 0]$ are predicted to be exactly 1.0, the precise value of $[2, 1]$ in the final prediction is 0.999999999957586, which we consider to be accurate enough.

Brief Analysis

Final Layer The final layer consists only of multiplying its weights (Figure 4.6) by the output from the locality layer, and it is thus relatively easy to intepret what the model has learned at this stage. As the first two values of of the weight matrix are strongly positive, we can conlude that the first two values in each vector in the output from the previous layer are highly important in the determination of connection presence, with some weight also placed on the fourth item.

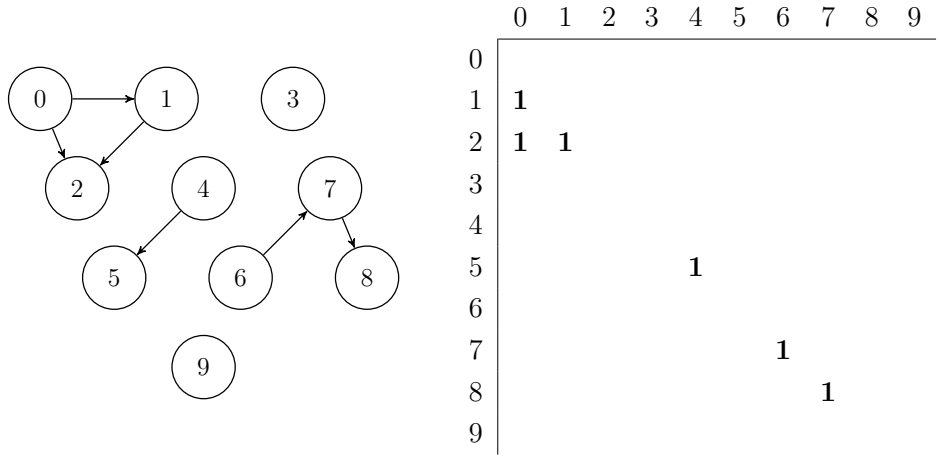
Figure 4.6: Final weights
(max: 7.31)

Locality Layer Functionality Note that, following the locality layer (4.5c), the model has located the existent connections: if we transpose 4.5c from $(d \times n^2)$ to $(n \times n \times d)$, as in Figure 2.4, the columns with high values, 3, 6, and 7, correspond with d-vectors $[1, 0]$, $[2, 0]$, and $[2, 1]$, respectively. These tuples each correspond with a connection present in the adjacency matrix (Figure 2.1) the model is trying to predict.

4.3 Higher-order Datasets

Because a 3-node generator not does contain much in terms of locality, we created a graph structure containing slightly more complex relationships to benchmark our model on; that generator can be found in Figure 4.7.

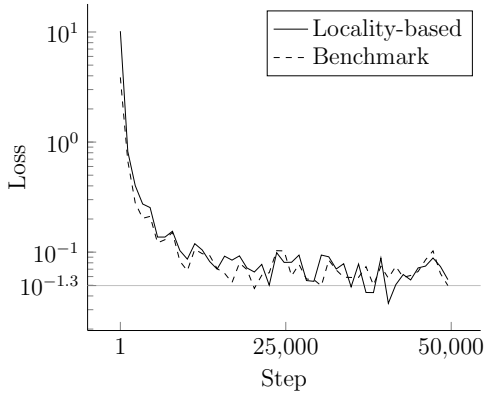
We trained 100 model/benchmark pairs on the data produced by this generator; the losses and parameters of the best-performing models of each type can be found in Figure 4.8. The results, in which the losses of both types of networks stayed extremely close, demonstrate that, while the locality-based



	0	1	2	3	4	5	6	7	8	9
0										
1	**1**									
2	**1**	**1**								
3										
4										
5					**1**					
6										
7							**1**			
8								**1**		
9										

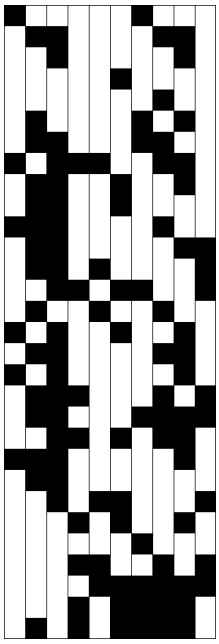
Figure 4.7: Ten neuron generator and adjacency matrix. For purposes of clarity, all zero values in the matrix have been omitted.

approach is able to reconstruct networks, it does not offer substantive improvement over the much more straightforward benchmark model, at least in the cases that we have considered. An example run can be found in Figure 4.9.

