Spatial and Temporal Epidemic Prediction by Neural Networks

Brian Lee, Nick Talati, Chen Shi

May 2022

Outline

In this project, we will investigate the modeling of epidemic spread dynamics. Our ultimate goal is to investigate various neural network architectures that can predict the evolution of disease over multiple regions in a given period of time. The project is divided into four sections:

In Section 1, we will begin with a brief treatment of the SEIR model, a common tool for describing the spread of a disease over time. We first give an overview of the traditional, deterministic model, and then explain how we have incorporated stochastic processes into the model to increase the realism. Finally, we show how the revised stochastic model can be altered to model an epidemic on a graph.

In Section 2, we give a more detailed illustration of how the stochastic SEIR model on graphs was used to generate various simulations of epidemics. Here we also introduce and explain our experimental setting, including the actual parameter values used, time scale used, and so on.

Before demonstrating the predictive performance of our neural network models on our simulated data, we give an overview of the different architectures used in this report in Section 3. Key architectures used include vanilla recurrent neural networks (RNNs), long short-term memory networks (LSTMs), as well as RNNs and LSTMs that have been modified through the use of graph-convolution operations.

Finally, in Section 4, we illustrate the predictive performance of the various architectures on our simulated data from the previous section.

1 The Epidemic Model

In this section, we describe the epidemic model used to generate our artificial data. We begin with a relatively basic variation—the deterministic SEIR model. This is then modified to make the model non-deterministic, and it is further modified to model the spatial aspect of epidemics. In the next section, we explain our simulation algorithm and parameter settings in detail.

1.1 Deterministic SEIR Model

The SEIR model [1] is an epidemiological model that predicts how a population is affected by an epidemic over time. A population of size N is partitioned into the following four compartments [8]:

- S: Susceptible individuals,
- E: Exposed individuals,
- I: Infected individuals,
- R: Recovered individuals,

so that $S + E + I + R = N$. S represents the number of people who have not yet been affected by the disease, E the number of people who have been exposed to the disease but who are not yet infective or symptomatic, I the number of people who are infective, and R the number of people who have recovered from the disease or who have been removed from the population (i.e. through death). Throughout this report, we assume that the total population does not change over the course of an epidemic, i.e. N remains constant. The values for S, E, I, and R do change, however, and thus we use the notation $S(t)$ to refer to the number of susceptible individuals at time t, and likewise for $E(t)$, $I(t)$, and $R(t)$.

In the SEIR model, individuals can transition from S to E, from E to I, and from I to R. There can be no "jump" transitions, i.e. from S to I, nor can there be any "backwards" transitions, i.e. from E to S. Movement between the four compartments is governed by the following system of ordinary differential equations:

$$\begin{aligned}
\frac{dS}{dt} &= -\frac{\beta IS}{N} \\
\frac{dE}{dt} &= \frac{\beta IS}{N} - \alpha E \\
\frac{dI}{dt} &= \alpha E - \gamma I \\
\frac{dR}{dt} &= \gamma I.
\end{aligned}$$

Using the above system, we confirm that N is constant:

$$\frac{dN}{dt} = \frac{dS}{dt} + \frac{dE}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0.$$

The three parameters α, β, and γ in the above equations control the rate at which individuals transition from one compartment to another. Specifically, α governs the rate at which people move

from E to I, with lower values indicating a longer incubation period for the disease and higher values indicating a shorter incubation period; β describes the average contact rate between persons, with higher values indicating higher average contact rates and lower values indicating lower average contact rates; finally, γ controls the rate at which people recover, with lower values indicating a longer infectious period and higher values indicating a shorter infectious period.

