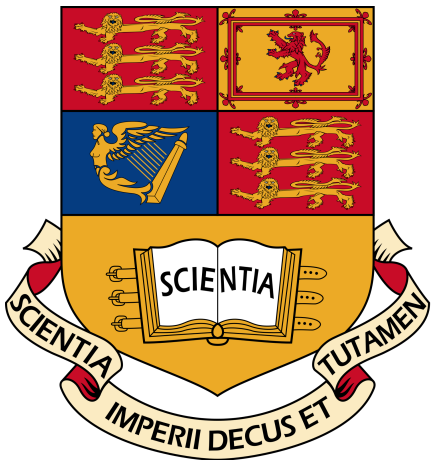
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017



Project Title:	**Estimating the Topology of Networks from Distributed Observations**
Student:	**Pranav Malhotra**
CID:	**00823617**
Course:	**EEE4**
Project Supervisor:	**Prof. Pier Luigi Dragotti**
Second Marker:	**Prof. Danilo P. Mandic**

Abstract

This project tackles the problem of inferring synaptic connections within biological neural networks from distributed observations of spiking activity. Rather than focusing on structural or functional connectivity, the emphasis of the project is to understand the cause and effect relationships between neurons in a network. The problem is solved by adapting NetRate, a maximum-likelihood estimator, originally developed to understand the spread of diseases and information in diffusion networks. Biological neural networks are simulated using Izhikevich's two-dimensional nonlinear dynamic model of a neuron.

Before inference can be performed, a probabilistic model of neural networks has to approximated. The probabilistic model obtained displays properties that make NetRate a suitable choice for neural network inference. Simulations with various network sizes show that the adapted algorithm accurately estimates directed neuronal connections within a network. This project provides an easy-to-use software package that generates a neural network, runs simulations, extracts spiking activity, infers the network and visualises the results obtained.

Acknowledgements

I would like to express my heartfelt gratitude to my project supervisor Dr. Pier Luigi Dragotti for his patience, guidance and supervision throughout the course of this project. I would also like to thank Ms Stephanie Reynolds for all her invaluable advice.

I would like to thank the dear friends that I have made at Imperial for being constant pillars of strength and motivation over the last four years. I wish them all the best in their future endeavours.

Last but not least, I would like to thank my brother and my parents for their unwavering love and support throughout my studies. Without them, I would not have had the opportunity to attain an outstanding and fulfilling education at Imperial College London.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

This project aims to tackle the problem of inferring synaptic connections within a biological neural network from distributed observations of spiking activity. In recent years, an immense amount of accurate data for a wide range of complex systems has been collated and studied. A key insight drawn from these studies is that complex systems share surprisingly similar macroscopic behaviour despite differing extensively in their unique element wise behaviour and element-to-element interaction mechanisms [2]. One common behaviour that appears across multiple complex systems is the small-world phenomenon. Complex systems often have clusters, cliques and sub-systems. Biological neural networks are amongst the wide range of complex systems that exhibit the small-world behaviour [3]. The small-world topology has been shown to exist in the neural network of *C. elegans* [2] and in the brain anatomy of macaques and in a cat's cortex [4].

Understanding biological neural network connectivity is extremely important as early studies have identified its use as a diagnostic marker. Brain networks parameters, more specifically, the clustering coefficient, derived from functional magnetic resonance imaging (fMRI) data are altered in patients with Schizophrenia or Alzheimer's disease [5]. In addition, brain networks have been used to study disconnection syndromes, an umbrella term for multiple neurological symptoms caused by the breakdown of communication pathways within the brain[1].

In addition to its use as a diagnostic marker, network connectivity can be used to understand the pathogenesis of neuronal disorders. Most neuronal disorders are highly heritable and thus,

[1]The breakdown of communication pathways is associated with damaged white matter (axons).

clinical studies of network connectivity can be used as endophenotypes that will give insight into the potential genetic risks of brain disorders. The clustering coefficients and average path lengths of functional brain networks[2] derived from EEG data were shown to have high heritability, a necessary prerequisite for their candidacy as disease endophenotypes [6].

[2]Functional connectivity models the statistical dependence between two neurons in a network; it is able to model dependence between two neurons that may be spatially distributed. Functional connectivity is contrasted with structural/anatomical connectivity which models physical synaptic connections between neurons. A more detailed analysis of different interpretations of connectivity is provided in section 2.1.

Chapter 2

Background

2.1 Definition of Connectivity

In the context of brain networks, the term connectivity is highly ambiguous [7]. The term is used to refer to one of three things: structural connectivity, functional connectivity or effective connectivity. Structural connectivity refers to the physical synaptic connections between individual neurons. It can also refer to physical connections between different parts of the brain. The structural connectivity of biological neural networks can currently only be studied using invasive imaging techniques [8]: invasive methods are the only way to unambiguously confirm the presence of synaptic connections.

Functional connectivity on the other hand is inferred using symmetrical measures of statistical association, most commonly correlation [9], coherence [10] and mutual information [11]. Due to their symmetry, these statistical tools can only be used to infer undirected connections. Undirected connectivity is assumed to be possible between all neurons in a network. The statistical tools used for inference do not take into account any underlying structural/anatomical model or directional effects that might make a certain connection impossible or unidirectional.

Effective connectivity extends the framework of functional connectivity by incorporating directional effects. Directional effects are described by studying causal relationships. It is no surprise then that techniques used to infer effective connectivity include the Granger Causality test [12] and Dynamic Causal Models [13]. This project will aim to infer effective connectivity between neurons

within a network. **Connectivity will be loosely used to mean effective connectivity for the rest of this report.**

2.2 Spiking Neuron Models

2.2.1 Leaky Integrate-and-Fire Neuron

To preserve the biological significance of the inference problem that this project tries to solve, it is important to use an appropriate spiking model. The most commonly used spiking model in computational neuroscience is the leaky integrate-and-fire neuron. The general form of the leaky integrate-and-fire model presented in [14] is replicated below. The differential equation describes the evolution of a neuron's membrane potential, $v(t)$:

$$C_m \frac{dv(t)}{dt} = I_{leak}(t) + I_s(t) + I_{inj}(t)$$

where C_m is the membrane capacitance, $I_{leak}(t)$ is the current due to passive leak of the membrane, $I_s(t)$ is the current from synaptic inputs and $I_{inj}(t)$ is the current injected into the neuron by external stimuli. The model has other parameters such as τ_m, which is the passive membrane time constant and V_{th} and V_{reset} which represent the threshold and reset voltages respectively. The key behind the widespread use of the leaky integrate-and-fire neuron is two-fold: it is linear and it is one-dimensional. The linear nature of the differential equation allows for an analytical solution to be found while the one-dimensional nature of the dynamics makes running simulations computationally simple. However, the leaky integrate-and-fire neuron is unable to replicate phasic spiking or bursting behaviour which are fundamental features observed in biological neurons [15].

2.2.2 Hodgkin-Huxley Model

The Hodgkin-Huxley neuron [16] is another extremely important model in computational neuroscience. Experiments performed on the giant axon of a squid led to the discovery of three unique ion currents that characterize the membrane potential of neurons:

1. A sodium (Na) ion-based current.
2. A potassium (K) ion-based current.
3. A leak current consisting mainly of chlorine (Cl) ions.

The lipid bilayer[1] also generates a difference in potential between the inside of the axonal membrane and the surrounding area. This behaviour is described with a capacitor. By the law of conservation of electric charges, the total current through the cell membrane is given by (2.1), where $V_m(t)$ describes the membrane potential, C_m describes the membrane capacitance, g_K, g_{Na} and g_l represent the potassium, sodium and leak conductances respectively and V_K, V_{Na} and V_l represent the potassium, sodium and leak reversal potentials respectively.

$$I(t) = C_m \frac{dV_m(t)}{dt} + \bar{g}_K n^4(t)[V_m(t) - V_K] + \bar{g}_{Na} m^3(t) h(t)[V_m(t) - V_{Na}] + \bar{g}_l[V_m(t) - V_l] \quad (2.1)$$

The complexity of the model is further increased by the fact that $n(t)$, $m(t)$ and $h(t)$ are time-varying variables, taking values between 0 and 1, that are related to the current channel parameters. The differential equations that describe the evolution of these variables are presented below:

$$\begin{aligned}
\frac{dn(t)}{dt} &= \alpha_n V_m(t)[1 - n(t)] - \beta_n V_m(t) n(t) \\
\frac{dm(t)}{dt} &= \alpha_m V_m(t)[1 - m(t)] - \beta_m V_m(t) m(t) \\
\frac{dh(t)}{dt} &= \alpha_h V_m(t)[1 - h(t)] - \beta_h V_m(t) h(t)
\end{aligned}$$

The Hodgkin-Huxley model consists of a system of nonlinear differential equations with four state variables, namely $V_m(t)$, $n(t)$, $m(t)$ and $h(t)$. The strong nonlinearity and large dimensionality make this model prohibitively complex for this project.

2.2.3 Izhikevich Model

All simulations run for this project will utilise the neuron model formulated in (2.2) and (2.3). The neuron model, formulated by Izhikevich [1], is a two-dimensional set of nonlinear differential equations with four parameters. Izhikevich neurons, with the right tuning parameters, are able to replicate most of the prominent spiking behaviours exhibited by biological neurons. The model has two state variables, the membrane potential, $v(t)$, and the membrane recovery variable, $u(t)$, and four static parameters a, b, c and d.

[1]The lipid bilayer is a thin polar membrane that forms a continuous barrier around all cells. The membrane is made up of lipid molecules.

$$\frac{dv(t)}{dt} = 0.04v^2(t) + 5v(t) + 140 - u(t) + I \tag{2.2}$$

$$\frac{du(t)}{dt} = a(bv(t) - u(t)) \tag{2.3}$$

The recovery of the neuron after an action potential is described below:

$$\text{if } v(t) \geq 30 \text{ mV, then } \begin{cases} v(t) \leftarrow c \\ u(t) \leftarrow u + d \end{cases} \tag{2.4}$$

The spike data obtained from simulating a network consisting of 1000 Izhikevich neurons is presented in figure 2.1 in the form of a raster plot. The simulation captures cortical-like asynchronous dynamics; neurons fire Poisson spike trains with mean firing rates around 8Hz [1]. The simulation also reveals the presence of alpha and gamma waves that are associated with the mammalian cortex in the awake state.

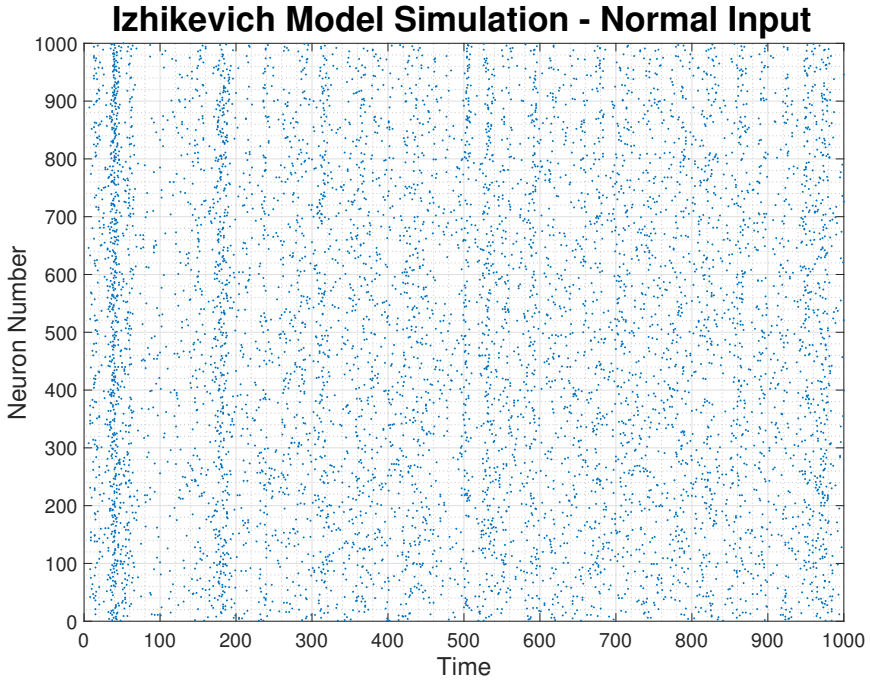


Figure 2.1: Simulation of a 1000 neuron network using Izhikevich's two-dimensional nonlinear spiking model. Simulation is originally found in [1]

All future simulations will similarly be depicted in the form of raster plots: the raster plot provides the clearest and most concise visualisation of firing events. The primary form of data that this project will utilise are firing events. They represent the distributed observations that are used to infer connectivity. A detailed explanation of connectivity between neurons in a network and how the spiking activity of pre-synaptic[2] neurons impacts post-synaptic neurons is provided in section 3.1.

2.3 Overview of Existing Solutions for the Network Inference Problem

In 1996, Y. Vardi [17] coined the term **network tomography** to describe his areas of research. Vardi's work involved estimating node-to-node traffic intensity from repeated measurements of traffic within the link-layer of a computer network. He coined the term due to the similarity between the network inference problem and medical tomography, a technique used to understand the internal characteristics of an object from external observations. The birth of network tomography can be largely attributed to the explosive growth of the internet. Although early research focused primarily on inferring link-layer topologies, the field of network tomography has greatly diversified in its applications in recent years. In addition, research has shifted away from intricate node-based analysis towards the formulation of generic models that describe global behaviour. Neuroscience has greatly benefited from the methodologies and techniques developed to solve network tomography problems; the same methods and techniques are now being applied to infer biological neural networks.

To infer effective connectivity, many algorithms utilise statistical measures such as the Granger Causality test [18]. Used in this context, the Granger Causality test has two main shortcomings. Firstly, the test only produces positive values. This is not ideal as neurons are known to be both excitatory and inhibitory; the causality test is incapable of capturing the inhibitory behaviour. Secondly, causality scores vary with the length of the signals that are compared and thus scores are only useful for relative comparisons and cannot be used determine absolute cause and effect relationships. For these reasons, the Granger Causality test fails to correctly identify networks with strong oscillatory components [19].

[2]Within the nervous system, a synapse is a structure that allows nerve cells (neurons) to pass electrical or chemical signals to other neurons.

Algorithms developed to solve biological network inference problems often implement variations of the Generalized Linear Model (GLM). Modelling spike trains as point processes and using coupling functions composed of spline basis functions led to the successful reconstruction of a physiological circuit [20]. Although effective, if implemented properly, GLMs have high computational costs. In addition, biases are introduced if the model used to generate spike data does not match the model used in the GLM.

Literature in the field also presented a novel algorithm that adapts the perceptron learning rule for the purpose of neural network inference [21]. The algorithm determines weights for each node-to-node interaction. The relative weights are then clustered into three groups, namely no connection, an inhibitory connection and an excitatory connection, using the K-means algorithm. The adapted perceptron only infers ternary values, -1, 0 and 1, whereas the network inference algorithm that this project adapts is able to infer continuous weights between 0 and 1.

2.4 NetRate Algorithm

2.4.1 Diffusion Processes

This project will adapt the NetRate algorithm [22], developed by Rodriguez, to infer directed connections in brain networks. The algorithm was first proposed as a general purpose tool to study the temporal dynamics within a diffusion network. Many real-world interactions can be modelled as diffusion processes: bloggers sharing information after reading it on another website, a person catching a virus by interacting with someone who is carrying it, or a person buying a product after noticing someone he or she knows who has the same product. **The key element underlying the diffusion process in all three scenarios is a node-to-node interaction mechanism that propagates an observable artefact from the source node to the target node.** Moreover, we will soon learn that the rate of diffusion is directly dependent on the strength of a directed connection between the source node and the target node. The spread of diseases within a population will be used as an illustrative example of a diffusion process.

NetRate is built upon a framework that describes the spatiotemporal structures that generate diffusion processes. Four fundamental assumptions are made. They are:

1. The diffusion process occurs over a static directed network. The organisational structure of the network is time-invariant.

2. The observable artefacts used for network inference are binary. A person has the disease or does not have the disease.

3. Individual node-to-node interactions are independent. The probability of person A being infected by person B does not depend on his or her probability of being infected by person C.

4. The exact times of all observable artefacts within the system are known. It is immediately known when a person contracts the disease.

2.4.2 Mathematical Constructs

The algorithm uses a probabilistic approach to model directed node-to-node interactions. More specifically, the model uses a pairwise transmission likelihood that defines the chance of a person being infected by someone else. $f(t_i|t_j, \alpha_{j,i})$ **is defined as the conditional likelihood of person** i **being infected at time** t_i **given that person** j **was infected at time** t_j**.** The conditional likelihood is parameterised by $\alpha_{j,i}$ which is the directed pairwise transmission rate: the value of $\alpha_{j,i}$ will directly control the rate of diffusion. It is imperative to realise that pairwise transmission likelihoods are associated with connections rather than with nodes. Each node within the network will have multiple connections and each connection will have a corresponding $\alpha_{j,i}$ value. It is exactly these parameterisation constants, $\alpha_{j,i}$, that NetRate infers. To perform the inference, NetRate uses maximum likelihood (ML) estimation to determine $\alpha_{j,i}$ values based on observed spike data.

The concepts of survival and hazard functions have to be formalised before a likelihood function can be formed. The cumulative density function $F(t_i|t_j, \alpha_{j,i})$ is defined as the conditional probability that node i is infected by node j before or at time t_i given that node j was infected at time t_j. The survival function $S(t_i|t_j, \alpha_{j,i})$ is defined as:

$$S(t_i|t_j, \alpha_{j,i}) = 1 - F(t_i|t_j, \alpha_{j,i}) \tag{2.5}$$

The hazard function $H(t_i|t_j, \alpha_{j,i})$ is defined as the conditional instantaneous rate of infection of node i by node j. It is defined in (2.6) as the ratio of the conditional likelihood of transmission and the survival function.

$$H(t_i|t_j, \alpha_{j,i}) = \frac{f(t_i|t_j, \alpha_{j,i})}{S(t_i|t_j, \alpha_{j,i})} \tag{2.6}$$

2.4.3 Cascade Generation

It is crucial to understand the structure of the observed data-set over which the likelihood function will be maximised. To generate our data-set, we have to run multiple trials of an experiment. Each experiment will produce a **cascade**, $\mathbf{t}^c$. A cascade is a mathematical construct that will provide insight into the temporal dynamics of the underlying network. To run an experiment, we follow the following procedure:

1. At time $t = 0$, a person (node) within the network is randomly selected and is infected with a disease.

2. The disease is then allowed to propagate through the network based on the individual pair-wise transmission likelihood functions, $f(t_i|t_j, \alpha_{j,i})$, of each connection in the network. The simulation is run for a fixed amount of time, T, the **horizon**.

3. At the end of the simulation, information about the exact times at which nodes within the network were infected are collated and a cascade is formed.