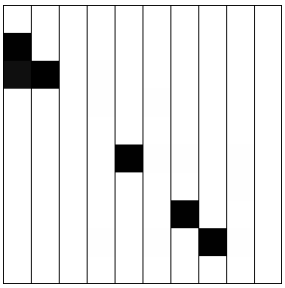


b (timesteps)	30
d	40
Batch size	32
Learning rate	.001
Training samples	36000
Validation samples	9000

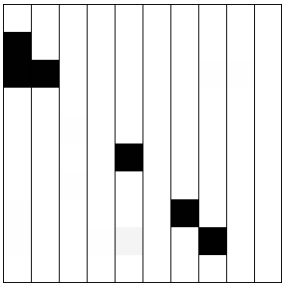
Figure 4.8: Loss & parameters for model trained on data from generator given in Figure 4.7



(a) Input



(b) Locality-based model prediction



(c) Benchmark model prediction

Figure 4.9: Example of data from generator defined in Figure 4.7, passed through the locality-based and benchmarks models.

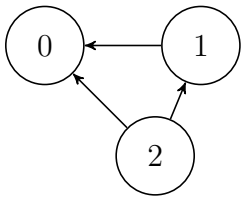
The same results were found with generators of various sizes and topologies, although none larger than that in Figure 4.7 were used. It is important, however, not to overstate the importance of this comparison,

4.4 Applicability Beyond Training Data

As described in 2.1.3, the fact that our model is trained on data produced by only one generator is of little consequence; due to its structure, the only information it can learn is relational, per neuron pair. Consider the following examples, in which data was produced from several generator networks and fed into the model described in 4.2:

TODO: add examples of input and output data to 3.2.1 and 3.2.2.

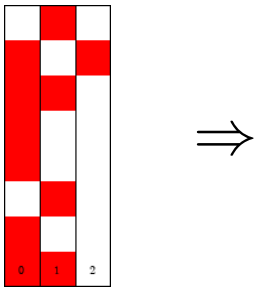
4.4.1 Inverted Network



	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Figure 4.10: Inverted version of Figure 4.4

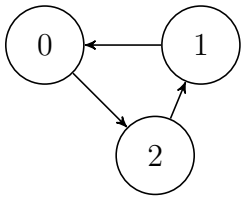
Despite being a complete inversion of the generator used to train the model in 4.2, reconstruction of this network is simple.



$\Rightarrow$

	0	1	2
0	.01	1.00	1.00
1	.01	.02	1.00
2	0	.02	.01

4.4.2 Cyclical Network



	0	1	2
0	0	1	0
1	0	0	1
2	1	0	0

Figure 4.11: Cyclical 3-neuron network

For a cyclical network, the situation is not quite so simple. Due to the perpetual propagation of spikes through the generator, additional random spiking can cause the input data to become an impenetrable mess. Tempering the spike rate to 0.05 produces workable data, but the results are neither so clean nor consistent as for terminating networks.

5 Discussion

As described in 4.3, in all cases tested, models equipped with our locality layer tended to stay very close to the benchmark models in loss, with a slight tendency towards higher loss. This tendency is explained by the simple presence of more values to optimize over. More to the point, we must return to the motivation behind incorporating locality into network reconstruction: we hope that our model will learn to recognize recurrent local structures in biological networks, and use that information to judge individual connection probability in the context of its neighbors. Two potential factors in our model's failure to manifest this behavior are apparent: data used, and specific locality algorithm design.

