Neural Network Reconstruction via Graph Locality-Driven Machine Learning

Bard College

Hayden Sartoris

Contents

List of Figures

Abstract

A consistent 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-indepent 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

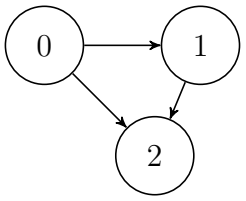
2.1 Data

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.

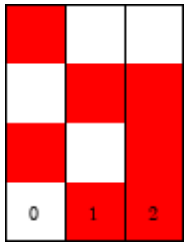
For a network accepting t timesteps of data from n neurons, the data fed into the network takes the following form:

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

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 2.2.4

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 APPENDIX SECTION.

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.

Convolutional 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.

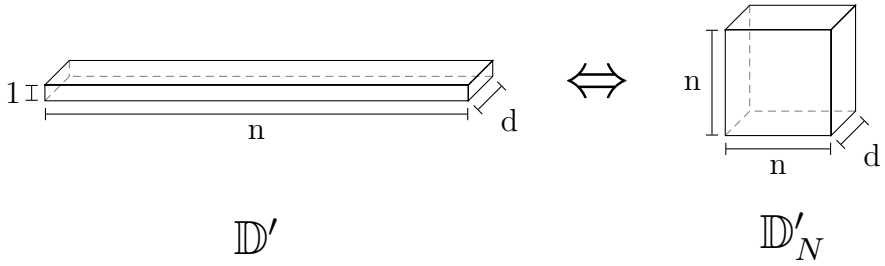


Figure 2.4: Relationship between $\mathbb{D}'$ and $\mathbb{D}'_N$.

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 activation

entrywise product computation. While we considered the same concatenation process for use in (**2.1b**), the apparent difficulty of integrating the calculated locality data into the prediction of ij led the model to rapidly adapt its weight matrices to ignore the locality portion of the data. For more discussion on this difficulty, see TRAINING SECTION.

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

First Layer

Convolutional Layer

Final Layer

2.2.4 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 trainable values:

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 d'_{ij}

Convolutional 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 d''_{ij}

Final Layer

$\underset{1\times d}{\mathbb{W}^{f}}$: weight matrix used to collapse previous outputs into one value

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.

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

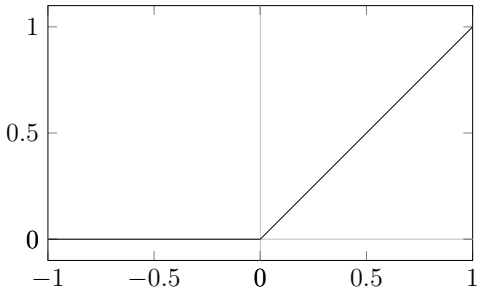


Figure 3.1: ReLU function definition and graph

Alternative Activations

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

[1]NEEDS CITATION

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

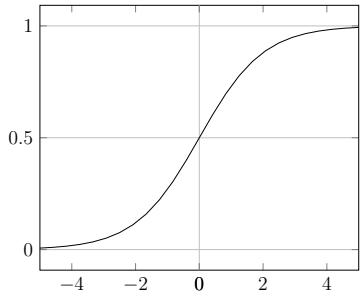


Figure 3.2: Sigmoid function definition and graph

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.

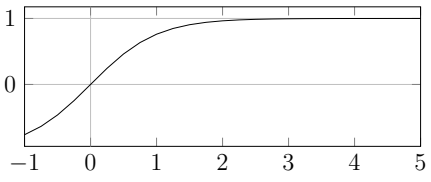


Figure 3.3: Graph of $y = \tanh(x)$

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. 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:

$$\begin{aligned} \beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ lr_t &= lr_{init} \times \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \end{aligned} \qquad \textbf{(3.2)}$$

3.3 Datasets

3.4 Matrix 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

[3]Citation

`tf.random_normal(<dimensions>)`. Due, however, to the cumulative nature of our matrix operations (in the convolutional layer, for instance, there are three separate multiplications 2.2.3), we found that the values

3.5 Hyperparameter Optimization

4 Results

4.1 Overfitting

As discussed in 2.1.3 and 2.2.4, 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 network

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 4.3. 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: $0.\overline{6}$

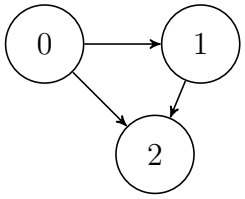
	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0