A cascade is an N-dimensional vector, where N is the number of nodes in the network, such that $\mathbf{t}^c \in [0, T] \cup \{\infty\}$. Consider a network such that $N = 5$ and $T = 20$. At $t = 0$, node 3 is randomly chosen to be the start point of the experiment. The disease propagates through the network such that node 5 is infected at $t = 4$ and node 1 is infected at $t = 19$. This experiment would produce the following cascade:

$$\mathbf{t}^c = \{19,\ \infty,\ 0,\ \infty,\ 4\}$$

The ∞ placeholder in the cascade indicates the fact that a node remained uninfected by the end of the simulation; nodes 2 and 4 remained uninfected in the above example. Using only this cascade, we can only draw one concrete conclusion: there is a directed connection from node 3 to node 5, $\alpha_{3,5} \neq 0$. Since only node 3 has the disease before $t = 4$ (the point at which node 5 is infected), the only way that node 5 could have caught the disease is by contracting it from node 3. However, there are still multiple questions about the underlying network that this cascade leaves unanswered:

1. What the numerical value of $\alpha_{3,5}$? The cascade does not contain sufficient information for us to accurately determine the value of $\alpha_{3,5}$ which is in the range $(0, \infty)$.

2. At $t = 19$, only nodes 3 and 5 have the disease. We can thus conclude that node 1 must have contracted the disease from either node 3 or node 5: the exact node that transmitted the disease to node 1 is unknown. This implies that either $\alpha_{3,1} \neq 0$, or $\alpha_{5,1} \neq 0$. In actuality, both $\alpha_{3,1}$ and $\alpha_{5,1}$ could be non-zero.

3. Assuming that node 1 caught the disease from node 3, we would be led to believe that $\alpha_{3,5} \geq \alpha_{3,1}$. $\alpha_{3,5}$ should be larger than $\alpha_{3,1}$ because node 5 was infected by node 3 in four seconds while node 1 took 19 seconds to be infected by node 3. However, since the pairwise transmission likelihood functions are probabilistic rather than deterministic, there is no way to conclusively show this.

4. Are there directed connections between nodes 2 and 4? Since both these nodes remained uninfected during this experiment, this cascade does not shed any light on their interconnectivity.

5. Furthermore, it is inconclusive whether there are any directed connections between set of infected nodes (1, 3 and 5) and the set of uninfected nodes (2 and 4).

2.4.4 Likelihood Function

The unanswered questions above led Rodriguez to formulate a ML estimator to explain the observed data. The likelihood function of observing a single cascade, $\mathbf{t}^c$, is made up of three components. They are:

1. The first component of the likelihood function is the instantaneous probability of node i being infected at t_i. All the nodes that were infected before t_i will contribute to the total instantaneous probability of node i contracting the disease at t_i. It is assumed that any infected node can spread the infection to node i. Thus, the first component of the likelihood function is in the form $\sum_{j:t_j<t_i} H(t_i|t_j, \alpha_{j,i})$.

2. The second component of the likelihood function describes the fact that, given that node j was infected at t_j, node i was not infected before t_i. The survival function captures the concept of remaining uninfected. Thus the second term is in the form $\prod_{j:t_j<t_i} S(t_i|t_j, \alpha_{j,i})$.

3. Finally, a node that remains uninfected at the end of the experiment also provides useful information about the temporal dynamics of the network. This information is best described by the survival function and is in the form $\prod_{m:t_m>T} S(T|t_j, \alpha_{j,m})$.

The likelihood function for a single cascade is given by (2.7). The likelihood is parameterised by $\mathbf{A} := \{\alpha_{j,i} | i, j = 1, ..., n, i \neq j, \alpha_{j,i} \in [0, \infty)\}$, the adjacency matrix[3]. The entries in the matrix correspond to the pairwise transmission rates $\alpha_{j,i}$; these are the parameters over which the likelihood function will be maximised.

$$f(\mathbf{t}^c; \mathbf{A}) = \prod_{i:t_i<T} \prod_{m:t_m>T} S(T|t_i, \alpha_{i,m}) \prod_{k:t_k<t_i} S(t_i|t_k, \alpha_{k,i}) \sum_{j:t_j<t_i} H(t_i|t_j, \alpha_{j,i}) \tag{2.7}$$

The unanswered questions lead us to believe that one cascade simply does not contain enough information about the temporal dynamics of the network. The experiment is repeated multiple times[4] and each cascades is added to the set of all cascades, $\mathbf{C}$. Using the likelihood function of a single cascade, we can form a likelihood function to maximise the probability of observing all the cascades, $\mathbf{C}$, our entire data-set. The updated likelihood function is described in (2.8). **To form (2.8), cascades are assumed to be independent.**

$$\prod_{\mathbf{t}^c \in \mathbf{C}} f(\mathbf{t}^c; \mathbf{A}) \tag{2.8}$$

Thus, NetRate will determine the individual node-to-node transmission rates $\alpha_{j,i}$ such that the likelihood of observing $\mathbf{C}$, the set of all cascades, is maximised. This will allow us to approximate the answers to all the questions that were left unanswered. The optimisation problem is presented in (2.9).

$$\begin{aligned} &\underset{\mathbf{A}}{\text{minimise}} && -\sum_{\mathbf{t}^c \in \mathbf{C}} \log f(\mathbf{t}^c; \mathbf{A}) \\ &\text{subject to} && \alpha_{j,i} \geq 0, i, j = 1, ..., N, i \neq j \end{aligned} \tag{2.9}$$

[3]In section 3.1, we discuss adjacency matrices, their role in this project and how they are visualised. It is worth mentioning at this point at the variable i refers to the row indices whereas the variable j refers to the column indices. Thus $\alpha_{j,i}$ represents the entry in the ith row and jth column.

[4]To assess performance of the NetRate algorithm, Rodriguez generated 5000 cascades to infer a 1024 node network. Each node was randomly chosen as the start point of the experiment about five times on average.

2.4.5 Convexity of Optimisation Problem

If the survival function of the conditional probability distribution function, $f(t_i|t_j, \alpha_{j,i})$, is log-concave and if the hazard function is concave then the optimisation problem in (2.9) becomes convex in $\boldsymbol{A}$ and a global solution can be found.

2.4.6 Performance Metrics

The performance of the inference algorithm will be analysed using 4 numerical metrics.

1. Accuracy is defined as $1 - \frac{\sum_{j,i} |I(\alpha_{j,i}) - I(\alpha^*_{j,i})|}{\sum_{j,i} I(\alpha_{j,i}) + \sum_{j,i} I(\alpha_{j,i})}$, where $\alpha_{j,i}$ is the true transmission rate and $\alpha^*_{j,i}$ is the inferred transmission rate. Also, $I(\alpha) = 1$ if $\alpha > 0$ else $I(\alpha) = 0$. This is an important metric as networks with no edges or with only false edges will have zero accuracy. Accuracy is the primary metric used to quantify performance of the NetRate algorithm.

2. The difference in the inferred value and the true value is captured in the normalised mean absolute error (MAE). This is defined as $\mathbb{E}[|\alpha_{j,i} - \alpha^*_{j,i}|/\alpha_{j,i}]$. Note that the MAE places a larger penalty on over-predicting rather than under-predicting. This effect is magnified if $\alpha_{j,i}$ is small.

3. Recall will be used to quantify how efficient the algorithm is at identifying the underlying network. It is measured as the ratio of true edges in the inferred network to the **absolute number of true edges**.

4. It is possible to obtain 100% recall by predicting that all edges in the network exist: $\alpha_{j,i} \neq 0$ for all $\{i, j \in 1, 2, \ldots, N, i \neq j\}$. As such, it is important to measure how precise the network is. Precision will be used to quantify how frivolous the algorithm is at identifying connections. It is measured as the ratio of true edges in the inferred network to the **absolute number of inferred edges.**

Chapter 3

Biological Neural Network Simulation

In the previous chapter, we found that the NetRate algorithm takes cascades, $\mathbf{C}$, as an input and returns the adjacency matrix, $\mathbf{A}$, which contains the pairwise transmission rates for the entire network. In this chapter, we focus on the three essential elements of biological neural network simulation, namely:

1. Network structure
2. Individual neuron behaviour
3. Input stimulus model

3.1 Structure of Network

Before running any simulations, it is important to understand how neural networks[1] are connected. Network connectivity is defined in terms of an adjacency matrix. Adjacency matrices can be binary ($\alpha_{j,i} \in \{0, 1\}$), ternary ($\alpha_{j,i} \in \{-1, 0, 1\}$) or real ($\alpha_{j,i} \in \mathbb{R}$). All graphs, uniquely defined by their adjacency matrix, are generated using the Krongen[2] software, which is a Kronecker graph

[1] The terms graph and network will be used interchangably to mean a collection of nodes organised according to some structure.

[2] The Krongen software package is developed and maintained by the Stanford Network Analysis Project (SNAP) and can be accessed at: https://github.com/snap-stanford/snap/tree/master/examples/krongen

generator [23]. Krongen is capable of producing graphs with very different structures, namely random graphs [24], hierarchical graphs [25] and core-periphery graphs [26]. For the purpose of this project, we will narrow the scope and consider only Erdős & Renyi random graphs. To generate random graphs using Krongen, the initialization parameters $[0.5, 0.5; 0.5, 0.5]$ are used.

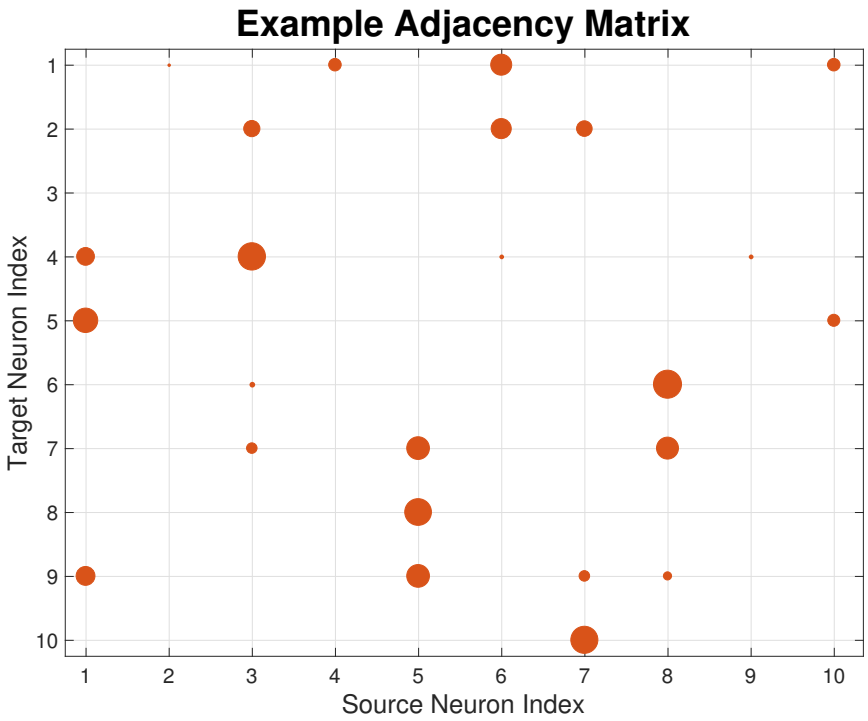


Figure 3.1: Example of how networks will be visualised: adjacency matrix will be plotted

Since a network's connectivity is completely defined by its adjacency matrix, connectivity of networks will be visualised by plotting the adjacency matrix. The y-axis shows the target node, i, while the x-axis shows the source node, j, and the size of the marker (dot) is relative to the weight of the directed connection. This is in keeping with the general practice of using i to represent row indices and j to represent column indices. If a directed connection exists from node j to node i, then $\alpha_{j,i}^{BNN} \neq 0$. The superscript BNN is used to differentiate between an entry in the adjacency matrix of a biological neural network and an entry in the adjacency matrix of a diffusion network. **Since this project aims to infer the size of directed connections using the NetRate algorithm, the weight of the connection, $\alpha_{j,i}^{BNN}$, is analogous to the pairwise transmission rate, $\alpha_{j,i}$.**

On a technical note, the Krongen software is only able to produce binary directed graphs. The software does not assign a weight to each of the non-zero directed connections. A value of $\alpha_{j,i}^{BNN}$ is programmatically assigned to each non-zero directed edge by sampling uniformly from $(0, 30]$.

When a network is simulated, the weight of directed connection, $\alpha_{j,i}^{BNN}$, is a very important variable. $\alpha_{j,i}^{BNN}$ will determine how much the membrane potential, v, of node i (the post-synaptic neuron), will increase when node j (the pre-synaptic neuron) spikes. With respect to the network graphed in figure 3.1, when node 10 spikes, the membrane potential, v, of node 1 and node 5 will be increased by $\alpha_{10,1}$ and $\alpha_{10,5}$ respectively. In chapter 4, we analytically prove that increasing a neuron's membrane potential may cause it to spike. The similarity between biological neural networks and diffusion networks is starting to become evident. When a biological neuron spikes, the neurons connected to it are likely to spike. When a person catches a disease, his or her neighbours are likely to get a disease.

Notice that the adjacency matrix is not symmetric. Symmetry would imply that the network is undirected; however, for this project, we focus our study on directed networks. Notice however that the diagonal entries of the adjacency matrix are zero. Nodes in the network are not connected to themselves because when a node spikes it does not increase its own membrane potential. Rather, the dynamics of what happens when a neuron spikes are described in equation (2.4).

3.2 Type of Neurons within the Network

Izhikevich has developed a model for biological neurons that consists of two nonlinear differential equations. Although the model is only two-dimensional, it can be adapted to replicate multiple types of spiking behaviour using four tuning parameters, a, b, c and d.

Neurons can be broadly classified into two classes based on their spiking behaviour, namely, excitatory and inhibitory neurons. Within the broad class of excitatory neurons, neurons still exhibit remarkably different behaviours. Regular spiking neurons and chattering neurons are two examples of excitatory neurons. In contrast, within the broad class of inhibitory neurons, neurons typically display fast spiking behaviour. The response of regular spiking, chattering and fast spiking neurons to a constant input is graphed in figure 3.2. Note that the y-axis represents the membrane potential, v, of the neurons. The membrane recovery variable, u, which is not an observable state of biological neurons and is instead a tool to model refractory behaviour, has not been graphed.

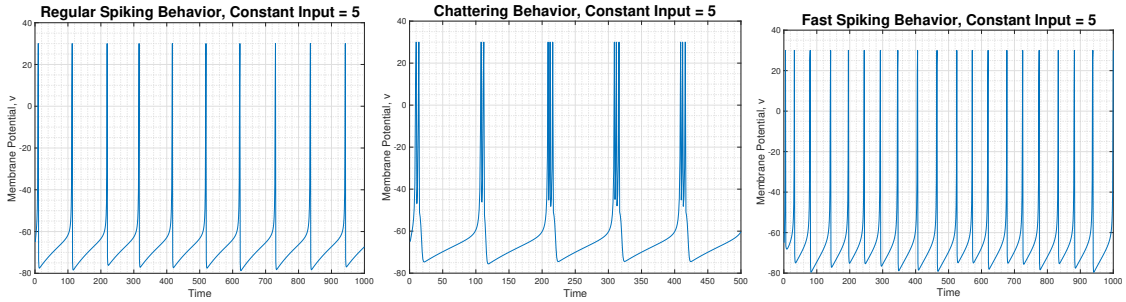


Figure 3.2: Responses of different types of neurons to constant input stimulus

The graphs above exhibit the full capability of Izhikevich's model to replicate biologically observable spiking behaviour. However, to reduce the complexity of our inference problem, we will only consider networks consisting of excitatory regular spiking neurons. Regular spiking behaviour is obtained by setting the tuning parameters in Izhikevich model to the following: $a = 0.02$, $b = 0.2$, $c = -65$, $d = 8$.

3.3 Input Stimulus Model

3.3.1 Importance of Appropriate Input Stimulus Model

Next, we focus our attention on the input stimulus that will be provided to each neuron within the network. Izhikevich defines the input stimulus, I, in equation (2.2) as being normally distributed. The normally distributed input models noise due to the spiking behaviour of neurons **not** in the network. Modelling this noise as Gaussian is not a definitive rule. Often input stimuli are modelled as Poisson processes[3] rather than Gaussian random variables.

Arbitrarily choosing a input stimulus model can lead to exceptionally contrasting network behaviour. To illustrate the importance of an appropriate input, consider the two plots in figure 3.3. The plot on the left is a reproduction of Izhikevich's 1000-neuron simulation that he presented in [1]. Each neuron in the network is provided with an independent normally distributed random

[3]Poisson processes are used to model the spike train of neurons. Although real neurons do not generate Poisson spike trains because of their inherent refractoriness, Poisson processes capture general characteristics of the spike train well. Gaussian noise assumes that each neuron within the network is influenced by a collection of neurons **outside** the network, whereas Poisson noise assumes that each neuron within the network is only influenced by one neuron **outside** the network.

input at each time step. The plot on the right is a modified simulation where the input stimulus to each neuron in the network is an independent Poisson process.

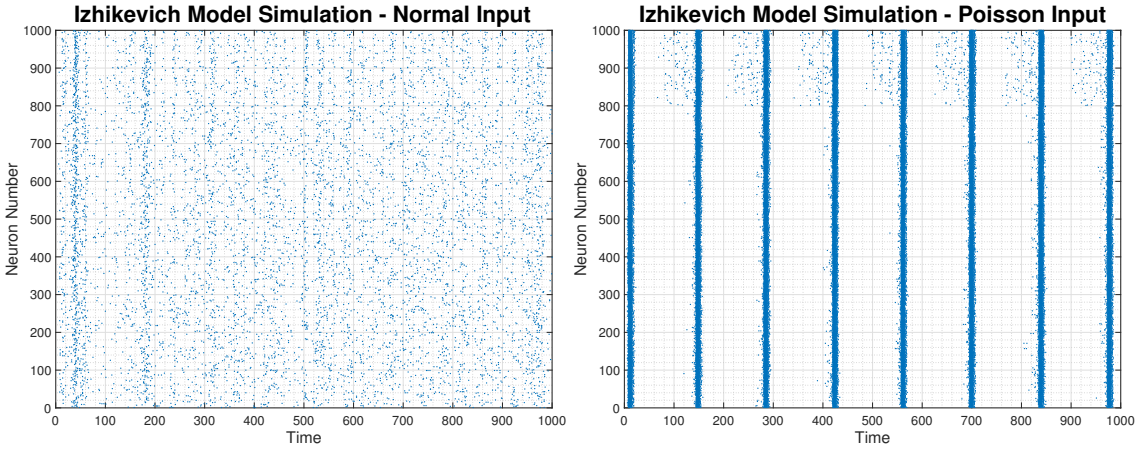


Figure 3.3: Effect of input stimulus on simulation

The above graphs show the drastically different ways in which the network responds to different input signals. Network simulations will provide data from which cascades will be formed. The cascades will then be used by NetRate to infer network connectivity. Hence, the quality of the simulation will directly determine the accuracy of the inferred network. **It is also worth mentioning that the simulation has to be physically reproducible.** One of the main motivations behind using the Izhikevich model of a neuron is its ability to replicate a whole array of naturally occurring spiking behaviour. Using the Izhikevich neuron in our networks allows us to preserve the biological significance of this project. The same governing principle should be applied when settling on a input stimulus model. We should only run simulations and work with data that can feasibly be collected in the laboratory. The technologies that make certain input stimulus models feasible while precluding other input stimulus models is briefly discussed in section 3.3.3.