The remaining relevant parameters are the values of $S(0)$, $E(0)$, $I(0)$, and $R(0)$, the number of individuals in each compartment at the beginning of the epidemic. Generally, $S(0)$ will be close to N, and $E(0)$, $I(0)$, and $R(0)$ will be close to 0. We will often find it convenient to refer to $S(t)$, $E(t)$, $I(t)$ and $R(t)$ at the same time, and so we adopt the following notation:

$$\mathbf{x}(t) = \begin{bmatrix} S(t) \\ E(t) \\ I(t) \\ R(t) \end{bmatrix} \in \mathbb{Z}^4.$$

Importantly, the SEIR model described above is deterministic, meaning that $\mathbf{x}(t)$ for $t \geq 0$ is determined solely by α, β, γ, and $\mathbf{x}(0)$; there is no "randomness" in this model, which does not accurately reflect reality. Additionally, this model does not incorporate any spatial information; the spread of a disease over time is modeled, but the spread of a disease across space is not. In the next subsection, we modify the model to make it nondeterministic.

1.2 Stochastic SEIR Model

Before explaining how we transform the previous deterministic model into a stochastic model, it is important to understand *exponential distrubutions* and *Poisson processes*. From Section 2.1 of [3], a random variable T is said to have an exponential distribution with rate λ, or $T \sim \text{exponential}(\lambda)$, if

$$P(T \leq t) = 1 - e^{-\lambda t} \text{ for all } t \geq 0.$$

Such a random variable has an expected value $E[T] = 1/\lambda$ and variance $\text{var}(T) = 1/\lambda^2$. An important property of an exponentially distributed random variable T is the *lack of memory property* [3]:

$$P(T > t + s \mid T > t) = P(T > s).$$

To illustrate this property, suppose a student is awaiting a school bus, and the arrival time for the bus is an exponentially distributed random variable. The lack of memory property asserts that the probability that the bus arrives in the five minute time window beginning when the student started waiting is the same as the probability that the bus arrives within the five minute time window beginning an hour after the student is still waiting.

Additionally, if $\tau_1, \tau_2, \ldots, \tau_n$ are independent exponentially distributed random variables, each with rate λ, $T_n = \tau_1 + \cdots + \tau_n$ for $n \geq 1$, $T_0 = 0$, and $N(s) = \max\{n \mid T_n \leq s\}$, then $N(s)$ is considered to be a Poisson process. If each of $\tau_1, \tau_2, \ldots, \tau_n$ is considered an *event*, then the numbers $X_1 = \tau_1 - 0, X_2 = \tau_2 - \tau_1, \ldots$ are considered the *interarrival times* of the Poisson process [12].

We modify the deterministic SEIR model from the previous subsection through the use of Poisson processes. To illustrate, consider the interarrival time for an individual transitioning from E to I at

time t in the deterministic model; this is given by $1/\alpha E$ (since the rate is αE.) For the stochastic model, we instead sample the interarrival time from exponential(αE). However, this interarrival time will be inaccurate if a different individual transfers from S to E within this time frame (because we will not have made use of the updated value of E). To remedy this situation, at time $t = 0$, we sample the interarrival times for each of the three transitions, and we update $\mathbf{x}$ according to the transition with the minimum interarrival time, $t_{\min}$. We then repeat the process for as long as desired. The lack of memory property of exponential distributions ensures that this is a sound approach for our stochastic simulation.

The stochastic nature of the updated model is illustrated in Figure 1 below.