(b) loss: 1.0

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

4.2 3-neuron generator

We first consider a generator network consisting of three nodes connected as in Figure 4.3. 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

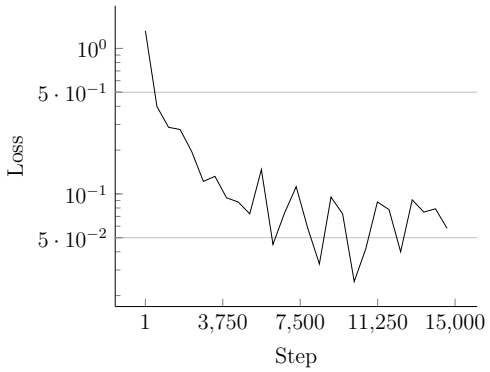
Figure 4.3: Network structure and adjacency matrix of the generator. (Reproduced from Figure 2.1)

Reconstructing this simplified graph allows us to demonstrate that our convolutional 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.

[1]SEE APPENDIX for information on spike rates

4.2.1 Example Model

The following data are pulled from a model trained on data produced by the generator in Figure 4.3. Figure 4.4 demonstrates the model's loss over time. In this example, b and d were pushed down in order to allow for better comprehension of the internal mechanics; the loss tends to converge more effectively and evenly given more computation power.



b (timesteps)	8
d	5
Batch size	32
Learning rate	.0005
Training samples	17984
Validation samples	4512

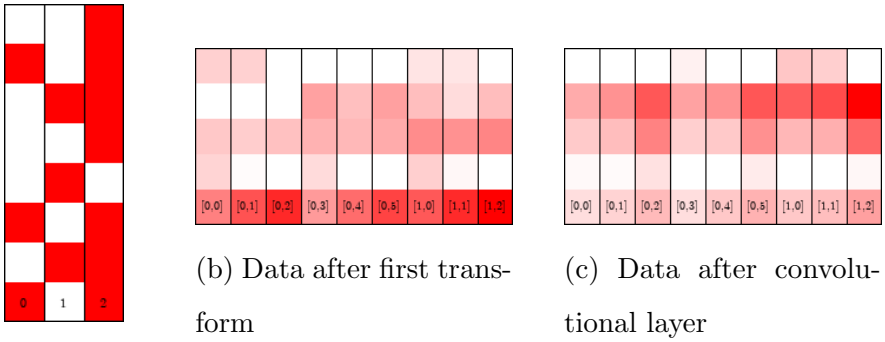
Figure 4.4: Training loss and parameters for model described in 4.2.1. The loss here is somewhat choppier than usual, due to the limited matrix size made available to the model.

Trained Network Operation

Here, we examine in brief the internal operation of the trained model over a single input. For a complete look through the procedure of reconstruction for this network, please see APPENDIX.

The last transformation of the network involves a matrix multiplication of the final layer weights[2] with the data in 4.5c. This produces the adjacency matrix found in figure 4.6.

[2]see appendix



(a) Input

(b) Data after first transform

(c) Data after convolutional layer

Figure 4.5: Path of data through network, up to final transform

	0	1	2
0	.02	.03	.01
1	.99	.01	-.01
2	1.0	1.0	.02

Figure 4.6: Output for input data in Figure 4.5a

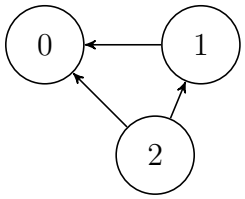
4.3 Applicability Beyond Training Data

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; i.e., 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.

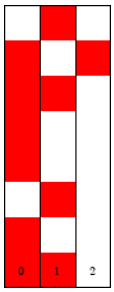
4.3.1 Inverted Network

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



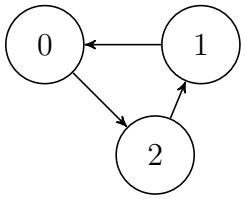
	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Figure 4.7: Inverted version of Figure 4.3



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

4.3.2 Cyclical Network



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

Figure 4.8: 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

5.1 Potential Improvements

5.1.1 Algorithm

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

5.1.2 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.

A Parameter Optimization Miscellanea

A.1 Data

A.1.1 Spike Rate Determination

As seen in section 4.3.2, oversaturated data hamstrings the ability of our model to perform accurate reconstructions. As the

B Model

B.1 Batched Architecture Calculations