3.3.2 Izhikevich Input Stimulus Model

Input Stimulus Model

We first dissect the Izhikevich input stimulus model and attempt to extract spike data in the form of cascades from his simulation. The input stimulus that Izhikevich applies to each and every neuron in the network is an independent normally distributed random variable with a mean of 0

and a standard deviation[4] of 5.

Cascade Generation

We revert to the illustrative example of disease propagation. A randomly selected starting node is infected with a disease and its spread through the network is captured in a cascade. To extract cascades from Izhikevich's simulation, we can consider each spike as an independent disease that has been released into the population. Using that spike as a starting point, we then consider a fixed observation window. **Horizon** is the term that Rodriguez uses to describe the length of the observation window. With respect to the starting point, we find the relative delay for each neuron that fires in the observation window summarising this information into a cascade. The simulation is essentially broken down into bins and firings within each bin are collated into a cascade.

Note that the maximum length of the horizon is limited by the fact that no neuron should fire more than once within one observation window. This is a necessary requirement that will ensure that we obey Rodriguez's third assumption about observable artefacts[5] in diffusion networks: observable artefacts are binary.

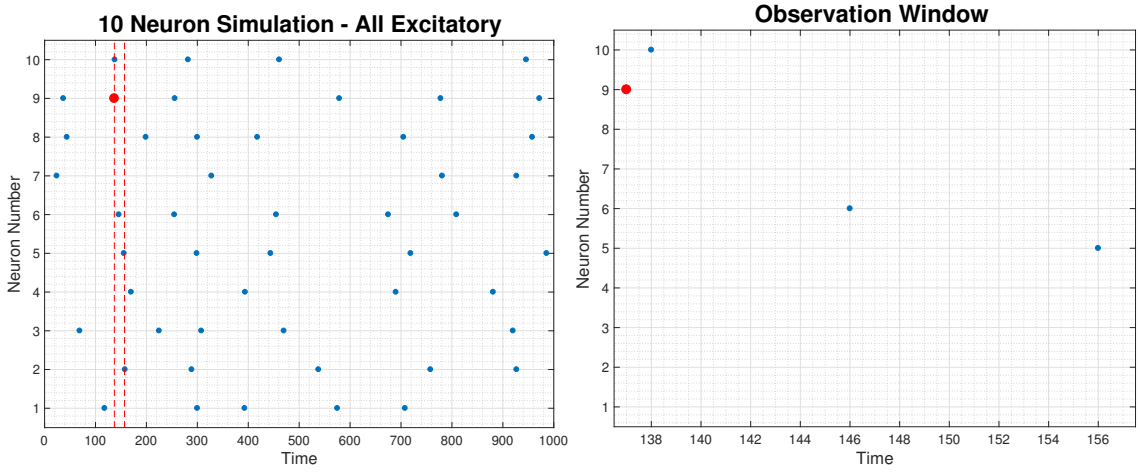


Figure 3.4: Generation of cascades using Izhikevich's input stimulus model

[4]Izhikevich's original simulation contained both excitatory and inhibitory neurons. The standard deviation of the input stimulus applied to excitatory neurons was 5 while the standard deviation of the input stimulus applied to inhibitory neurons was 2. As mentioned earlier, we will be narrowing the scope of the project and analysing networks consisting of only regular spiking excitatory neurons.

[5]Spikes in neural networks are analogous to observable artefacts in diffusion networks. A detailed discussion of similarities and differences between diffusion networks and neural networks is presented in chapter 4.

To illustrate the cascade generation process, consider the simulation presented in figure 3.4. The point highlighted in red, neuron 9 firing at $t = 137$, will be used as an example to demonstrate the cascade generation process. Selecting a firing event to begin a cascade is analogous to starting a new experiment by introducing a new disease into the diffusion network. The next neuron to fire (contract the disease) is neuron 10 at $t = 138$, followed by neuron 6 at $t = 146$ and finally neuron 5 at $t = 156$. The sequence of firings produces the following cascade:

$$\mathbf{t}^c = \{\infty,\ \infty,\ \infty,\ \infty,\ 19,\ 9,\ \infty,\ \infty,\ 0,\ 1\} \tag{3.1}$$

Again the ∞ placeholder in the cascade represents the fact that the neuron did not fire within the observation window. In the above example, neurons $1, 2, 3, 4, 7$, and 8 did not fire between $t = 137$ and $t = 157$. The above example details the cascade generation process for a randomly chosen firing event. The cascade generation process has to be repeated for every single firing instance; the simulation represented in figure 3.4 had 49 firing events and thus 49 cascades will be generated.

Fundamental Problem with Cascade Generation Methodology

The aforementioned method to generate cascades is the simplest way to extract information from the Izhikevich simulation. However, extracting data in this manner is fundamentally flawed. **The cascades that are generated will not be independent.** Cascade independence was assumed when equation (2.8) was formulated: independence ensures that the joint probability of two events is equivalent to the product of their individual probabilities. However, in the aforementioned method, we generate a cascade for neuron 9 firing at $t = 137$. The firing of neuron 10 at $t = 138$ is captured in this cascade. However, as mentioned, we would also have to generate a separate cascade for neuron 10 firing at $t = 138$. Information is duplicated and thus cascades generated in this manner are not independent.

Extracting independent cascades is not straightforward. **The main obstacle preventing us from generating independent cascades is that we do not have a systematic way to select which firing events should initialise cascades and which firing events should not.** The firing events that do not initialise their own cascades will become part of other cascades.

The process of extracting independent cascades is much simpler in diffusion networks. In diffusion networks, independent experiments are run by infecting a randomly selected a node with a disease

and collating the disease's propagation into a cascade. For each experiment, one cascade was formed. This is possible because there is a clear entry point into the experiment: the very start of the experiment. However, in the simulation presented in figure 3.4, there is no clear entry point[6]. The aforementioned cascade generation method considered each firing event as the start of a unique experiment, however this led to cascade dependence.

In the next section, we modify the input stimulus model and systematically identify clear entry points into the biological neural network simulation such that we can generate independent cascades.

3.3.3 Enhanced Input Stimulus Model

Instead of devising a systematic method of identifying firing events such that the generated cascades do not duplicate information and are completely independent, we solve the problem of cascade dependence by altering the input stimulus model. Before describing the enhanced input stimulus model, it is imperative to pay particular attention to its practicality. The proposed solution to the network inference problem that this project puts forward would be purely academic and would lose biological significance if the data required to run the NetRate algorithm is physically unattainable.

Izhikevich's simulation, presented in figure 2.1, was meant to show the capability of his two-dimensional neuron model. He wanted to show that his model of a neuron works not only at a single neuron level but also as part of larger network. The real-life applicability of Izhikevich neurons was proven by their ability to reproduce well-described neural activity such as neural oscillations, primarily alpha and gamma rhythms[7].

He utilised an input stimulus model that allowed him to reproduce neural oscillations. However, the purpose of this project is to solve a network inference problem and thus obstinately abiding to his input stimulus model is incorrect.

Advancements in optogenetic actuators have made it possible for us to enable activation and inactivation, with millisecond precision, at the spatial scale of populations of neurons [27]. With this in mind, it is realistic to suggest an input stimulus model that supplies individual neurons in the

[6]Due to fundamental differences between the two types of networks, it is not possible to run experiments on neural networks in the way that Rodriguez proposed running experiments on diffusion networks

[7]Alpha and gamma rhythms are clearly observed in figure 2.1.

network with a **constant input**. Recall that we will only analyse networks consisting of excitatory regular spiking neurons. Figure 3.2 depicts how excitatory regular spiking neurons respond to a constant input stimulus; they spike periodically.

Independent Cascade Generation using Enhanced Input Stimulus Model

Using the enhanced input stimulus model, we shall simulate an entire network and attempt to generate independent cascades. We randomly select node 3 and supply it with a constant input of 12 rather than with a normally distributed random input: this will cause the neuron to spike periodically. The rest of the neurons in the network are supplied with Gaussian noise[8]. The resulting simulation is presented below. **It is evident that if we generate cascades only for the highlighted firing events, i.e., each time the stimulated node spikes, the cascades will be completely independent; there will be no duplication of information.** Since node 3 fires 10 times, we are able to generate 10 independent cascades from this simulation.

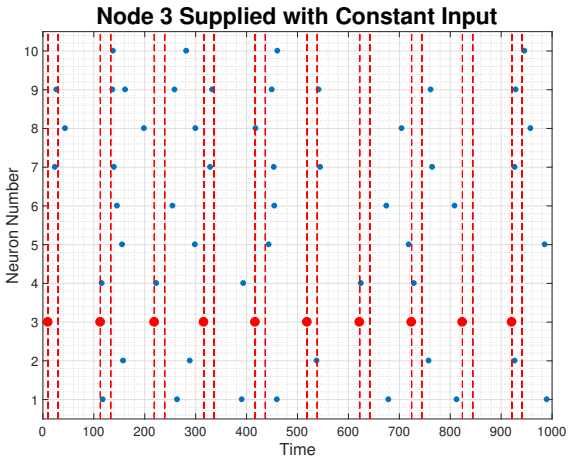


Figure 3.5: Network simulation with enhanced input stimulus model

The enhanced input stimulus model not only grants us a systematic method of generating independent cascades but it also mimics Rodriguez's experiments. He randomly selects a node to infect with a disease and subsequently tracks the disease's propagation though the network for a fixed amount of time. By stimulating node 3 with a constant input, we are introducing a new independent disease into the network each time the node spikes. This represents a clear entry point

[8]The Gaussian noise is applied to model the noise present in the laboratory.

into the simulation and thus corresponds to a firing event that should initialise a cascade. We can control the frequency at which node 3 spikes by changing the magnitude of the constant input. If the selected node is spiking very rapidly, we would have to shorten the horizon; NetRate performs poorly if the horizon is too short. In addition, a relatively long break between consecutive spikes will allow the network to settle to a steady state. The previously introduced disease would have sufficient time 'die out' before a new independent disease is introduced into the network[9].

3.4 Concluding Remarks

In this chapter, we began by determining the types of networks we will consider in this project. The Krongen software was adapted to produce directed Erdős & Renyi random graphs. The size of each non-zero connection is uniformly distributed from $(0, 30]$. Next, the full capability of the two-dimensional Izhikevich neuron model was demonstrated by graphing the remarkably different spiking behaviour that Izhikevich neurons can display by simply adjusting 4 tuning parameters. To reduce the complexity of the inference problem, it was decided that only graphs consisting of excitatory regular spiking neurons will be studied. Finally, the problems inherent in generating independent cascades from Izhikevich's simulation were highlighted. Altering the input stimulus model to obtain an independent cascade generation scheme was considered and shown to be feasible.

We now have a systematic cascade generation scheme that can provide us with the data over which NetRate, the ML estimator, needs to work. We will now focus our attention on understanding the temporal dynamics of our network. In the next chapter, we will attempt to approximate the shape of $f(t_i|t_j;\alpha_{j,i})$. Without the general shape of the pairwise transmission likelihood function, the NetRate algorithm cannot work.

[9]It is also not ideal to have too long a gap between successive spikes. This would yield fewer cascades. Consider the above example. Node 3 was stimulated for 1000ms and 10 cascades were generated. If the delay between successive spikes is too long, we would have to stimulate node 3 for much longer to obtain 10 cascades. There is a computational cost associated with running simulations that should be considered when determining the magnitude of the constant input

Chapter 4

Temporal Dynamics of Neural Networks

NetRate is able to infer the weights in an adjacency matrix, $\mathbf{A}$, given an underlying probability distribution that describes how infections are spread through a network. Mathematically, given the general shape of $f(t_i|t_j;\alpha_{j,i})$, the algorithm is able to determine the values of $\alpha_{j,i}$. Furthermore, Rodriguez showed that the NetRate algorithm is convex in $\mathbf{A}$ when $f(t_i|t_j;\alpha_{j,i})$ follows one of the following parametric probability distributions: Exponential, Power Law or Rayleigh. These distributions satisfy the criteria that make (2.9) convex: their survival functions are log-concave and their hazard functions are convex.

To utilise the NetRate algorithm to solve the neural network inference problem, we first draw some parallels between diffusion networks and neural networks. Some of these similarities were alluded to in the previous chapter.

1. **The propagation of action potentials is analogous to the spreading of infections. They are both observable artefacts from source nodes to target nodes that are governed by a node-to-node interaction mechanism.**

2. In diffusion networks, infections are binary and once a node has an infection it cannot be infected again. In neural networks, spiking is a binary observable artefact. A neuron is either spiking or it is not. However, unlike diffusion networks, once a neuron spikes, it can spike again. To resolve this apparent difference, we artificially limit the horizon so that no neuron

can be infected twice within the same cascade. This was discussed in section 3.3.2.

3. **For both types of networks, the entries of the adjacency matrix represent how the network is physically connected.** If there is no directed connection from node j to node i, then the (i, j) entry in the adjacency matrix will be 0. If there is a directed connection from node j to node i, the (i, j) entry in the adjacency matrix will be non-zero. In addition, the magnitude of the entry will represent the strength of the directed connection.

While there are some similarities between the two network models, there are also some considerable differences. Here we list these differences. The rest of the chapter is devoted to resolving these differences and showing that under some loose approximations, we can theoretically utilise NetRate to infer the neural network connectivity.

1. Diffusion networks are described probabilistically. Infections spread through a network based on $f(t_i|t_j; \alpha_{j,i})$. In contrast, the Izhikevich model is described deterministically in the form of a two-dimensional nonlinear dynamical system. This apparent mismatch will be resolved in section 4.1 where we show that although each individual neuron is described deterministically, the entire network can be described probabilistically.

2. The fact that neural networks are described deterministically also raises another issue. In diffusion networks, $f(t_i|t_j; \alpha_{j,i})$, is parameterised by $\alpha_{j,i}$. **The transmission likelihoods, $\alpha_{j,i}$, determine the rate at which a disease will propagate from node j to node i.** The entries of the biological neural network's adjacency matrix, $\alpha_{j,i}^{BNN}$, are equal to the amount by which the membrane potential of post-synaptic neuron i increases when pre-synaptic neuron j spikes. In section 4.2, we estimate the conditional pairwise transmission likelihood, $f^{BNN}(t_i|t_j; \alpha_{j,i}^{BNN})$ for biological neural networks. **This is possible because how long a post-synaptic neuron i takes to spike is directly related to $\alpha_{j,i}^{BNN}$, the amount by which its membrane potential is increased by the spiking of a pre-synaptic neuron j.**

4.1 Feasibility of Probabilistic Description of Spike Propagation

In this section, we will explain why it is appropriate to describe a network consisting of Izhikevich neurons probabilistically even though the state of an individual neurons evolves deterministically.

To begin, recall that the propagation of action potentials, analogous to the spreading of infections, through a network is described deterministically through a set of nonlinear differential equations described in (2.2) and (2.3). Taking that into consideration, it is not an unreasonable proposition to model a network consisting of Izhikevich neurons probabilistically. As discussed in section 3.3.3, the variable I represents a normally distributed random variable[1]. Hence, although the system is deterministic, the input to the system is random and a probabilistic model can be approximated.

Formulating a probabilistic description of the neural network is equivalent to estimating the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$. The probabilistic description of the observed spike data will be parameterised by the entries of the adjacency matrix, $\alpha_{j,i}^{BNN}$. **Ideally, different values of $\alpha_{j,i}^{BNN}$ should not affect the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i})$.** For example, consider the graphs below. They show that the Exponential and Rayleigh distributions have unique shapes. The parameters λ and σ do not alter the unique shape of the distribution but simply stretch or shrink the graph.

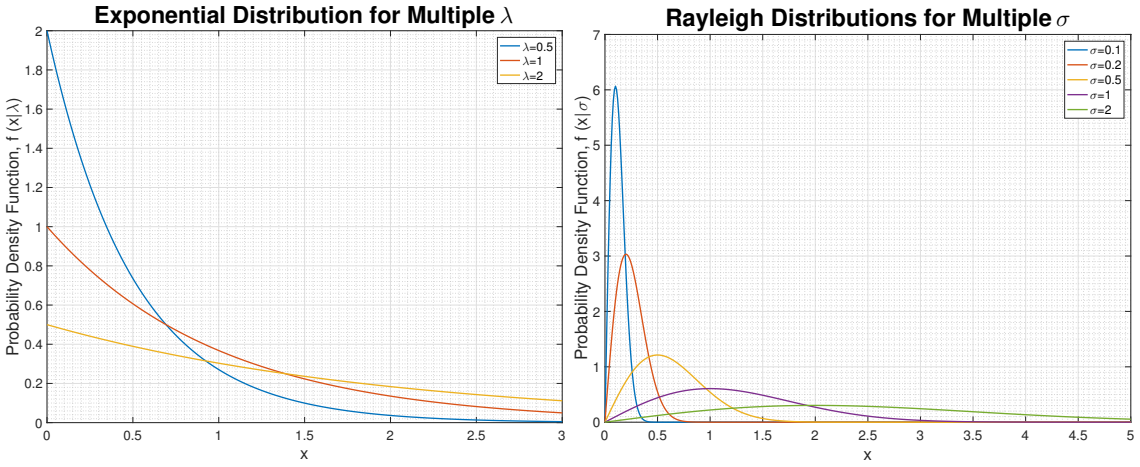


Figure 4.1: General shapes of Exponential and Rayleigh distributions and the effects of λ and σ on the shape of the distribution

[1]At any time, only one of the nodes in the network is stimulated with a constant input whereas all other neurons in the network are stimulated with independent Gaussian noise. Thus it is safe to assume that the input to the network is random.

4.2 Formulation of Probability Distribution

4.2.1 Ambiguity in Cause of Spike

The first step in estimating $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ is to formulate a systematic method for identifying the cause of a spike. Knowledge of the physical connectivity of the network is a necessary pre-requisite to identifying the cause of a spike. To that end, we assume knowledge of network connectivity for the sole purpose of deriving a probabilistic model of spike propagation. In the general case, we will be attempting to infer the underlying connectivity from the spike data, i.e. the converse problem.

Consider the following network.