5.1 Data

An important part of analyzing the performance of our locality layer is to understand what we are looking for. In the cases tested, the locality-enabled model was not able to outstrip the benchmark model in terms of loss or predictive accuracy, but this likely speaks more to the type of data being used to train the networks than to the relative efficacy of either architecture. If three matrix multiplications are sufficient to reconstruct the structure of a network,

there is little need for a model to involve abstract concepts like locality, and, even if forced to do so, it's not clear that having such concepts available would contribute to more effective reconstruction. The ideal test dataset, then, would be one on which the benchmark model does not converge to an accurate prediction, allowing us to train a locality-enabled model and get some idea of how much useful information is actually added. Here, we outline some directions we could go in data generation.

5.1.1 Complex Neurons

As it stands, every generator we used to produce data consisted of binary connections and created binary outputs. There are clearly more accurate methods of simulating biological neural network activity, such as implementing Izhikevich neurons[1], or going as far as generating data with NEST[2]. However, while more complex neurons would probably encourage the model to look to locality for information, this alone would not suffice.

5.1.2 Larger, Structured Networks

A model being able to leverage its access to locality data to locate 2-simplices will not encounter any particular benefit from this ability if the generators it is tasked with reconstructing contain at most one such structure. Indeed, preliminary results suggest that a large gap opens between models considering locality and those not when the training data is generated from a large ($n \approx 50$) network seeded with recurring motifs.

[1] Cite

[2] citation

5.2 Potential Improvements to Locality

5.2.1 Layering

There may need to be more initial layers to provide useful data to the Locality layers. As it stands, the model structure requires that the first layer both compare the activities of neuron pairs and format the resulting data in such a manner that the locality-based layer can usefully include it in determining node existence. Adding at least one intermediary processing layer might allow the network to format the data going in to the locality layer in a more useful way. Merits further testing.

5.2.2 Algorithm

Detail issues with summing inputs and outputs, and propose alternative algorithm here. Note issues with implementation.

5.2.3 Loss

As described in 3.2.1, our custom loss function equally weights false positives and false negatives. Consider these cases:

1. Output 0.3; target 1.0: adds $(0.3 - 1.0)^2 = 0.49$ to the loss

2. Output 0.7; target 0.0: adds $(0.7 - 0.0)^2 = 0.49$ to the loss

Despite the equivalent loss contributions, the latter case is the less correct of the two: while guessing a weak connection where there is a strong one is not ideal, it is preferable to guessing a strong connection where there is none.

Thus our loss function might be modified to more strongly disincentive false positives.

5.2.4 Optimizer

I think it's in the paper about ELU or SELU that they use a ramping up learn rate and then decline. That might be ideal for the original, concatenation-based implemenation of the network, to avoid pushing it down the gradient too fast and zeroing out the convolutional matrix sections.

5.3 Potential Applications/Further Development

This shit is portable as fuck

Imagine an app(lication) that contains pretrained networks for different types of biological networks that researchers can then apply to their own data without having to interact at all with the nitty gritty bullshitty.

Community run database of trained networks as specific as submitters make them.

A Parameter Optimization Miscellanea

A.1 Data

A.1.1 Spike Rate Determination

As seen in section 4.4.2, oversaturated data hamstrings the ability of our model to perform accurate reconstructions. As the

B Model

B.1 Batched Architecture Calculations

In order to allow processing of many pieces of data at once, the matrix model defined in 2.2.3 was adapted to a batched format. Given input matrices of shape $(b \times n)$, the actual input to the model is now of shape $(batchSize \times b \times n)$. As previously discussed, iteration across lists or dimensions is not a computationally efficient option. Therefore we use `tf.einsum`, an implementation of Einstein Sums. This allows, for example, the multiplication of two matrices, one of dimension $(i \times j \times k)$, and the other of dimension $(h \times j)$. An appropriate function call might appear as `tf.einsum('hj,ijk->ihk', mat2, mat1)`. The result is equivalent to the iterative multiplication of the $(h \times j)$ matrix across all i, without the computational overhead of CPU involvement. Every matrix multiplication in our model is implemented using this functionality.