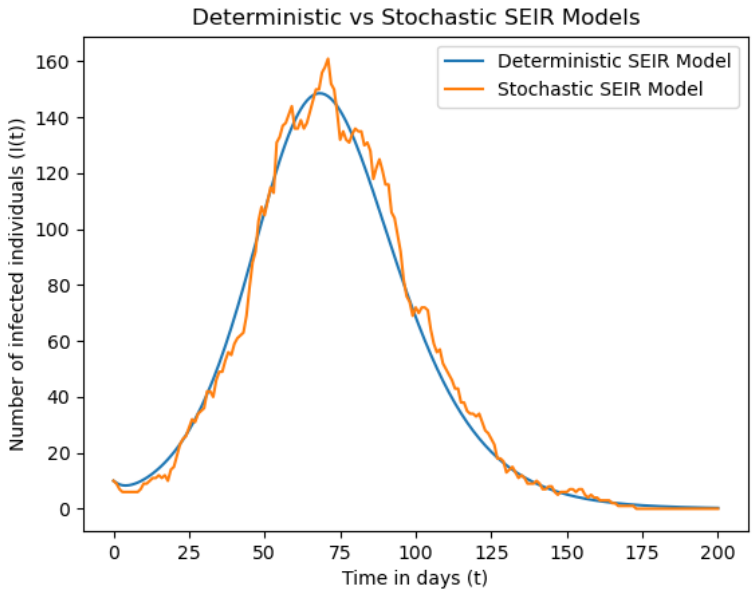

Figure 1: Comparison of deterministic and stochastic SEIR models

1.3 Stochastic SEIR Model on Graphs

To discuss the spread of an epidemic on a graph, we must first introduce new notation. Suppose we wish to model the spread of an epidemic on a graph $G = (V, E)$. Let $n = |V|$ denote the number of nodes of the graph, so that $V = \{1, \ldots, n\}$. Then we can consider each node of the graph to be a population, and so each node can contain its own values for S, E, I, and R. Thus we let $S_i(t)$ denote the value of S for node i at time t, and likewise for $E_i(t)$, $I_i(t)$, and $R_i(t)$. Similarly, we let

$$\mathbf{x}_i(t) = \begin{bmatrix} S_i(t) \\ E_i(t) \\ I_i(t) \\ R_i(t) \end{bmatrix}.$$

At any time t, we can also consider the graph signal

$$X(t) = \begin{bmatrix} \mathbf{x}_1(t)^\top \\ \vdots \\ \mathbf{x}_n(t)^\top \end{bmatrix} \in \mathbb{Z}^{n \times 4}$$

.

Both the deterministic and stochastic SEIR models can be used to describe the spread of an epidemic within a single population—but they fail to describe the spread of an epidemic across multiple, spatially distinct populations. However, this shortcoming can be remedied with relative ease through the use of graph diffusion. Intuitively, graph diffusion allows individuals from one region to mix with individuals from another region if an edge connects the two regions. For example, consider a graph of 2 connected nodes, Region 1 and Region 2. If Region 1 has 1000 infected individuals and Region 2 has 0, then because $(1, 2) \in E$, after graph diffusion Region 1 will have less than 1000 infected individuals and Region 2 will have more than 0 infected individuals. For the purpose of this report, we define graph diffusion as standard matrix multiplication:

$$\text{dif}(X(t)) = DX(t),$$

where $D \in \mathbb{R}^{n \times n}$ is a right stochastic matrix, meaning each entry is nonnegative and each row sums to 1. Additionally, if $(i, j) \notin E$, then the i, j entry of D must be 0—this ensures that no diffusion occurs between nodes that do not share an edge.

The stochastic SEIR model on graphs, then, works as follows: from $t = 0$ to $t = 1$, each node evolves independently according to the stochastic SEIR model outlined in the previous subsection. At $t = 1$, a diffusion step takes place. This process is repeated for integer values of t for the desired length of the simulation.

2 Data Simulation

In this section, we give more detailed algorithms for the simulations discussed in Section 1, and we discuss the specific parameters used to generate the dataset used for training in Section 3. To begin, we present an algorithm for simulating the stochastic SEIR model on a single node, and we then incorporate this algorithm into another that simulates the stochastic SEIR model on graphs.

2.1 Stochastic SEIR Simulation

A helpful subalgorithm for the stochastic SEIR simulation algorithm is one that generates interarrival times for a Poisson process. A method for generating arrival times for an exponentially distributed random variable (and therefore for generating interarrival times for a Poisson process) with mean μ was outlined by Knuth in section 3.4.1(D) of [11]. We modify this method slightly in Algorithm 1 by using the *rate* λ of the Poisson process as our input, rather than the mean of the underlying exponential distributions (noting that $\lambda = 1/\mu$).