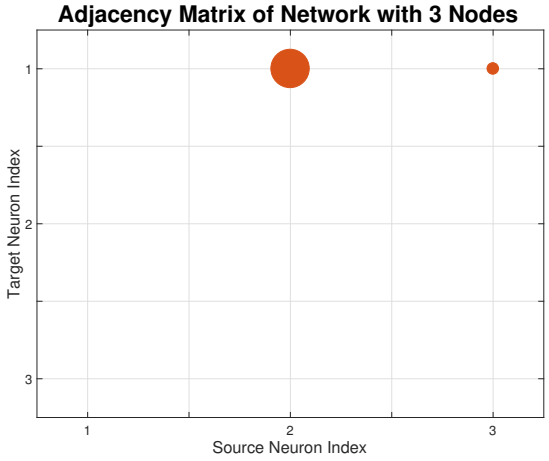


Figure 4.2: Three node network used to demonstrate ambiguity in identifying the cause of a spike

Consider the scenario in which node 2 spikes at $t = 0$. Since there is a connection between node 2 and node 1, the membrane potential of node 1 will be increased: the increase in membrane potential is equal to the size of their connection, $\alpha_{2,1}^{BNN}$. Consider now that node 3 spikes at $t = 1$. Again since there is a connection between nodes 3 and node 1, the membrane potential of node 1 will increase by $\alpha_{3,1}^{BNN}$. Imagine now that node 1 spikes at $t = 2$. We do not have a systematic way of identifying what caused node 1 to spike. Either node 2, node 3, or random noise could have caused node 1 to spike; even though we have full knowledge of how the network is connected, we are unable to accurately identify the cause of a spike. **The ambiguity highlighted in this illustrative example underscores the necessity to formulate a systematic way to identify**

the root cause of a spike.

In the next section, we develop a systematic method of identifying the cause of a spike. It is interesting to realise that it is only possible to reverse engineer $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ because the system is inherently deterministic. For a purely probabilistic system, the exact kind that Rodriquez refers to when discussing diffusion networks, it would be near impossible to identify the root cause of an observable artefact without any ambiguity.

4.2.2 Stability Region of Regular Spiking Excitatory Neuron

To formulate an unambiguous method for identifying the cause of a spike, we first have to understand the stability of excitatory regular spiking neurons: excitatory regular spiking neurons are obtained by setting $a = 0.02$, $b = 0.2$, $c = -65$, $d = 8$. Since the Izhikevich neuron is described in terms of a two-dimensional nonlinear dynamical system, we use control theory to determine the stability of the system. The input, I, has a mean of 0 and has been excluded from the differential equations.

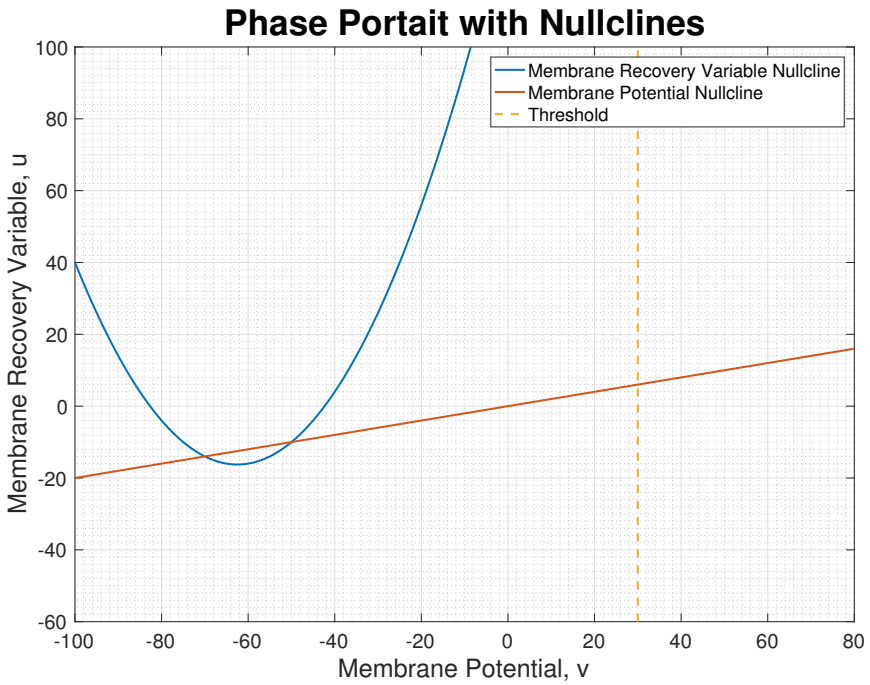


Figure 4.3: Phase portrait of excitatory regular spiking neuron with nullclines

Figure 4.3 shows the two nullclines obtained by setting $\frac{dv(t)}{dt}$ and $\frac{du(t)}{dt}$ to 0. The dotted line represents the threshold value beyond which the response of the system is governed by equation (2.4). The intersections of the two nullclines represent the equilibrium points.

$$\frac{dv}{dt} = f(v,u) = 0.04v^2 + 5v + 140 - u = 0 \implies u = 0.04v^2 + 5v + 140$$
$$\frac{du}{dt} = g(v,u) = a(bv - u) = 0 \implies u = bv$$

To find the intersection points, we perform a simple substitution and obtain the following quadratic equation:

$$0.04v^2 + 5v + 140 - 0.2v = 0$$
$$v = \frac{-4.8 \pm \sqrt{4.8^2 - 4(0.04)(140)}}{2(0.04)}$$

Solving the quadratic equation, we determine that the system has equilibrium points at $(v_1, u_1) = (-70, -14)$ and $(v_2, u_2) = (-50, -10)$. Since we want to determine the stability of each of the equilibrium points, we have to formulate the Jacobian matrix.

$$J = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial u} \\ \frac{\partial g}{\partial v} & \frac{\partial g}{\partial u} \end{bmatrix} = \begin{bmatrix} 0.08v + 5 & -1 \\ ab & -a \end{bmatrix}$$

We substitute the value of $v = v_1 = -70$ and compute the eigenvalues of the matrix. The eigenvalues of the Jacobian matrix are $\lambda_1 = -0.5930$ and $\lambda_2 = -0.02730$. Both the eigenvalues are negative real numbers and so the equilibrium point at $(v_1, u_1) = (-70, -14)$ is stable. Next, we substitute $v = v_2 = -50$ and compute the eigenvalues: we obtain $\lambda_1 = 0.9961$ and $\lambda_2 = -0.0161$. Since λ_1 is a positive real number, the equilibrium point is unstable. This is exactly as expected. When the state of the neuron starts to diverge and tend to infinity, the neuron is effectively spiking. We have to manually reset the membrane potential, v to the value of c and the membrane recovery variable, u to $u + d$ when the value of v goes over the threshold value. As a result, although the system has an unstable equilibrium point, we do not allow system to diverge.

The equilibrium point at $(v_2, u_2) = (-50, -10)$ is unstable. However, since one of the two eigenvalues is negative, we can numerically determine the stability boundary. Implementing the algorithm found in [28], we find the stability boundary. The Matlab code for generating the stability boundary is replicated in listing 1.

```
% vectors to hold results
positive_direction=[];
negative_direction=[];

unstable_v=-50;
simulation_resolution=0.2;
a=0.02;
b=0.2;

% we are transforming the system into one that is linear around the equilibrium point
J = [0.08*unstable_v+5, -1; 0.02*0.2, -0.02];

% get the eigenvalues of the system
[eigenvectors,~]=eig(J);

% initialise variables in appropriate directions
vp = unstable_v+eigenvectors(1,2);
up = b*unstable_v+eigenvectors(2,2);
vm = unstable_v-eigenvectors(1,2);
um = b*unstable_v-eigenvectors(2,2);

% run simulations
while (abs(vp) < 100)
    vp=vp-simulation_resolution*(0.04*vp^2+5*vp+140-up);
    up=up-simulation_resolution*a*(b*vp-up);
    positive_direction = [positive_direction; vp,up];
end

while(abs(vm)<1000)
        vm=vm-simulation_resolution*(0.04*vm^2+5*vm+140-um);
        um=um-simulation_resolution*a*(b*vm-um);
        negative_direction = [negative_direction; vm,um];
end

% plotting
hold on;
v=[negative_direction(end:-1:1,1);positive_direction(:,1)];
u=[negative_direction(end:-1:1,2);positive_direction(:,2)];
plot(v,u,'LineWidth',3);
hold off;
```

Listing 1: Matlab code to generate stability boundary for excitatory regular spiking neurons

The graph below shows the stability boundary.

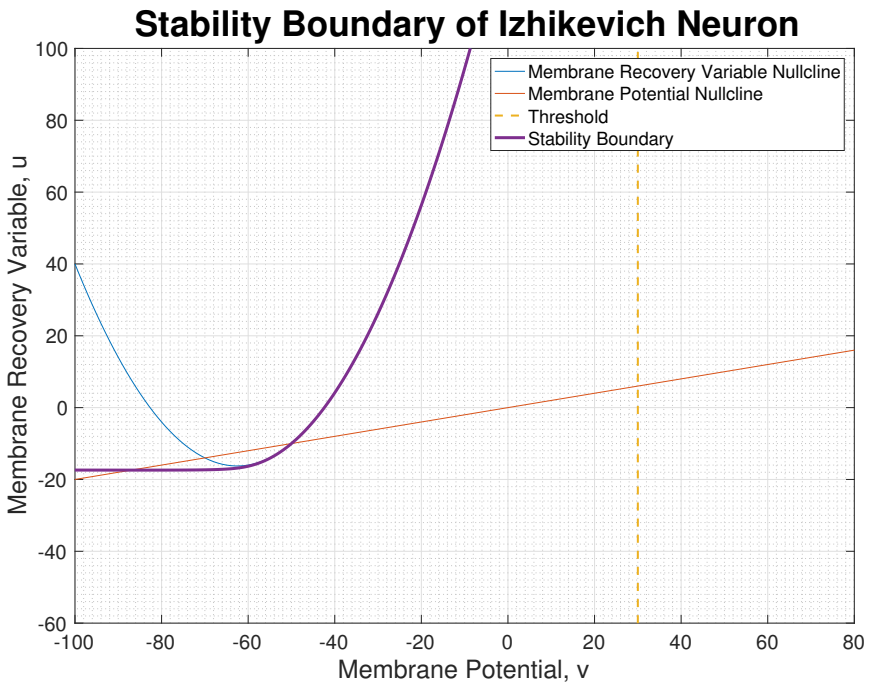


Figure 4.4: Stability boundary of excitatory regular spiking neuron

Any point above the stability boundary will converge to the equilibrium point at $(v_1, u_1) = (-70, -14)$, whereas any point below the stability boundary will result in a spike and the resetting of the v and u variables. The graphs presented in figure 4.5 illustrate that the behaviour of a neuron is dependent on which side of the stability boundary it is.

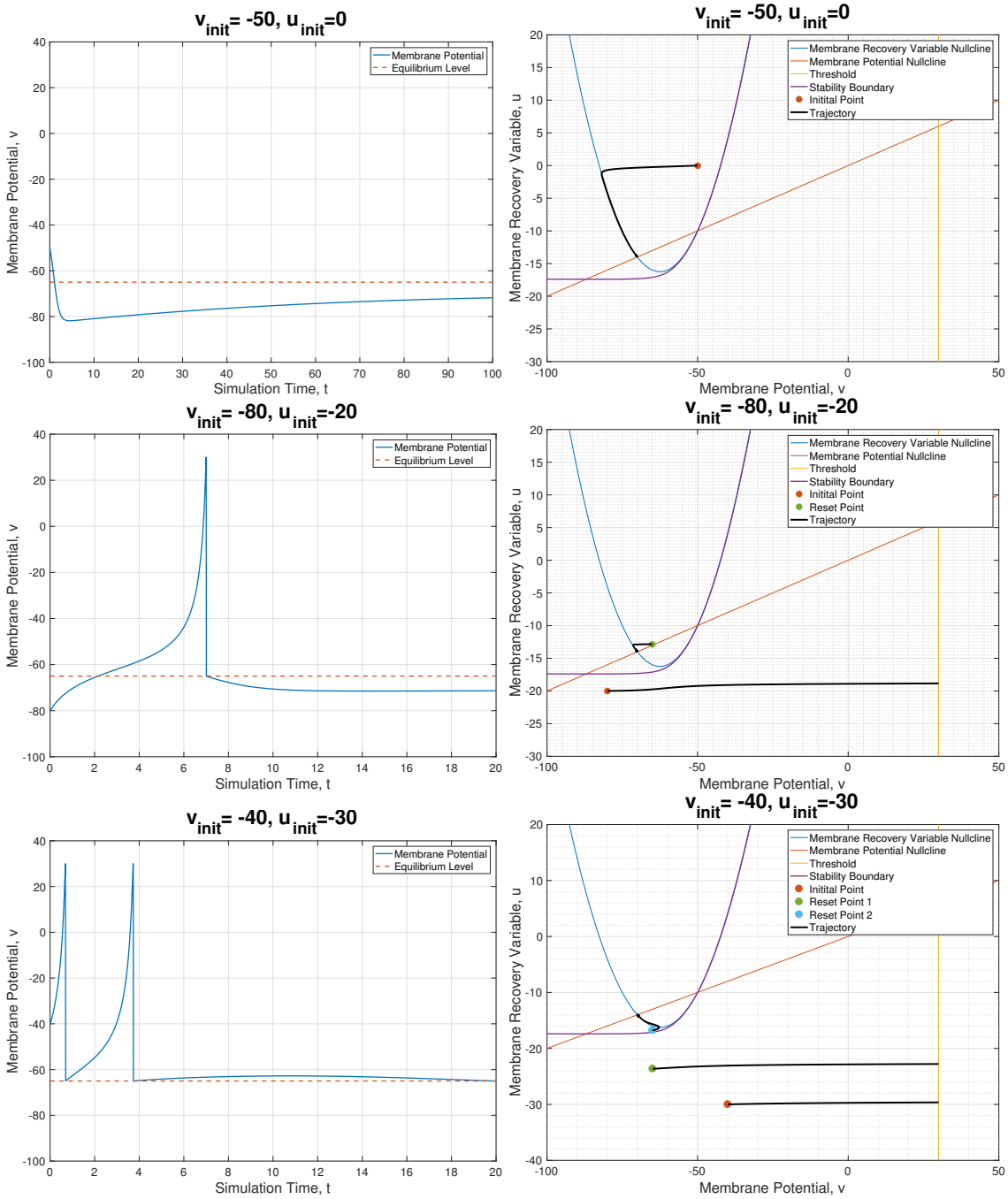


Figure 4.5: Trajectories for three unique initial points

The above analysis yields 2 extremely important results:

1. The analysis above shows that the time to spike is directly dependent on the initial state of the neuron. For example, when the initial state is $(v_{init}, u_{init}) = (-80, -20)$, the neuron takes approximately 7 seconds to spike. In contrast, when the initial state is $(v_{init}, u_{init}) = (-40, -30)$, the neuron takes only 1 second to spike. After spiking, the neuron is reset to a state of $(v_{reset}, u_{reset}) = (-65, 23)$. Since the reset state is still below the stability boundary, the neuron will spike again. Notice that the neuron takes approximately 3 seconds to spike from the time it is reset.

2. Identification of stable and unstable regions allows us to pin-point the exact moment when a neuron enters the unstable region. If we know when a neuron enters the unstable region, we can, without any ambiguity, identify the cause of a spike, be it another neuron in the network or the random noise.

The first result validates our earlier claim that the time post-synaptic neuron i takes to spike is directly related to $\alpha_{j,i}^{BNN}$, the amount by which its membrane potential is increased by pre-synaptic neuron j. The larger the value of $\alpha_{j,i}^{BNN}$, the closer post-synaptic neuron i is pushed to the threshold by pre-synaptic neuron j and the faster it is going to spike. **The first result affirms that estimating $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ is fundamentally possible and the second result equips us with the necessary tool to make the estimation.**

4.2.3 Simulation Analysis to Determine Transmission Likelihood

With the power to identify the cause of any spike, we analyse a biological neural network simulation. We note that we are trying to identify the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$. Since this function is parameterised by $\alpha_{j,i}^{BNN}$, for the purpose of analysis, we have to limit the possible range of values which $\alpha_{j,i}^{BNN}$ can take. Recall that in section 3.1, we assigned $\alpha_{j,i}^{BNN}$ by sampling uniformly from $(0, 30]$. To allow us to visually estimate the shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$, we shall analyse a network where the values of $\alpha_{j,i}^{BNN} \in 0, 3, 6, \ldots, 30$. Discretising the values of $\alpha_{j,i}^{BNN}$ enables us form a histogram for each value of $\alpha_{j,i}^{BNN}$. Using these histograms, we will not only be able to determine the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ but will also be able study the effect that $\alpha_{j,i}^{BNN}$ has on the function.

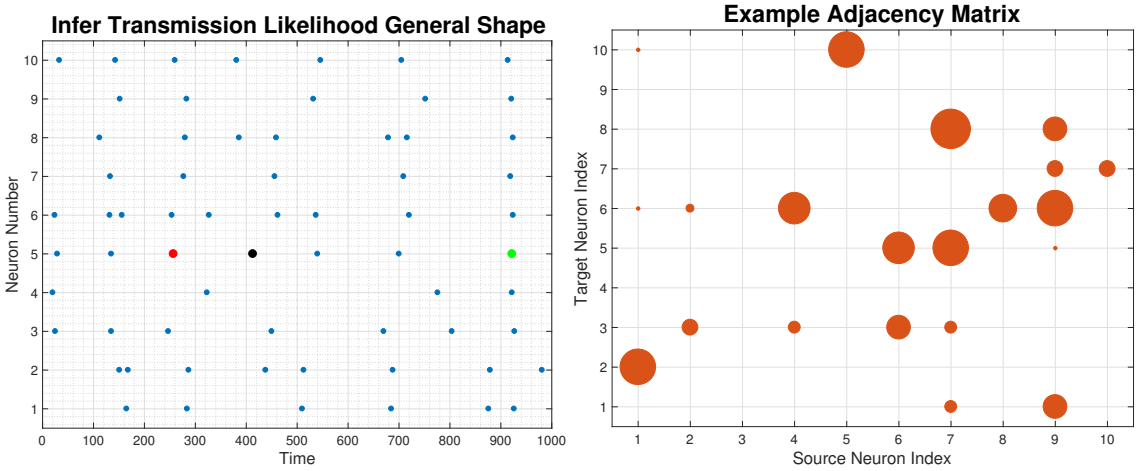


Figure 4.6: Example simulation to explain how shape of transmission likelihood is determined

Consider the simulation above. For each firing event, we wish to identify the root cause of the spike. We have highlighted 3 firing events that are of particular interest. All three points are firing events associated with neuron 5. The adjacency matrix shows that neurons 6, 7 and 9 have directed connections to neuron 5. As such, any firing event associated with neuron 5 could have been caused by either neuron 6, 7 or 9. It is also wholly possible that neuron 5 fires due to random noise and not due to its neighbours.

1. Red dot: Neuron 5 fires at $t = 257$. It becomes unstable at $t = 254$. This is the same time that neuron 6 fires. The firing event at $t = 257$ occurs due to neuron 6. Since the weight of the directed connection from neuron 6 to neuron 5 is 21 and the time delay between the spikes is three, we place this firing event into the third bin of the histogram for $\alpha_{j,i} = 21$. The histogram is graphed in figure 4.7.

2. Green dot: Neuron 5 fires at $t = 921$. It becomes unstable at $t = 920$. This is the same time that neuron 7 fires. The firing event at $t = 921$ occurs due to neuron 7. Since the weight of the directed connection from neuron 7 to neuron 5 is 27 and the time delay between the spikes is one, we place this firing event into the first bin of the histogram for $\alpha_{j,i} = 27$. The histogram is also graphed in figure 4.7.

