Sproj – TODO

Bard College

Hayden Sartoris

April 2018

Contents

1 Background **5**

1.1 Biological Neural Networks 5

1.2 Graph Structures 5

1.2.1 Graph Locality 5

1.3 Convolutional Neural Networks 5

1.4 Concepts and Terms 5

1.4.1 Adjacency Matrices 5

2 Model **7**

2.1 Data 7

2.1.1 Generation 8

2.1.2 Restructuring 9

2.2 Architecture 10

2.2.1 First Transition 10

2.2.2 Convolutional Layer 11

3 Training **12**

3.1 Datasets 12

3.2 Matrix Initialization 12

3.3 Loss Optimization 12

3.3.1 Loss Function 12
3.3.2 Optimizer Function 12

4 Results **13**

4.1 3-neuron generator 13
4.1.1 Example Model 14
4.2 Applicability Beyond Training Data 15
4.2.1 Inverted Network 16
4.2.2 Cyclical Network 16

List of Figures

4.1 Network structure and adjacency matrix of the generator. . . . 13
4.2 Training loss and parameters for model described in 4.1.1 . . . 14
4.3 Path of data through network, up to final transform 15
4.4 Output for input data in figure 4.3a 15
4.5 Inverted version of figure 4.1 16
4.6 Cyclical 3-neuron network 16

1 Background

1.1 Biological Neural Networks

1.2 Graph Structures

1.2.1 Graph Locality

1.3 Convolutional Neural Networks

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 the neuron. We thereby arrive at a simple expression of the mechanics of adjacency matrices: weights in some row i define inputs to neuron i, weights in some column j define outputs from neuron j, and 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.

FIG: 3 neuron model & associated 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 follows, 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 $(n \times t)$.

APPENDIX BIT: have a section on how spike rates were determined for each network, as well as an example of producing data for the 3-neuron case.

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

[1]SEE APPENDIX

spit out the desired label, rendering it useless. However, due to the unique structure of our model, this sort of overfitting is impossible (SEE SOME ARCHITECTURE SECTION). 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 ultimately feed in to test.

2.1.2 Restructuring

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:

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

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 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 $\mathbb{W}_0$ of dimensionality $(d \times 2b)$.[2] Here d is an arbitrary number that provides the network enough space to maintain information about each neuron pair through the subsequent transitions; we arrive at this value experimentally.

$\mathbb{W}_0$ 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 $\mathbb{I}_x$ is the input column at x:

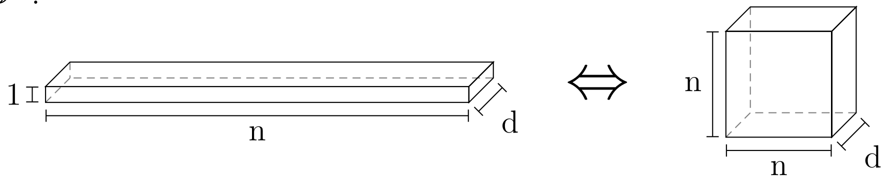
$$\forall i, j \mid i, j < n : \underset{d\times 1}{d_{ij}} = \underset{d\times 2b}{\mathbb{W}_0} \times \left(\frac{\mathbb{I}_i}{\mathbb{I}_j} \right)$$

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

[2]Recall that the time dimension fed into the network is characterized by b.

2.2.2 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 $\mathbb{D}^N$.



3 Training

3.1 Datasets

3.2 Matrix Initialization

3.3 Loss Optimization

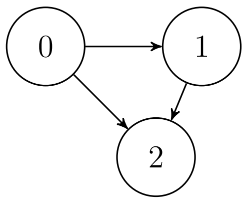
3.3.1 Loss Function

3.3.2 Optimizer Function

4 Results

4.1 3-neuron generator

We first consider a generator network consisting of three nodes connected as in FIGURE HERE (should contain visual structure and adjacency matrix). 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

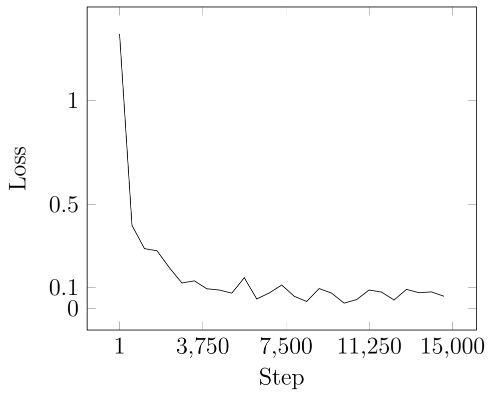
Figure 4.1: Network structure and adjacency matrix of the generator.

While reconstructing a graph comprising only three nodes is not much of a feat, this simplified case allows us to demonstrate that our convolutional approach is capable of reconstruction at all. Furthermore, the small network 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

4.1.1 Example Model

The following data are pulled from a model trained on data produced by the generator in figure 4.1. Figure 4.2 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

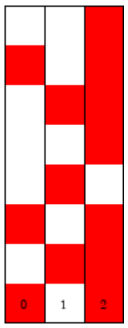
Figure 4.2: Training loss and parameters for model described in 4.1.1

Trained Network Operation

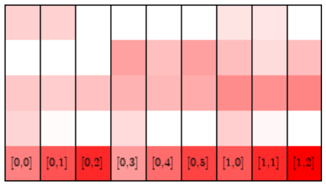
Here, we examine in brief the internal operation of the trained model over a single input. For a complete look through the procedure of reconstruction for this network, please see APPENDIX.

The last transformation of the network involves a matrix multiplication of the final layer weights[2] with the data in 4.3c. This produces the adjacency matrix found in figure 4.4.

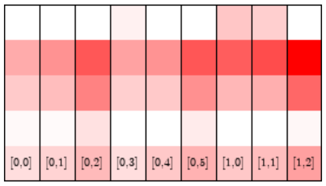
[2]see appendix



(a) Input



(b) Data after first transform



(c) Data after convolutional layer

Figure 4.3: Path of data through network, up to final transform

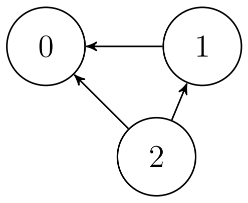
	0	1	2
0	.02	.03	.01
1	.99	.01	-.01
2	1.00	1.00	.02

Figure 4.4: Output for input data in figure 4.3a

4.2 Applicability Beyond Training Data

As described in 2.1.1, 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 4.1:

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

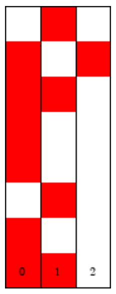


	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Figure 4.5: Inverted version of figure 4.1

4.2.1 Inverted Network

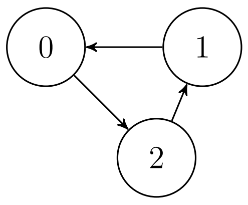
Despite being a complete inversion of the generator used to train the model in 4.1, reconstruction of this network is simple.



$\Rightarrow$

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

4.2.2 Cyclical Network



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

Figure 4.6: 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.