Algorithm 1: Generate Poisson Process Interarrival Time

Input: Rate λ

Output: A value t giving the interarrival time for a Poisson process with rate λ

function generateInterarrivalTime(λ)
    Generate $u \sim$ `uniform(0, 1)`;
    Set $t = (-\ln u)/\lambda$;
    return t

We utilize the NumPy package [6] in our implementation of the algorithm, and we return `numpy.Inf` if the sampled u is equal to 0.

Section 3.2.2 of [2] gives an algorithm for simulating a homogeneous Poisson process with rate λ. In order to simulate the the stochastic SEIR model, we greatly modify this algorithm, closely following the description given in Section 1.2. The details are given in Algorithm 2.

Algorithm 2: Stochastic SEIR Simulation for Isolated Population

Input: Total time T, initial value vector $\mathbf{x}(0)$, and parameter values α, β, and γ

Output: An array H giving $\mathbf{x}(t)$ values for $t = 0, 1, \ldots, T$

function simulateIsolatedPopulation($T, \mathbf{x}(0), \alpha, \beta, \gamma$)
- Initialize array $H = [\,]$, array $Q = [\,]$, $t = 0$
- Initialize $S, E, I, R = S(0), E(0), I(0), R(0)$ `// Obtained from the components of` $\mathbf{x}(0)$
- **while** $t < T$ **do**
  - Add $(t, S, E, I, R)^\top$ to Q
  - Set $se_{\text{coeff}} = (\beta IS)/N$
  - Set $ei_{\text{coeff}} = \alpha E$
  - Set $ir_{\text{coeff}} = \gamma I$
  - Set t_{se} = generateInterarrivalTime(se)
  - Set t_{ei} = generateInterarrivalTime(ei)
  - Set t_{ir} = generateInterarrivalTime(ir)
  - Set $t_{\min} = \min(t_{se}, t_{ei}, t_{ir})$
  - **if** $t_{min} = t_{se}$ **then**
    - Set $S = S - 1$
    - Set $E = E + 1$
  - **else if** $t_{min} = t_{ei}$ **then**
    - Set $E = E - 1$
    - Set $I = I + 1$
  - **else if** $t_{min} = t_{ir}$ **then**
    - Set $I = I - 1$
    - Set $R = R + 1$
  - Set $t = t + t_{\min}$
- Add $(T, S, E, I, R)^\top$ to Q
- **for** $t = 0, 1, \ldots, T$ **do**
  - Sample $\mathbf{x}(t)$ from Q
  - Add $\mathbf{x}(t)$ to H
- **return** H

2.2 Stochastic SEIR Simulation on Graphs

The algorithm for simulating the stochastic SEIR model on graphs is exactly as described in Section 1.3; between integer time values, each node in the graph evolves according to the stochastic SEIR model for a single population, and at integer time values a diffusion step takes place. Algorithm 3 gives the procedure in detail.

Algorithm 3: Stochastic SEIR Simulation on Graph

Input: Total time T, initial value matrix $X(0)$, parameter values $\alpha_i, \beta_i, \gamma_i$ for $i = 1, \ldots, n$, and diffusion matrix D

Output: An array H giving $X(t)$ values for $t = 0, 1, \ldots, T$.

function simulateGraph($T, X(0), \alpha_1, \ldots, \gamma_n, D$)
- Initialize $H = [\,]$, $t = 0$
- Initialize $\mathbf{x}_i = \mathbf{x}_i(0)$ for $i = 1, \ldots, n$ `// Obtained from the components of` $X(0)$
- **while** $t \leq T$ **do**
  - Initialize matrix $X = [\,]$
  - **for** *i = 1, . . . , n* **do**
    - Set $Q =$ simulateIsolatedPopulation$(1, \mathbf{x}_i, \alpha_i, \beta_i, \gamma_i)$
    - Set $q = Q[-1]$
    - Concatenate $q^\top$ to bottom of X
  - Add DX to H
  - **for** *i = 1, . . . , n* **do**
    - Update $\mathbf{x}_i$ `// Obtained by sampling from` $H[-1]$
- **return** H

2.3 The Dataset

In order to create the dataset used for training in the next section, several decisions needed to be made. The first concerned the graph structure. While it may theoretically be possible to create a model that can generalize to various graphs, we found through experimentation that this was infeasible due to the large model sizes required to perform such a task. Because of this, we selected to use the graph structure pictured in Figure 2 for all of our simulations.