3. Black dot: Neuron 5 fires at $t = 413$. It becomes unstable at $t = 408$. None of its connections have fired at that time. The firing event at $t = 413$ is due to noise. Since the neuron spikes

due to noise and not because of any of its neighbours, this firing event it is not included in any of the histograms.

Figure 4.7 shows how the firing events at $t = 257$, $t = 408$ and $t = 921$ affect the histograms.

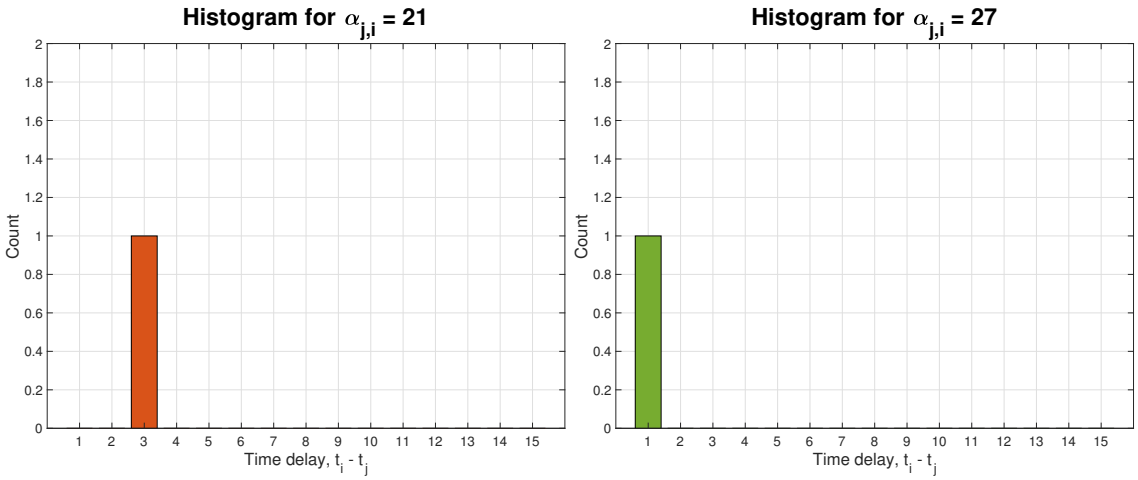


Figure 4.7: Accumulation of causes of firing events into appropriate histograms

We repeat this procedure for each firing event in the entire network and formulate a histogram for each discrete value of $\alpha_{j,i}$: remember that for the sake of understanding how the probability distribution is shaped, we have artificially created a network with only 10 possible values of $\alpha_{j,i}$. In reality, our study is focused on networks for which entries of the adjacency matrix can take any value between 0 and 30, inclusive.

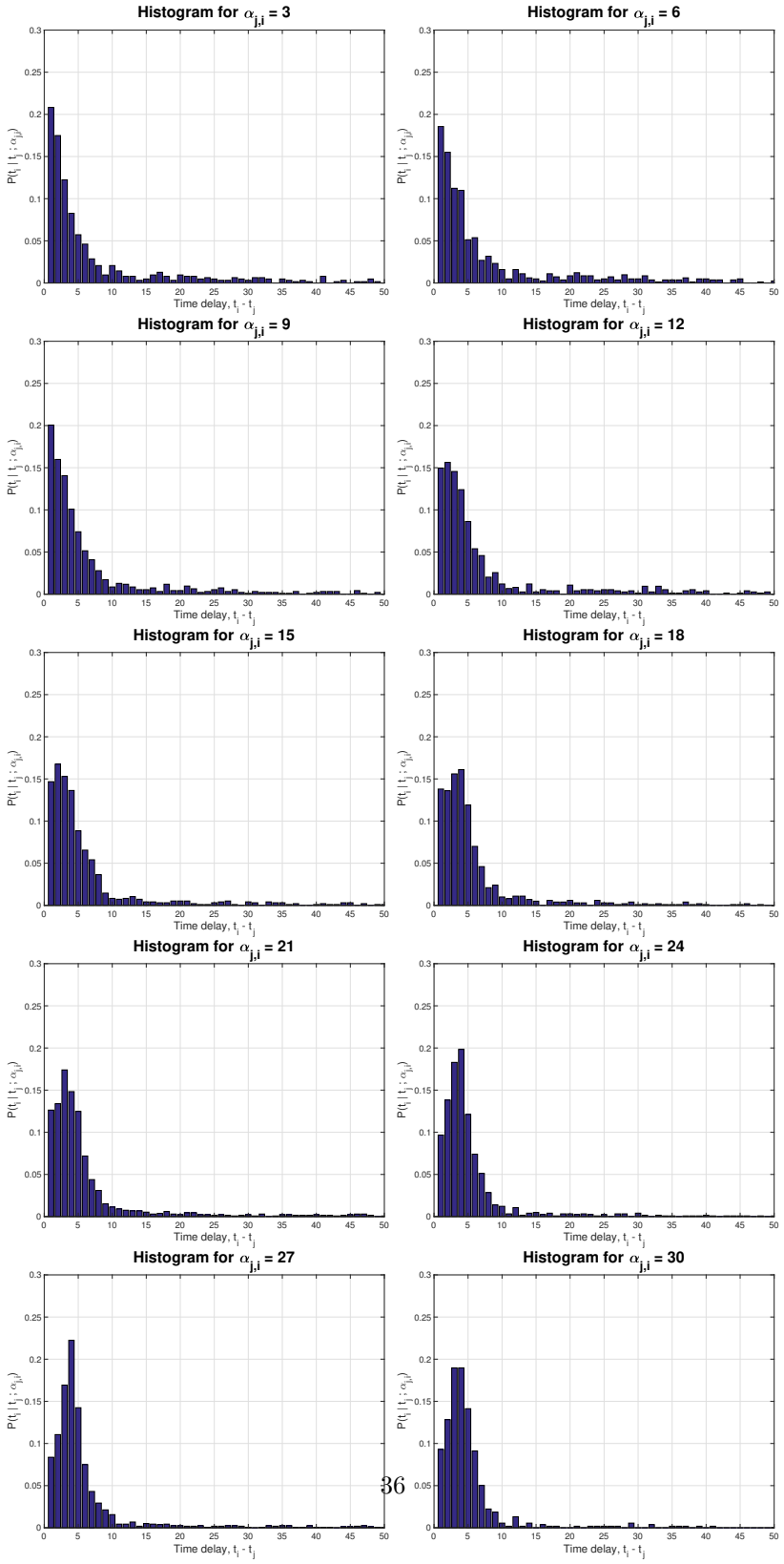


Figure 4.8: Empirical node-to-node transmission probability distribution

4.2.4 Analysis of Approximate Transmission Likelihoods

The histograms presented in figure 4.8 reveal some interesting findings. For small values of $\alpha_{j,i}$, the probability distribution is approximately Exponential whereas for large values of $\alpha_{j,i}$ the probability distribution is approximately Rayleigh. Notice the effect that the parameterising constant has on the shape of the distributions in figure 4.1: the values of λ and σ only stretch or shrink the distributions. **However, in figure 4.8, the parameterising constant changes the entire shape of the underlying distribution. Nonetheless, it is ideal that both the Exponential and Rayleigh distributions ensure that the NetRate algorithm is convex in A.**

Although the NetRate algorithm is convex for both the Exponential and Rayleigh distributions, the algorithm only works with one distribution at a time. Since we wish to identify edges that have larger weights (major edges) with greater accuracy than edges with small weights (minor edges), we shall run the NetRate algorithm using the Rayleigh distribution as the underlying probability distribution governing the propagation of action potentials through the neural network.

4.3 Modelling Biological Neural Networks as Diffusion Networks

This chapter began by listing some of the similarities and differences that exist between biological neural networks and diffusion networks. The main similarity was that both networks display observable artefacts from source nodes to target nodes that are governed by a node-to-node interaction mechanism. This is exactly the sort of network with which NetRate is meant to work. However, some apparent differences did exist.

Firstly, biological neural networks are described deterministically in the form of dynamical systems. This is not just the case for the Izhikevich neuron, which this project utilises, but for all well-recognized neuronal models. Just as most models describe neurons as dynamical systems, most models also provide a random input stimulus to excite the neuron. Since the input to the system is random, modelling the entire network probabilistically is not unrealistic. Secondly, diffusion networks are described in terms of $f(t_i|t_j;\alpha_{j,i})$. The transmission likelihoods describe how long an infected node takes to infect its neighbours. Stability analysis of the excitatory regular spiking neuron showed that $\alpha_{j,i}^{BNN}$ directly controls how long a neuron takes to induce its neighbour to

spike. Thus, neurons in biological neural networks are completely analogous to nodes in diffusion networks.

We have now shown that biological neural networks can be viewed as diffusion networks where the diffusion of action potentials from node j to node i follows a Rayleigh distribution parameterised by $\alpha_{j,i}^{BNN}$. In the following chapter, we will use NetRate to infer the connectivity of neural networks. We will find that the adapted NetRate algorithm works extremely well in identifying the physical connectivity of the network: the accuracy, precision and recall will be high. The NetRate algorithm will however not be able to correctly determine the numerical values in the network's adjacency matrix, $\alpha_{j,i}^{BNN}$. In other words, NetRate will perform poorly when the MAE is considered[2].

[2]Refer to section 2.4 for a more detailed description of how performance metrics are calculated.

Chapter 5

Network Inference Results

In chapter 2, we introduced the Izhikevich model of a neuron and NetRate, an algorithm developed to infer the temporal dynamics of diffusion networks. In chapter 3, we provided details of how simulations would be run and outlined an independent cascade generation scheme. In chapter 4, we showed that it is mathematically viable to use NetRate to infer the temporal dynamics of the biological neural networks. In this chapter, we run simulations, collect data and infer networks. The results obtained will corroborate the earlier claim that NetRate is indeed a suitable algorithm for neural network connectivity inference.

To assess how well the algorithm works, we will be utilising the four performance metrics outlined in section 2.4.6. The structure of the adjacency matrix dictates algorithmic performance. Although we restrict the scope of this project and only consider Erdős & Renyi random graphs, small differences in the adjacency matrix occurring due to the inherent randomness will determine the algorithm's performance. **For these reasons, each time a network parameter is varied, the algorithm is used to infer** 10 **unique networks: the performance metrics are then averaged over the ensemble before they are presented in this report.**

As alluded to in section 2.4.4, the NetRate algorithm requires a large enough dataset before it can accurately infer network connectivity[1]. In section 3.3.3, we created a stimulation strategy that

[1]Note that we are discussing the amount of data required to infer just one network. There is a distinction between the following argument and the argument made in the previous paragraph. In this paragraph we are talking about the amount of data needed to infer one network, whereas previously we were highlighting the need to average results across multiple networks to truly understand how the algorithm behaves when certain network parameters are varied.

would allow us to generate independent cascades.

Recall the simulation presented in figure 3.5. We supplied a node 3 with a constant input while the other nodes in the network were supplied with random noise. We initialised a cascade each time node 3 fired. We ran the simulation for 1000ms and obtained 10 independent cascades. Ten cascades do not contain enough information for us to infer a network consisting of 10 nodes. We have to generate more cascades and we have two options to do so. Both involve increasing the length of the simulation from 1000ms to 10000ms.

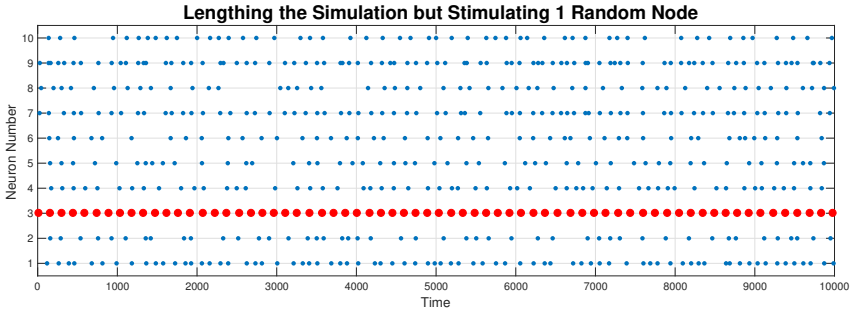


Figure 5.1: Stimulating only 1 randomly selected node but increasing the length of the simulation to extract more cascades

Our first option is to increase the length of the simulation but provide only one randomly selected node with a constant input throughout the entire duration of the simulation. This option is visually represented in figure 5.1. **Note that all cascades in our dataset would be initialised by node 3 if we were to perform the simulation in this manner.**

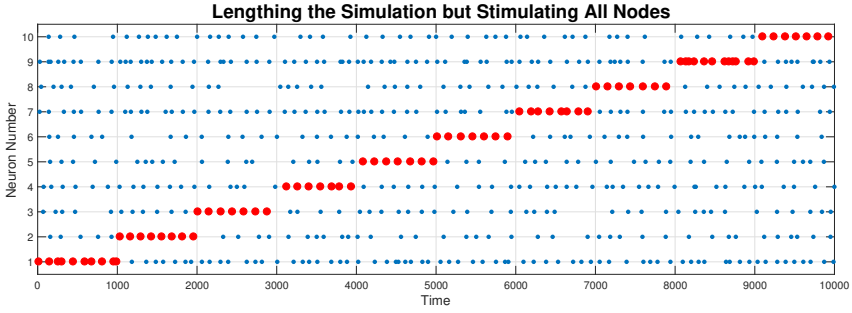


Figure 5.2: Stimulating each node in the network with a constant input for 1000ms

Our second option is to increase the length of the simulation but to provide each node in the network with a constant input for a fixed duration of time. From $0 - 1000$ms, we provide node 1 with a constant input. Within this time frame, we initialise a cascade each time node 1 fires. From $1001 - 2000$ms, we stimulate node 2 with a constant input. Within this time frame, we initialise a cascade each time node 2 fires. In this manner, by the end of the 10000ms simulation, each node in the network would have been stimulated with a constant input for 1000ms. This option is visually represented in figure 5.2. **Performing the simulation in this manner would ensure that each node in the network initialises roughly the same number of cascades.**

There is a clear disadvantage in running the simulation using the first option. The cascades generated will not contain any concrete information about the connectivity of nodes that are not direct neighbours of node 3. This exact problem was identified in section 2.4.3 when we considered a five node network. More specifically, unanswered questions four and five reveal the need for a more diverse dataset. Moreover, while this problem was evident in a 5 node network, it is greatly exacerbated when considering larger networks. Network sparsity will also amplify this problem. As a result, unless otherwise stated[2], cascades will be generated by stimulating each node in the network for 1000ms. A network consisting of 10 nodes will thus have to be simulated for a total of 10000ms whereas a network consisting of 50 nodes will have to be stimulated for 50000ms.

[2]In section 5.2.3, we consider the impact that the number of cascades has on the performance of the algorithm and in section 5.2.2 we see the damning effects of poor cascade quality. The simulations run to generate cascades will differ slightly in those specific sections.

5.1 Proof of NetRate Suitability for Biological Neural Network Inference

Before analysing how performance varies with network parameters, we must first verify that NetRate is actually a suitable algorithm for biological neural network inference. The following results support the arguments made in chapters 3 and 4. Ten networks consisting of 10 nodes each are considered. Each node within the each of the networks is stimulated for 1000ms. Approximately 143 cascades are generated for each network. The networks are then inferred using NetRate. The resulting performance metrics are averaged and presented in the graph below:

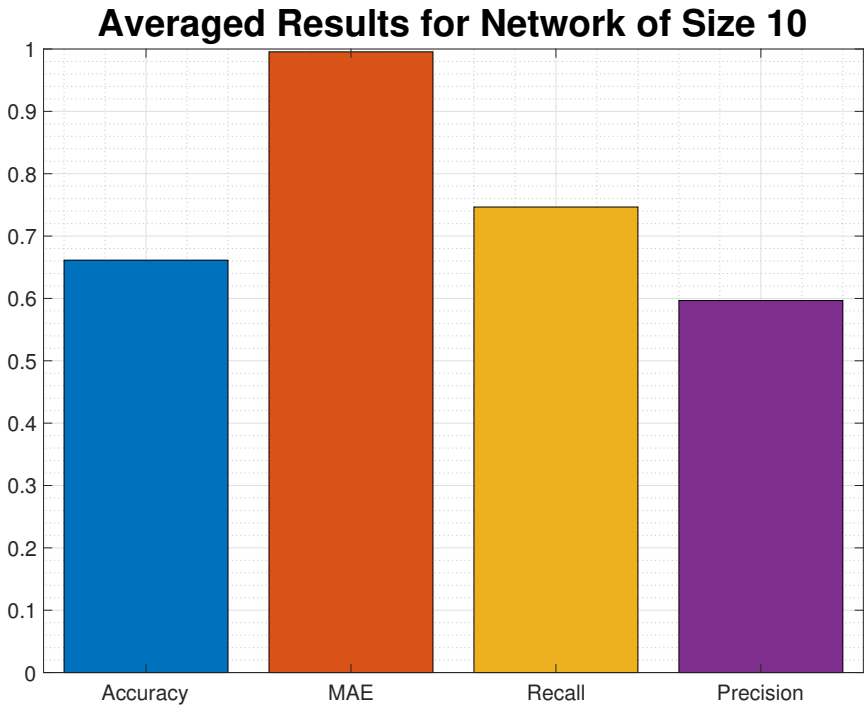


Figure 5.3: Averaged performance metrics for 10 node networks

It is evident that networks are accurately inferred. The high accuracy, precision and recall values lead us to draw this conclusion. These metrics ignore the actual weight of the connections and only focus on the existence of the inferred edges in the true network. They simply consider both the true network and the inferred network as binary graphs and compute performance accordingly.

In figure 5.4, we graph one of the 10 networks in our ensemble. This specific network is graphed as its performance is relatively close to the averages obtained in 5.3. The true network has been graphed in red whereas the inferred network has been graphed in blue. **The size of the red 'dot' representing the true network is proportional to the size of the connection, $\alpha_{j,i}^{BNN}$. However, for the inferred network, all the blue 'dots' have the same size.** This is not because NetRate has concluded that the size of all inferred connections, $\alpha_{j,i}^{BNN,*}$, is the equal. On the contrary, we have artificially set all the inferred connections to have the same weight. A detailed explanation of why this is done is presented in section 5.1.2.

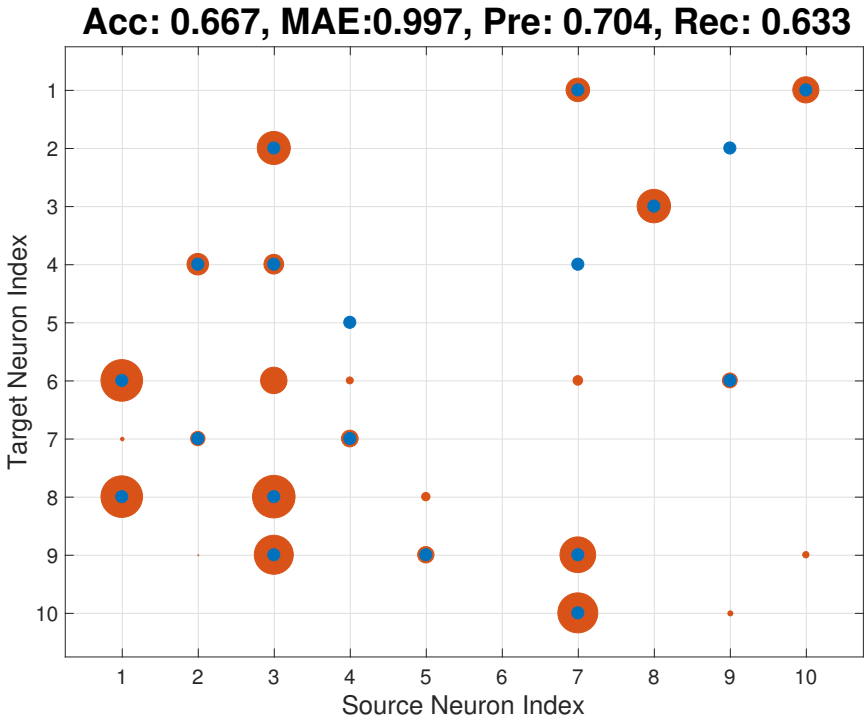


Figure 5.4: Adjacency matrix of network which performs relatively similar to the ensemble average

In figure 5.5, we plot the best (left) and the worst (right) performing networks in our ensemble. It is clear that although slight differences in the shape of the adjacency matrix affect performance, the performance does not vary widely and still primarily depends on general network parameters such as the size of the network and the number and quality of cascades generated.

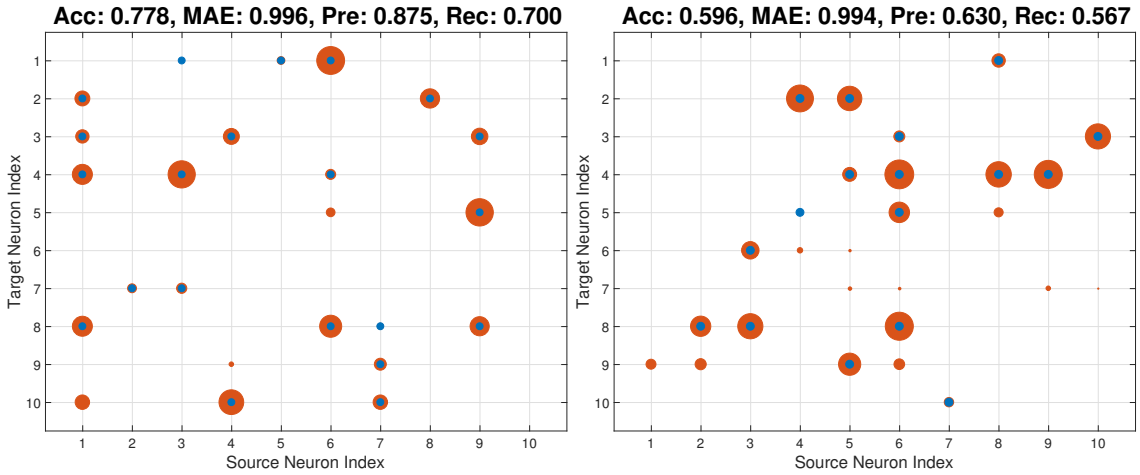


Figure 5.5: Adjacency matrices of two networks. The right network performed the best while the left performs the worst

5.1.1 Analysis of Results

It is abundantly clear that the NetRate algorithm works well. **Most importantly, the algorithm is able to identify the majority of major edges[3] in the network.** The high accuracy, precision and recall are due to the fact that the algorithm is identifying most major edges and not because it is identifying most minor edges.

The fact that the majority of major edges have been being identified is proof that the algorithm has the ability to identify the temporal dynamics of the network rather than arbitrarily, albeit successfully, guessing the location of directed edges.

It should be noted that the NetRate algorithm is a convex optimisation problem. The convex optimisation problem has been set up in a manner that naturally incorporates l_1-regularisation and thus promotes sparsity. However, numerical solutions to convex optimisation problems rarely converge to exactly zero. $\alpha_{j,i}^{BNN,*}$ are thus set to 0 if they are below a pre-determined threshold.

[3]'Major edges' refers to edges that are large in relation to the other edges in the network. Recall that all edges in the network have an associated weight that is within the range $[0, 30]$. Edges with weights above 20 are considered 'major' while edges with weights below 10 are said to be 'minor'.

5.1.2 Extraordinarily High MAE

Above, we concluded that the NetRate algorithm works extremely well and is able to infer network connectivity effectively. In this section, we will discuss the MAE.

$$\text{MAE} = \mathbb{E}\left[\frac{|\alpha_{j,i} - \alpha^*_{j,i}|}{\alpha_{j,i}}\right] \tag{5.1}$$

Equation (5.1) shows that MAE captures the absolute normalised difference between the inferred weights and the true weights. Notice that MAE is obtained by averaging the normalised difference over all directed connections that are present in both the inferred and true networks. Another point to note is that the value of MAE is not necessarily constricted to the same range as the accuracy, precision and recall. The latter are limited to the range $[0, 1]$, whereas MAE $\in \mathbb{R}$. If the true value of an edge is $\alpha_{j,i} = 0.1$ and the NetRate algorithm returns a value of $\alpha^*_{j,i} = 10$, MAE for that particular edge will be 99.

One plausible explanation for why the MAE is extraordinarily high is as follows: in diffusion networks, all entries $\alpha_{j,i}$ are limited to the range $[0, 1]$. Meanwhile in neural networks, the entries $\alpha^{BNN}_{j,i}$ are limited to the range $[0, 30]$. As such, there is a mismatch between the absolute values for which the algorithm is designed and for which it is used in this project[4]. We can try to simply multiply the weight of inferred connections by 30. Consider again the network shown in figure 5.3. We simply multiply the inferred edges by 30 and graph them on the same scale as the true edges. Notice that multiplying by 30 has a very small effect on the value of MAE. Moreover, even after multiplying by 30, it is clear that the inferred edges and the true edges still differ by one order of magnitude. This leads us to believe that there is no simple mapping between the size of the inferred edges and the size of the true edges.

[4]It should be noted that Rodriguez claims that the NetRate algorithm works for $\alpha_{j,i} \in \mathbb{R}$. However, he only provides example networks where $\alpha_{j,i} \in [0, 1]$. We tested NetRate with diffusion networks in which $\alpha_{j,i} \in [0, 100]$ and the results obtained were significantly worse that those presented in [22]. We conclude that although NetRate can work with large values of $\alpha_{j,i}$, the algorithm is intended to work with networks in which $\alpha_{j,i} \in [0, 1]$.

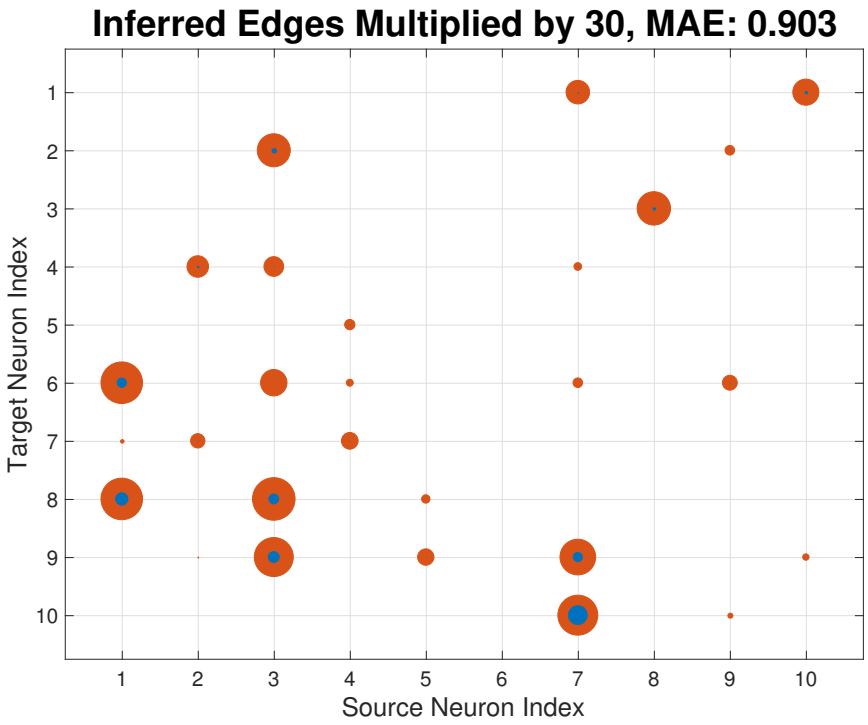


Figure 5.6: Adjacency matrix of average performing network with weight of inferred edges multiplied by 30

In fact, there is a more fundamental explanation for the extraordinarily high MAE. For a deeper understanding, we now visualise the inferred network and the true network on different plots. On each plot, the size of the 'dot' represents the normalised weight of the edge. For the true network, the maximum value of the any edge is 29.1826 and so all edges in the network are divided by 29.1826. For the inferred network, the maximum value of any edge is 0.4148 and so all edges in the network are divided by 0.4148. Also, only edges that occur in both the true and the inferred networks have been graphed[5].

[5]When MAE is calculated, the expectation is only performed over the edges that exist in both networks.

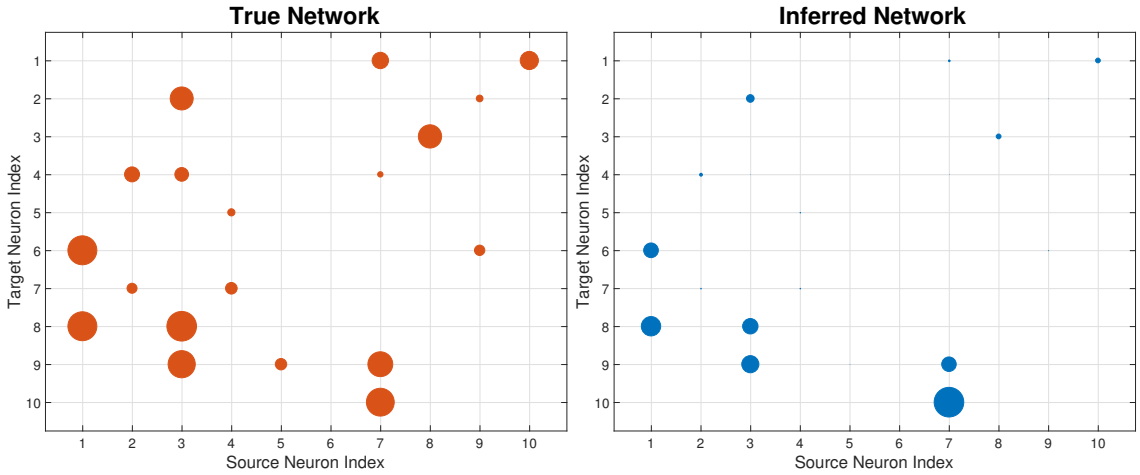


Figure 5.7: Graphing true adjacency matrix and inferred adjacency matrix on separate plots with unique scales for weight of edges

We now arrange the data in order to compare the relative size of each true edge to its corresponding inferred edge and to the rest of the edges in the given network. Figure 5.8 tells us two things:

1. For very small true edges, the inferred edges are also very small.
2. For larger true edges, there is no clear relation between the relative size of an edge in the true network and the relative size of the corresponding edge in the inferred network.

It would be ideal if the biggest edge in the true network corresponded to the biggest edge in the inferred network, however this is obviously not the case.

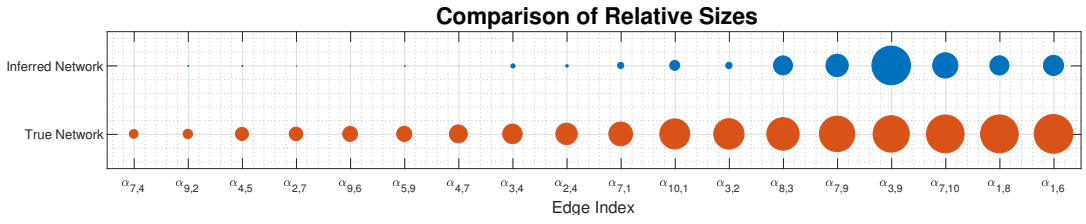


Figure 5.8: Comparison of each true edge to its corresponding inferred edge

There is an underlying reason for why the biggest edges in the true network does not correspond to the biggest edges in the inferred network. To understand this, we recall the analysis that led to the

conclusion that the NetRate algorithm can be used to infer biological neural network connectivity. In section 4.2.4, we visually approximated the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$. It was concluded that for large values of $\alpha_{j,i}^{BNN}$, the pairwise transmission likelihood follows a Rayleigh distribution. However, it was conceded that the parameter $\alpha_{j,i}^{BNN}$ does not stretch or shrink $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$. Instead $\alpha_{j,i}^{BNN}$ completely changes the general shape of $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$. **There is a clear mismatch between NetRate's prediction of the impact $\alpha_{j,i}^{BNN}$ has on $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ and the actual impact that $\alpha_{j,i}^{BNN}$ has on $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$.**

This underlying reason is why NetRate falls short at estimating the values $\alpha_{j,i}^{BNN}$. **Moreover, figure 4.8 shows that $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ has exactly the same shape regardless of whether $\alpha_{j,i}^{BNN} = 24$, $\alpha_{j,i}^{BNN} = 27$ or $\alpha_{j,i}^{BNN} = 30$. This is another reason why the algorithm has trouble differentiating the relative strength of major connections.**

5.2 General Algorithmic Performance

5.2.1 Performance for Various Network Sizes

While we have shown that the NetRate algorithm works well with networks consisting of 10 neurons, we shall now show that it also works well with larger networks. Again, for each network size, namely 15, 20 and 30, we infer an ensemble of 10 independent networks and average the results. The averaged results are presented below.

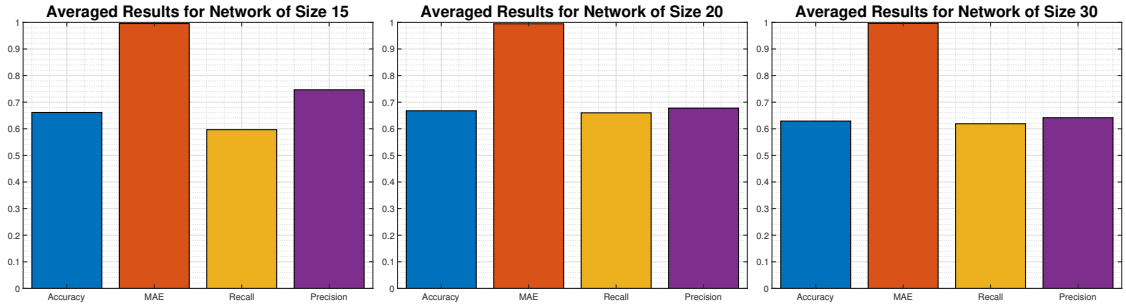


Figure 5.9: Averaged performance of NetRate on larger networks

The NetRate algorithm takes a significant amount of time to run and thus it is computationally impractical to run multiple iterations for networks larger than 30 nodes. Nonetheless, one network consisting of 50 nodes is inferred. The results presented in figures 5.9 and 5.10 show that performance does not decrease as the size of the network is increased.

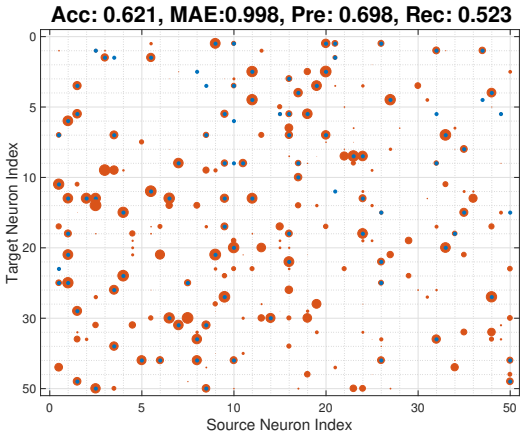


Figure 5.10: Original and inferred 50 node network

5.2.2 Performance with Quality of Cascades

Next we analyse how the algorithm's performance varies with the quality of cascades. Above we discussed two methods of obtaining sufficient cascades for effective inference. We argued that the first cascade generation scheme will generate poor cascades that lack diversity of information. To test how the quality of cascades affects algorithmic performance, we construct poor cascades using option one and perform inference. To clarify, we generate an ensemble of 10 independent networks. For each network, we stimulate node 1 with a constant input for 10000ms. We then perform inference for each of the 10 unique networks and plot the averaged performance metrics in figure 5.11.

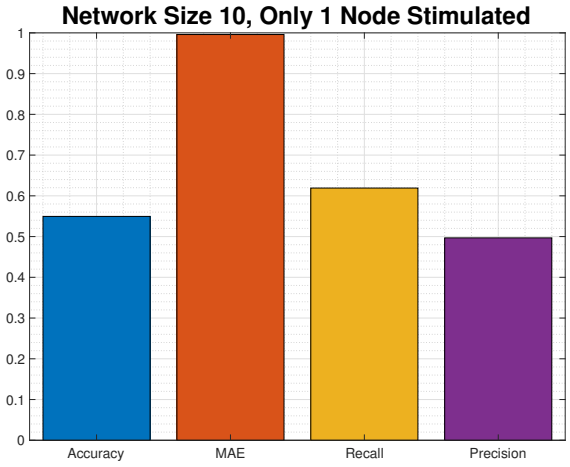


Figure 5.11: Averaged results when inference is performed by stimulating only 1 node

There is a marked decrease in accuracy, precision and recall. Furthermore, we select the best (left) and worse (right) performing networks from our ensemble and visualise them in figure 5.12. In both instances, we only supplied node 1 with a constant input while the rest of the nodes were supplied with random noise. It is evident that the majority of true connections in the best performing network are either neighbours of node 1 or neighbours of neighbours of node 1. In direct contrast, for the worst performing network, the majority of true connections have almost no direct or indirect connections to node 1. These findings are exactly as expected. **If we only introduce diseases into the networks through node** 1**, inferring connectivity about parts of the network where node** 1 **has virtually no influence is impossible.**

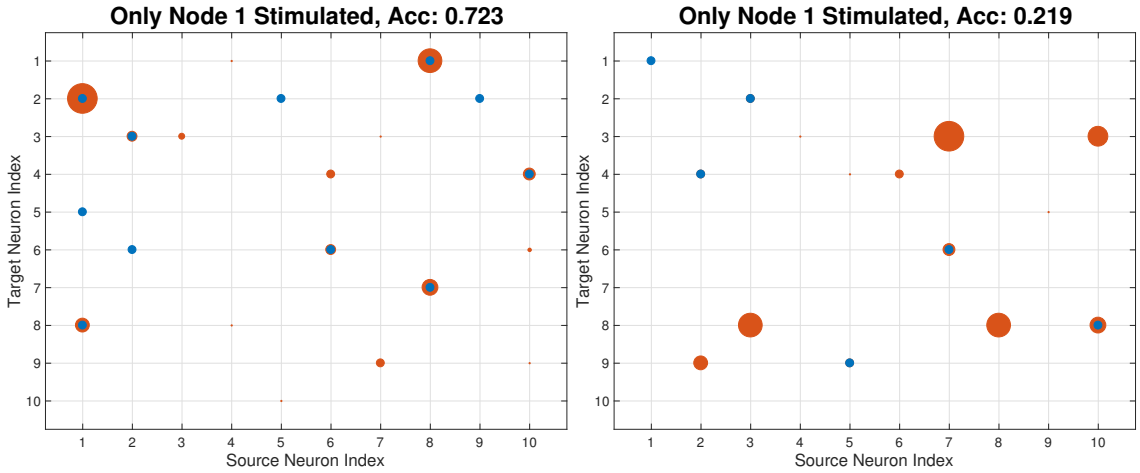


Figure 5.12: Only node 1 is stimulated. Graphs show the best performing and the worst performing networks

5.2.3 Performance with Absolute Number of Cascades

In this section, we discuss the impact that the absolute number of cascades has on algorithmic performance. To analyse performance of the algorithm with respect to the amount of data available, we run the following experiment. We first create an ensemble of 10 independent networks consisting of 10 nodes each. For each network, we **stimulate** each node in the network for the following times: $100, 150, 200, 250, 300, 350, 400, 450, 500, 1000, 2000, 3000, 4000, 5000$ms. Since testing is performed on a network consisting of 10 nodes, the total **simulation** time is in the set: $1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 10000, 20000, 30000, 40000, 50000$ms. We then infer network connectivity and plot the accuracy, recall and precision as a function of the stimulation time. Again, the accuracy, recall and precision has been averaged over an ensemble of 10 independent networks before graphing

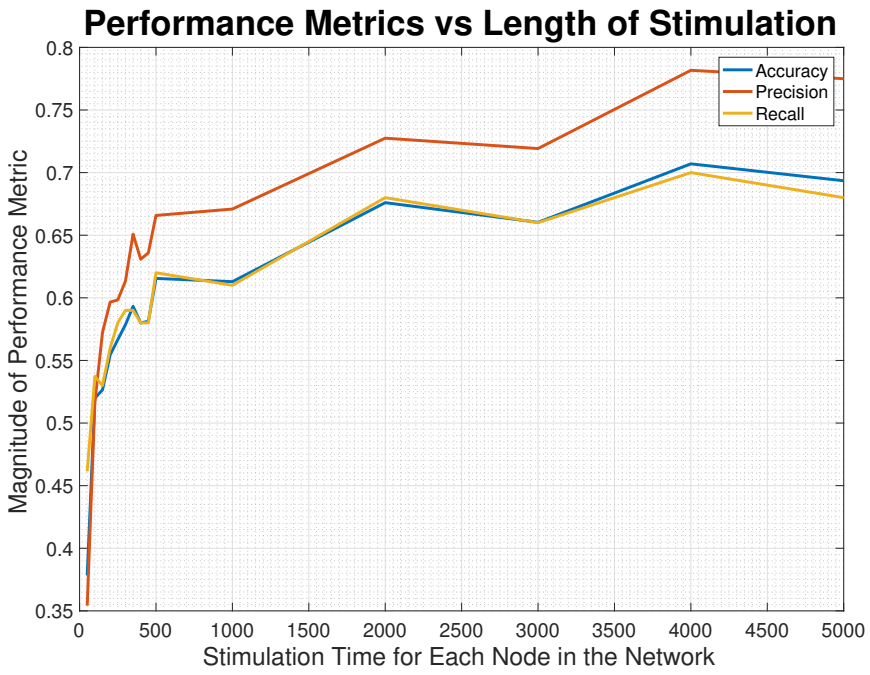


Figure 5.13: Averaged performance metrics as a function of stimulation time

The graph above corroborates the fact that performance is dependent on the absolute number of cascades. However, arbitrarily increasing the number of cascades will not result in perfect inference. This can be concluded from the fact that the graphs for all three performance metrics start to plateau. **After about** 4000**ms of stimulation, performance is limited by the inherent randomness present in the spike propagation mechanism rather than the lack of data.**

Chapter 6

Software Package

The previous chapter showed concrete results proving that the NetRate algorithm can be used to infer biological neural network connectivity. We aim to provide an easy-to-use software package that will allow users to reproduce the results presented in this report and also to input their own spike data and infer network connectivity. The inner workings of the software package are omitted from the main part of this report: essential scripts detailing how the core of the package works are provided in Appendix A. This chapter will instead cover general issues with respect to implementing the NetRate algorithm in software.

6.1 NetRate Properties

6.1.1 Unfeasible Rates

The first issue that needs to be addressed when NetRate implementation is discussed is regarding unfeasible rates. Unfeasible rates refer to pairwise transmissions rates, $\alpha_{j,i}^{BNN}$, that are not possible and thus have to be equal to 0. In section 2.1, we briefly mentioned that statistical measures of connectivity do not take into account underlying structural/anatomical models or directional effects that might make a certain connection impossible/unidirectional. However, the NetRate algorithm makes it possible to incorporate prior knowledge about connectivity. Constraints can be placed on the values of $\alpha_{j,i}^{BNN}$ to preclude certain directed connections within the network based on underlying structural/anatomical models.

Prior knowledge of the connectivity is not the only way to determine unfeasible rates. In fact, we can preclude some directed connections by directly analysing spike data. In [22], Rodriguez states that if node i does not fire after node j in any cascade, then the $\alpha_{j,i}$ term only appears in the third term[1] of the likelihood function. The optimal solution is obtained by setting $\alpha^*_{j,i} = 0$. **The NetRate algorithm thus allows us to preclude some connections by not only incorporating prior knowledge of the network but also by analysing spike data.**

6.1.2 Parallelism

One of the remarkable properties of the NetRate algorithm is its ability to be parallelised [22]. The optimisation problem presented in equation (2.9) can be broken down into N independent optimisation problems, where N represents the number of nodes. Each optimisation problem will infer $N - 1$ values within the adjacency matrix. More specifically, each optimisation problem will determiner one row of the inferred adjacency matrix. For example, we consider the optimisation problem with respect to node 1. This optimisation problem will infer the weight of all incoming connections to node 1, $\alpha^*_{j,1}$ for $j \in 2, 3, \ldots, N$. Recall that no node has a connection to itself and thus the diagonal elements of the adjacency matrix are 0.

Rodriguez provides an implementation of the NetRate algorithm in Matlab. CVX [29], a Matlab software for disciplined convex programming, is used to solve each convex optimisation problem. CVX makes use of a symmetric primal/dual solver, namely SDPT3 [30, 31], that does not natively support exponential and logarithm functions. Support for solving exponential and logarithm functions is provided in the form of a successive approximation heuristic.

Although the CVX software provides support for solving exponential and logarithm functions, the software cannot be naturally parallelised in Matlab[2]. To bypass this, it is possible to start N independent CVX convex optimisation problems using Matlab's inbuilt utility `parfor`. However, attempting to parallelise CVX in this way does not yield any speed-up.

[1] Section 2.4.4 details the 3 terms of the likelihood function. The third term is mathematically represented as $\prod_{m:t_m>T} S(T|t_j, \alpha_{j,i})$.

[2] Michael C. Grant, the creator of CVX, maintains a forum and has verified that the parallelism cannot be applied to CVX programs. His comments can be viewed at http://ask.cvxr.com/t/is-cvx-compatible-with-matlab-parallel-computing-toolbox/3305.

A Python implementation of the algorithm is attempted: using Python's `multiprocessing` library, the implementation can be significantly sped up[3]. To solve the optimisation problem, CVXPY [32], a Python-embedded modelling language for convex optimisation problems was utilised. The software ships with multiple solvers such as ECOS [33], CVXOPT [34] and SCS [35, 36]. Our Python implementation does not converge to a solution when the CVXOPT solver is utilised while algorithmic performance, measured in terms of accuracy, precision and recall, using both the ECOS and SCS solvers is poor. Performance is hindered by the fact that CVXPY does not implement the successive approximation heuristic. It should be noted that the successive approximation heuristic is a feature of CVX rather than a feature of a particular solver. For example, even though the CVX ships with SDPT3, commercial solvers such as MOSEK [37] can be used to implement the successive approximation heuristic[4].

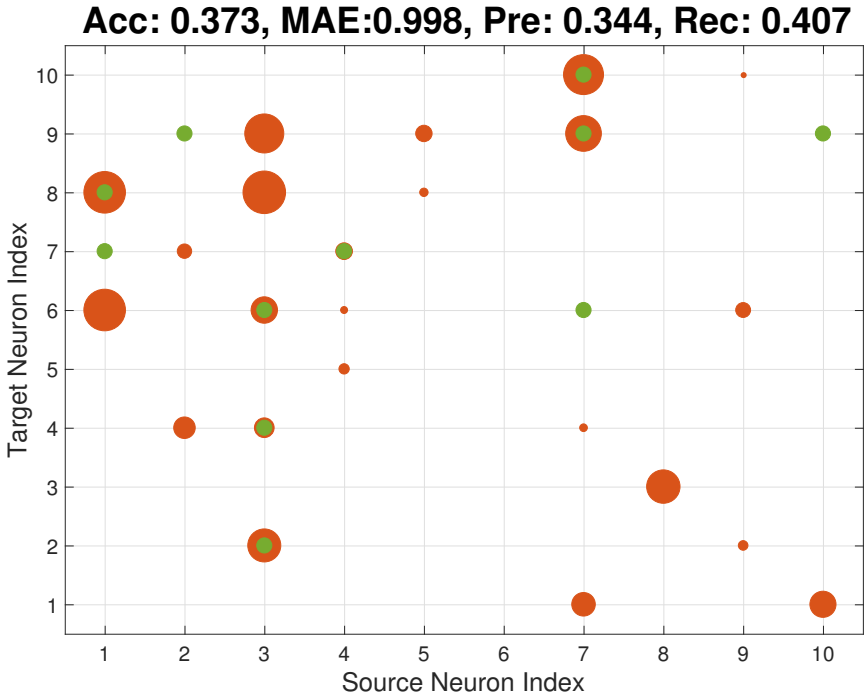


Figure 6.1: Network inference performed using Python NetRate implementation

[3]The time taken to solve a problem of the same size takes about half the time on a 2-core machine. A detailed analysis of speed-up is not provided because the implementation, albeit faster, does not yield accurate results.

[4]The successive approximation heuristic formulates an approximate model at each iteration and a solver (SDPT3 or MOSEK) is used to arrive at a solution.

We recall the graph presented in figure 5.4. It shows a network consisting of 10 nodes. The Matlab implementation of NetRate returned the following performance metrics:

1. Accuracy = 0.667
2. MAE = 0.997
3. Precision = 0.704
4. Recall = 0.633

Using exactly the same network, we generate exactly the same cascades but perform the inference using the parallel Python implementation. The inferred network is visualised in figure 6.1. The Python implementation underperforms on every performance metric. More importantly, the Python implementation fails to identify some major edges present in the network. This is directly in contrast with the Matlab implementation which identifies the majority of major edges.

6.2 Brian: Spiking Neural Network Simulator

Izhikevich introduced his model of a neuron in [15]: in this paper, he also provided a Matlab snippet that was utilised to generate the simulations presented in figures 2.1 and 3.3. However, utilising Matlab to simulate neural networks is not ideal. Brian [38] is a spiking neural network simulator. It provides a simple, easy-to-use interface to run neural network simulations. The most important reason for using Brian is the fact that the underlying model that determines the evolution of a neuron's behaviour can be easily specified in the form of differential equations. For example, consider the snippet below:

```
# Define number of neurons in the Network
N = 10

# Define the differential equations governing how the neuron works
eqs = '''
dv/dt = (0.04*v*v + 5*v + 140 - u + I + stimulation)/tau : 1
du/dt = (a*(b*v-u))/tau : 1
I = 5*randn() : 1 (constant over dt)
a : 1
b : 1
c : 1
d : 1
stimulation : 1
'''

# define threshold for spiking
```

```
17  threshold = 'v>30'
18
19  # define the reset conditions
20  reset = '''
21  v=c
22  u=u+d
23  '''
24
25  # Initialise the Neuron Group, use Euler Method for Numerical Solution
26  G = NeuronGroup(N, eqs, threshold = threshold, reset = reset, method = 'euler')
27
28  # Set Neurons Parameters
29  G.a = 0.02
30  G.b = 0.2
31  G.c = -65
32  G.d = 6
```

Listing 2: Snippet of Python code demonstrating the flexibility of Brian, the neural network simulator

The snippet is mostly self-explanatory. The code shows the ease with which the model can be altered if NetRate has to be tested with other mathematical models of spiking neurons. In addition, changing the model does not affect the way in which spike data is extracted from the simulation and the way in which cascades are generated.

Chapter 7

Conclusion and Future Works

7.1 Summary of Achievements

As part of this project, we have developed a novel method to solve the biological neural network inference problem. We started with a general algorithm developed for diffusion networks and successfully adapted it to infer neural network connectivity. The adapted algorithm was tested and its suitability was verified. Ideal properties such as the algorithm's propensity to identify major edges before minor edges were recognized and highlighted. A software package was developed that would allow verification of results provided in this report. The software package also allows smooth continuity for future works in this field.

7.2 Future Works

7.2.1 Widening the Scope

This project only studied networks consisting of excitatory regular spiking neurons. This was done to limit the scope of the project. However, the Izhikevich neuron is capable of replicating more diverse and intricate spiking behavior. Future works would involve considering networks consisting of a disparate array of different types of spiking neurons. Changing the behaviour of neurons in the networks would require formulation of an updated $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$; networks consisting of only excitatory regular spiking neurons have a $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ that is approximately Rayleigh. In fact, future works could involve formulation of a general $f^{BNN}(t_i|t_j;\alpha_{j,i}^{BNN})$ that describes the

temporal dynamics of neural networks regardless of the individual neuronal behavior.

The software package uses Brian which allows the underlying neuronal model to be changed easily. Future works could involve using the leaky integrate-and-fire neurons or using Hodgkin-Huxley neurons to form our neural network. This would serve to further verify the similarities between diffusion networks and neural networks and solidify NetRate as a primary tool to infer neural network connectivity.

7.2.2 Computational Efficiency

Computational costs were a major limiting factor when NetRate's performance was evaluated. It is not feasible to test networks with more than 30 nodes. Computationally, we are limited by the fact that CVX does allow use to exploit NetRate's inherent parallelism. Future works could involve experimentation with other convex optimisation tools.

Bibliography

[1] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[2] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 06 1998.

[3] Danielle Smith Bassett and ED Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006.

[4] Claus-C Hilgetag, Marc A O'Neill, and Malcolm P Young. Hierarchical organization of macaque and cat cortical sensory systems explored with a novel network processor. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 355(1393):71–89, 2000.

[5] Kaustubh Supekar, Vinod Menon, Daniel Rubin, Mark Musen, and Michael D Greicius. Network analysis of intrinsic functional brain connectivity in alzheimer's disease. *PLoS Comput Biol*, 4(6):e1000100, 2008.

[6] Dirk JA Smit, Cornelis J Stam, Danielle Posthuma, Dorret I Boomsma, and Eco JC De Geus. Heritability of small-world networks in the brain: A graph theoretical analysis of resting-state eeg functional connectivity. *Human brain mapping*, 29(12):1368–1378, 2008.

[7] Barry Horwitz. The elusive concept of brain connectivity. *Neuroimage*, 19(2):466–470, 2003.

[8] John G White, Eileen Southgate, J Nichol Thomson, and Sydney Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philos Trans R Soc Lond B Biol Sci*, 314(1165):1–340, 1986.

[9] Emery N Brown, Robert E Kass, and Partha P Mitra. Multiple neural spike train data analysis: state-of-the-art and future challenges. *Nature neuroscience*, 7(5):456–461, 2004.

[10] Pascal Fries. A mechanism for cognitive dynamics: neuronal communication through neuronal coherence. *Trends in cognitive sciences*, 9(10):474–480, 2005.

[11] Elad Schneidman, Susanne Still, Michael J Berry, William Bialek, et al. Network information and connected correlations. *Physical review letters*, 91(23):238701, 2003.

[12] Andrea Brovelli, Mingzhou Ding, Anders Ledberg, Yonghong Chen, Richard Nakamura, and Steven L Bressler. Beta oscillations in a large-scale sensorimotor cortical network: directional influences revealed by granger causality. *Proceedings of the National Academy of Sciences of the United States of America*, 101(26):9849–9854, 2004.

[13] Karl J Friston, Lee Harrison, and Will Penny. Dynamic causal modelling. *Neuroimage*, 19(4):1273–1302, 2003.

[14] Anthony N Burkitt. A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95(1):1–19, 2006.

[15] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070, 2004.

[16] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[17] Yehuda Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American statistical association*, 91(433):365–377, 1996.

[18] Steven L Bressler and Anil K Seth. Wiener–granger causality: a well established methodology. *Neuroimage*, 58(2):323–329, 2011.

[19] Tilman Kispersky, Gabrielle J Gutierrez, and Eve Marder. Functional connectivity in a rhythmic inhibitory circuit using granger causality. *Neural systems & circuits*, 1(1):9, 2011.

[20] Felipe Gerhard, Tilman Kispersky, Gabrielle J Gutierrez, Eve Marder, Mark Kramer, and Uri Eden. Successful reconstruction of a physiological circuit with known connectivity from spiking activity alone. *PLoS Comput Biol*, 9(7):e1003138, 2013.

[21] Amin Karbasi, Amir Hesam Salavati, and Martin Vetterli. Learning network structures from firing patterns. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 699–703. IEEE, 2016.

[22] Manuel Gomez Rodriguez, David Balduzzi, and Bernhard Schölkopf. Uncovering the temporal dynamics of diffusion networks. *arXiv preprint arXiv:1105.0697*, 2011.

[23] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.

[24] P Erdős and A Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi*, 2:482–525, 1976.

[25] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98–101, 2008.

[26] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008.

[27] Adam M Packer, Lloyd E Russell, Henry WP Dalgleish, and Michael Häusser. Simultaneous all-optical manipulation and recording of neural circuit activity with cellular resolution in vivo. *Nature Methods*, 12(2):140–146, 2015.

[28] Hsiao-Dong Chiang and Luís FC Alberto. *Stability regions of nonlinear dynamical systems: theory, estimation, and applications*. Cambridge University Press, 2015.

[29] Michael Grant and Stephen Boyd. Cvx: Matlab software for disciplined convex programming.

[30] Kim-Chuan Toh, Michael J Todd, and Reha H Tütüncü. Sdpt3a matlab software package for semidefinite programming, version 1.3. *Optimization methods and software*, 11(1-4):545–581, 1999.

[31] Reha H Tütüncü, Kim-Chuan Toh, and Michael J Todd. Solving semidefinite-quadratic-linear programs using sdpt3. *Mathematical programming*, 95(2):189–217, 2003.

[32] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 2016. To appear.

[33] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference (ECC)*, pages 3071–3076, 2013.

[34] Martin S Andersen, Joachim Dahl, and Lieven Vandenberghe. Cvxopt: A python package for convex optimization, version 1.1. 6. *Available at cvxopt. org*, 2013.

[35] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. SCS: Splitting conic solver, version 1.2.6. `https://github.com/cvxgrp/scs`, April 2016.

[36] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016.

[37] MOSEK ApS. *The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 28).*, 2015.

[38] Dan FM Goodman and Romain Brette. The brian simulator. *Frontiers in neuroscience*, 3:26, 2009.

Appendices

Appendix A

Inner Workings of Software Package

We have developed a software package that will enable others to easily verify the results presented in this report. The software package is geared to be used with macOS Sierra with limited modifications required for compatibility with standard Linux systems. Most importantly, the software package utilises Matlab to implement the NetRate algorithm. The software package also uses an array of open-source tools such as CVX (for Matlab), Krongen and the Brian Simulator which runs on Python2.7.

A.1 Entry Point

We first provide a bash script that used as the entry point into the software package. The bash script takes as an input certain network parameters as well as file names to which data and results are going to be stored.

```
#!/usr/local/bin/bash

# Created by Pranav Malhotra
# Script meant to be the main entry point of software package
# Takes network parameters as an input, generates a graph using Krongen
# Takes created graphed and runs simulation using Improved Input Stimulus Model
# Saves firing indices and times in text file
# Using Netrate, implemented in Matlab to solve the network
# Visualises the result in Python using matplotlib

seed="$1"
num_nodes="$2"
param1="$3"
```

```
param2="$4"
param3="$5"
param4="$6"
horizon="$7"
simulation_duration="$8"
diffusion_type="$9"
networkFileName="${10}"
firingsFileName="${11}"
indicesFileName="${12}"
matlabNetworkFileName="${12}"
threshold="${13}"

# print network parameters onto console
echo "seed: $seed, num_nodes: $num_nodes, krongen_parameter: [$param1 $param2; $param3 $param4] horizon: $horizon,
 ↪  simulation_duration: $simulation_duration"

# print update message to console
echo "Creating graph using krongen"

# Create kronecker graph using krongen
./krongen -o:$networkFileName -m:"$param1 $param2; $param3 $param4" -i:$num_nodes > /dev/null

# Remove headers and tabs from network created by krongen
sed -i '' '/#/d' $networkFileName;
sed -i '' 's/        /,/' $networkFileName;

# Attach probability of transition
./add_random_numbers.py $networkFileName

# print update message to console
echo "Done creating network, Running Simulation using Brian network simulator"

# run the simulation
# simulation will yield firing times and firing indices
# we store the results into files, the names of which are defined by the user
./network_simulation.py $seed $num_nodes $simulation_duration $firingsFileName $indicesFileName

# print update message to console
echo "Done with simulation, Starting Matlab to infer network using NetRate"

# open matlab and correctly assign the variables f, t, N, horizon, diffusion_type, threshold, m
# once the variables have been assigned, call pranav_network_solver
# network_solver will implement the NetRate Algorithm
# performance metrics will be printed into £resultsFileName
# inferred graph will be printed into £matlabNetworkFileName
# redirect output of command to junk, keep console clean
time matlab -nodesktop -nosplash -nojvm -r "f ='$firingsFileName'; t='$indicesFileName'; N=$num_nodes;
 ↪  horizon=$horizon; threshold=$threshold; diffusion_type='$diffusion_type'; m='$matlabNetworkFileName';
 ↪  pranav_network_solver; exit" 1> /dev/null

# print update message to console
echo "Done with Matlab, Printing Graph"

# remove first line of from £matlabNetworkFileName
# header line added by matlab
sed -i '' '1d' $matlabNetworkFileName

# graph the inferred network on top of original network
# all weights of inferred network at set to 1 (MAE is extremely poor)
# & will allow command to run in the background
# plotting is done using matplotlib in python, closing the graph will terminate bash process
./plot_adjacency.py $networkFileName $matlabNetworkFileName 0 $num_nodes &

# print update message to console
echo "Done printing graph, please close window to end background process"
```

Listing 3: Bash script that is entry point of the software package developed in this project

A.2 Brian Simulator

The following script details how the Brian simulator is used to run simulations and generate spike data.

```
#!/Users/pranavmalhotra/anaconda2/bin/python

# Created by Pranav Malhotra
# Script is used to run Izhikevich Simulation
# Script will save the indices of the fired nodes and the times at which they fired in text files

import sys
from brian2 import *
from utility import *
import numpy

# extract command line arguments
seed=int(sys.argv[1])
N=int(sys.argv[2])
simulation_duration=int(sys.argv[3])
networkFileName=sys.argv[4]
firingsNodeIndexFileName=sys.argv[5]
firingsNodeTimeFileName=sys.argv[6]

# set the seed - will determine the random noise that is generated
devices.device.seed(seed)

# suppress warnings
BrianLogger.suppress_hierarchy('brian2.codegen')

# vectors to hold indices of nodes that fire and the time at which they fire
fired_node_index=[]
fired_node_time=[]

# variable needed for brian simulator, to maintain consistency of units
tau=1*ms

# Define the differential equations governing how the neuron works
eqs= '''
dv/dt = (0.04*v*v + 5*v + 140 - u + I + stimulation)/tau : 1
du/dt = (a*(b*v-u))/tau : 1
I = 12*randn() : 1 (constant over dt)
a:1
b:1
c:1
d:1
stimulation:1
'''

# define threshold for spiking
threshold='v>30'

# define the reset conditions
reset='''
v=c
u=u+d
'''

# Initialise the Neuron Group and set the Parameters
G=NeuronGroup(N,eqs,threshold=threshold,reset=reset,method='euler')
G.a=0.02
G.b=0.2
G.c='-65+15*rand()**2'
G.d='8-6*rand()**2'

# Define the connection values and the conditions for spiking and post spiking behavior
# w is used to refer to the weight of a edge
```

```
S=Synapses(G, G, 'w:1', on_pre='v_post += w')

# call script to connect up the network as defined in the network file
pranav_connect_network(S, networkFileName)

# run the stimulation N number of times
# at each iteration, stimulate a different node
for stimulated_node in range(0,N):
        # reset v and u values
        G.v=-65
        G.u=G.b*G.v

        # only stimulate 1 node
        G.stimulation=0
        G.stimulation[stimulated_node]=3

        # create Brian simulator spike monitor
        # construct to detect and store spikes
        spikemon = SpikeMonitor(G)

        run(simulation_duration*ms)

        # store indices and time
        fired_node_index.append(list(spikemon.i))
        fired_node_time.append(list(spikemon.t/ms))

# Write Indices and Times to File
myFiringsNodeIndexFile=open(firingsNodeIndexFileName,'w')
for l in fired_node_index:
        myFiringsNodeIndexFile.write(",".join(map(str,l)))
        myFiringsNodeIndexFile.write("\n")
myFiringsNodeIndexFile.close()

myFiringsNodeTimeFile=open(firingsNodeTimeFileName,'w')
for l in fired_node_time:
        myFiringsNodeTimeFile.write(",".join(map(str,l)))
        myFiringsNodeTimeFile.write("\n")
myFiringsNodeTimeFile.close()
```

Listing 4: Python script that utilises Brian simulator to generate spike data

A.3 NetRate: Matlab Implementation

A.3.1 Matlab Entry Point

The following script is used to generate cascades from spike data. Cascades are generated in Matlab.

```
% Created by Pranav Malhotra
% Before calling script, initialise variables f, t, N, horizon, diffusion_type, threshold, m
% f: name of file with fired nodes
% t: name of file with the corresponding firing times
% N: number of nodes
% horizon: horizon after firings to consider spread
% diffusion_type: Either 'exp', 'rayleigh', 'pl'
% threshold: threshold value below which the inferred weight is set to 0
% m: name of file to store inferred matrix

%  read all the files obtained in python
firings_indices=csvread(f);
firing_times=csvread(t);

% initiliase vector to hold cascades
cascades=[];

% assuming that all nodes are stimulated
% cascades are only initialised if the stimulated node fires
for n=1:N
    % cycle through all firings
    for i=1:size(firings_indices,2)-1

        % skip if node is not the stimulated node
        if(firings_indices(n,i)~=n-1 || firing_times(n,i)==0)
            continue
        end

        % define the window of observation
        start=firing_times(n,i);
        end_of_period=start+horizon;

        % find the location of all the nodes that fired in the window
        index=find(firing_times(n,:)>start & firing_times(n,:)<end_of_period);

        % collate the fired nodes and compute the relative delays
        firings_in_window=firings_indices(n,index);
        firings_in_window(2,:)=firing_times(n,index)-start;
        firings_in_window=firings_in_window';

        % initialise a cascade, represent non-fired with -1; assume all non-fired at the start
        current_cascade=ones(1,N)*-1;

        % stimulated node starts the cascade
        current_cascade(firings_indices(n,i)+1)=0;

        % update cascade based on the rest of the firings
        for k=1:length(firings_in_window(:,1))
            % if 1 node fires twice in an observation winddow,
            % only the first one is considered
            % second firing attributed to noise
            if(current_cascade(firings_in_window(k,1)+1)==-1)
                current_cascade(firings_in_window(k,1)+1)=firings_in_window(k,2);
            end
        end

        % append current cascade to list of all cascades
        cascades=[cascades;current_cascade];
```

```
    end
end

% estimate the network
S_hat = estimate_network(cascades, N, horizon, diffusion_type);

% threshold matrix to remove extremely small values
S_hat(S_hat<threshold)=0;

% obtain estimated adjacency matrix
S_hat_adjacency=digraph(S_hat);

% write information to file
% writing adjacency in this way will append an unnecessary header line
writetable(S_hat_adjacency.Edges, m);
```

Listing 5: Matlab script that is entry point into NetRate algorithm

A.3.2 Analysis of Cascades and Identification of Unfeasible Rates

The following script rearranges cascades so that the NetRate algorithm can be solved using N sub problems, where N is equal to the number of nodes in the network. Analysis of cascades also allows unfeasible rates to be identified.

```
% Function created by Rodriguez, adapated by Pranav Malhotra
function [S_hat, total_obj] = estimate_network(cascades, num_nodes, horizon, type_diffusion)

num_cascades = zeros(1,num_nodes);
S_potential = zeros(size(A));
S_bad = zeros(size(A));
S_hat = zeros(size(A));
total_obj = 0;

% analyse cascades and arrange data to be solved by N sub-problems
% analysis will allow identification of unfeasible rates
for c=1:size(cascades, 1)
    % identify all nodes that fired then arrange
    idx = find(cascades(c,:)~=-1);
    [val, ord] = sort(cascades(c, idx));

    % loops over all nodes that were infected
    for i=2:length(val)
        % keep count of how many cascades a node appears in
        % will be used for unfeasible rates
        num_cascades(idx(ord(i))) = num_cascades(idx(ord(i))) + 1;
        % second term of likelihood function
        for j=1:i-1
            if (strcmp(type_diffusion, 'exp'))
                S_potential(idx(ord(j)), idx(ord(i))) = S_potential(idx(ord(j)), idx(ord(i)))+val(i)-val(j);
            elseif (strcmp(type_diffusion, 'pl') && (val(i)-val(j)) > 1)
                S_potential(idx(ord(j)), idx(ord(i))) = S_potential(idx(ord(j)), idx(ord(i)))+log(val(i)-val(j));
            elseif (strcmp(type_diffusion, 'ray'))
                S_potential(idx(ord(j)), idx(ord(i))) = S_potential(idx(ord(j)),
 ↪   idx(ord(i)))+0.5*(val(i)-val(j))^2;
            end
        end
    end

    % loops over all nodes that remained uninfected
    for j=1:num_nodes
        if isempty(find(idx==j))
            % third term of likelihood function
            for i=1:length(val)
                if (strcmp(type_diffusion, 'exp'))
                    S_bad(idx(ord(i)), j) = S_bad(idx(ord(i)), j) + (horizon-val(i));
                elseif (strcmp(type_diffusion, 'pl') && (horizon-val(i)) > 1)
                    S_bad(idx(ord(i)), j) = S_bad(idx(ord(i)), j) + log(horizon-val(i));
                elseif (strcmp(type_diffusion, 'ray'))
                    S_bad(idx(ord(i)), j) = S_bad(idx(ord(i)), j) + 0.5*(horizon-val(i))^2;
                end
            end
        end
    end
end

% iterate over all sub-problems
for i=1:num_nodes

    % due to unfeasible rates
    if (num_cascades(i)==0)
        S_hat(:,i) = 0;
        continue;
    end

```

```
60      % initialise cvx with sub optimisation problem
61      [s_hat, obj] = solve_using_cvx(i, type_diffusion, num_nodes, num_cascades, S_potential, S_bad, cascades);
62
63      % accumulate the total object function
64      total_obj = total_obj + obj;
65
66      % store inferred values in the matrix
67      S_hat(i,:) = s_hat';
68  end
```

Listing 6: Matlab script that analyses cascades, identifying unfeasible rates and arranging data for suitable use in sub problems

A.3.3 CVX: Solving of Sub Problems

In this script, CVX is used to solve a sub problem. The result of this sub problem will yield 1 row of the adjacency matrix

```
% Function Created by Pranav Malhotra
% Code adapted from Rodriguez NetRate Implementation
function [s_hat, obj] = solve_using_cvx(i, type_diffusion, num_nodes, num_cascades, S_potential, S_bad, C)
    cvx_begin quiet
    cvx_expert true

        % define optimisation variables
        % after the problem is solved, s_hat will become numbers
        variable s_hat(num_nodes);
        variable t_hat(num_cascades(i));

        % initialise the objective function
        obj = 0;

        % set contraint due to unfeasible rates
        s_hat(S_potential(:,i)==0) == 0;


        for j=1:num_nodes
            if (S_potential(j,i) > 0)
                % accumulate second and third term of likelihood function into the objective
                obj = -s_hat(j)*(S_potential(j,i) + S_bad(j,i)) + obj;
            end
        end

        % counting variable
        c_act = 1;

        % iterate over all cascades
        for c=1:size(C, 1)

            % identify nodes that were part of cascade
            idx = find(C(c,:)~=-1);

            % sorts the cascades so that the nodes are arranged in increasing order of infection times
            [val, ord] = sort(C(c, idx));
            idx_ord = idx(ord);

            % identify node of interest
            cidx = find(idx_ord==i);

            % ~isempty(cidx) -> if node i appeared in current cascade
            % cidx > 1 -> if node i was not the randomly chosen root node
            if (~isempty(cidx) && cidx > 1)
                % first term in likelihood function: instantaneous infection rates
                if (strcmp(type_diffusion, 'exp'))
                    t_hat(c_act) == sum(s_hat(idx_ord(1:cidx-1)));
                elseif (strcmp(type_diffusion, 'pl'))
                    tdifs = 1./(val(cidx)-val(1:cidx-1));
                    indv = find(tdifs<1);
                    tdifs = tdifs(indv);
                    t_hat(c_act) <= (tdifs*s_hat(idx_ord(indv)));
                elseif (strcmp(type_diffusion, 'ray'))
                    tdifs = (val(cidx)-val(1:cidx-1));
                    t_hat(c_act) <= (tdifs*s_hat(idx_ord(1:cidx-1)));
                end

                % add the hazard functions into the objective function,
                obj = obj + log(t_hat(c_act));

                % increment counting variable
                c_act = c_act + 1;
```

```
63              end
64          end
65
66          % sets the constraint that each element of s_hat has to be positive.
67          s_hat >= 0;
68
69          % maximise the likelihood function
70          maximize obj
71
72      cvx_end
73  end
```

Listing 7: Matlab script that uses CVX to solve sub problems

A.4 Calculation of Performance Metrics and Visualisation of Results

After the NetRate algorithm has been run, the performance metrics are calculated. Both the original network and the Matlab inferred networks are visualised using `matplotlib`. This script is meant to run in the background. The background bash process will automatically die when the `matplotlib` window is closed.

```
#!/Users/pranavmalhotra/anaconda2/bin/python

# Created by Pranav Malhotra
# Script is used to Visualise Original Network and Matlab Inferred Plot
# Performance Metrics are Calculated and Presented in Title
# Script meant to be started as Background Process
# Process will die when the Matplotlib Graph is closed

import csv
import matplotlib.pyplot as plt
import sys
import numpy as np

# extract command line arguments
networkFileName=sys.argv[1]
inferredNetworkFileName=sys.argv[2]
N=int(sys.argv[3])

# read original network
original_network=[[],[],[]]
with open(networkFileName, 'rb') as csvfile:
        spamreader = csv.reader(csvfile, delimiter=',')
        for row in spamreader:
                original_network[0].append(row[0])
                original_network[1].append(row[1])
                original_network[2].append(float(row[2])*20)

# read the inferred network
# due to matlab indexing, we need to offset the nodes
inferred_network=[[],[],[]]
with open(inferredNetworkFileName, 'rb') as csvfile:
        spamreader = csv.reader(csvfile, delimiter=',')
        for row in spamreader:
                inferred_network[0].append(int(row[0])-1)
                inferred_network[1].append(int(row[1])-1)

# create binary numpy matrices
# will be used to compute performance metrics
S=np.zeros((N,N),dtype=bool)
S_hat=np.zeros((N,N),dtype=bool)

for i,j in zip(original_network[0],original_network[1]):
        S[int(j),int(i)]=True;

for i,j in zip(inferred_network[0],inferred_network[1]):
        S_hat[int(j),int(i)]=True;

# compute the precision
# perform check to avoid divide by zero
if np.sum(S_hat)==0:
        precision=0
else:
        precision=float(np.sum(np.logical_and(S,S_hat)))/np.sum(S_hat)

```

```
# compute recall and accuracy
recall=float(np.sum(np.logical_and(S,S_hat)))/np.sum(S)
accuracy=1-float(np.sum(np.logical_xor(S,S_hat)))/(np.sum(S)+np.sum(S_hat))

# plot adjacency matrix
plt.figure(figsize=(10,7))
plt.scatter(original_network[1],original_network[0], original_network[2],c='b',label='Original Network')
plt.scatter(inferred_network[1],inferred_network[0], inferred_network[2], c='r',label='Matlab Inferred Network')

# insert performance metrics into the title
title='Matlab Results, Accuracy: {}, Precision: {}, Recall: {}'.format(round(accuracy,3), round(precision,3),
 ↪   round(recall,3))
plt.suptitle(title, fontsize=14, fontweight='bold')

# define axis labels and legend
plt.xlabel('Source neuron index')
plt.ylabel('Target neuron index')
plt.xlim(-0.5, N-0.5)
plt.ylim(-0.5, N-0.5)
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=2,  borderaxespad=0.)

# visualise plot
plt.show()
```

Listing 8: Python script that utilises `matplotlib` to compare adjacency matrix of original network and inferred network