An additional consideration to be made was the actual parameter values (α, β, γ) used. For each node in each simulation, we used the following values:

- α: 0.1
- β: 0.4
- γ: 0.1

Again, consistent parameter values were used across nodes and simulations due to computational considerations; it is likely that a neural network model could generalize results to data generated from more varied parameter values, but such models would too large to train in a reasonable amount of time given our computational resources. The exact diffusion matrix D used in each simulation

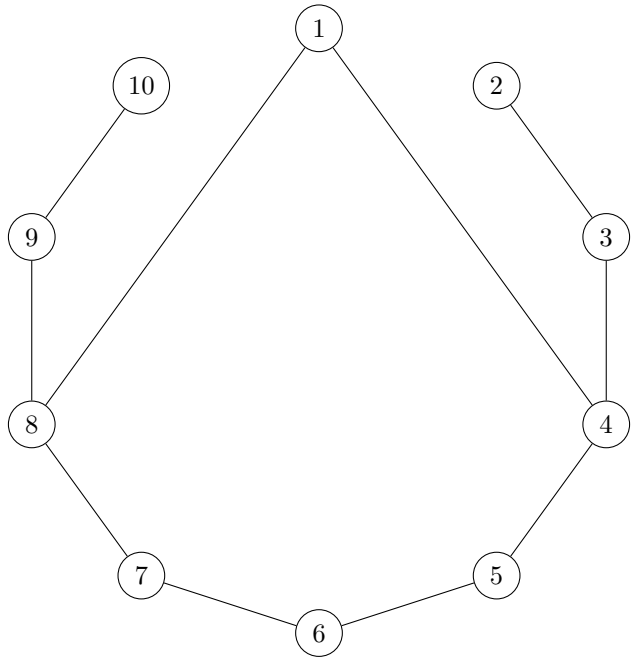

Figure 2: Graph structure used to generate training data

is as follows:

$$D = \begin{bmatrix} 0.92 & & & 0.04 & & & & 0.04 & & \\ & 0.96 & 0.04 & & & & & & & \\ & 0.04 & 0.92 & 0.04 & & & & & & \\ 0.04 & & 0.04 & 0.88 & 0.04 & & & & & \\ & & & 0.04 & 0.92 & 0.04 & & & & \\ & & & & 0.04 & 0.92 & 0.04 & & & \\ & & & & & 0.04 & 0.92 & 0.04 & & \\ 0.04 & & & & & & 0.04 & 0.88 & 0.04 & \\ & & & & & & & 0.04 & 0.92 & 0.04 \\ & & & & & & & & 0.04 & 0.96 \end{bmatrix},$$

and the initial values, $\mathbf{x}_i(0)$ chosen were all as follows:

$$\mathbf{x}_1(0) = \begin{bmatrix} 49,990 \\ 0 \\ 10 \\ 0 \end{bmatrix} \text{ for } i = 1 \text{ and } \mathbf{x}_i(0) = \begin{bmatrix} 50,000 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ for } i = 2, \ldots, 10.$$

We ran a total of 200 simulations with the above parameters, each for a total of 50 time steps. Figure 3 plots the number of infected individuals for Node 1 in each simulation; we note that the

stochastic nature of the simulation ensures that a large spread exists in the numbers of infected individuals over the course of the first 50 time steps. This allows our prediction task to remain non-trivial, despite the use of consistent parameter values across simulations.

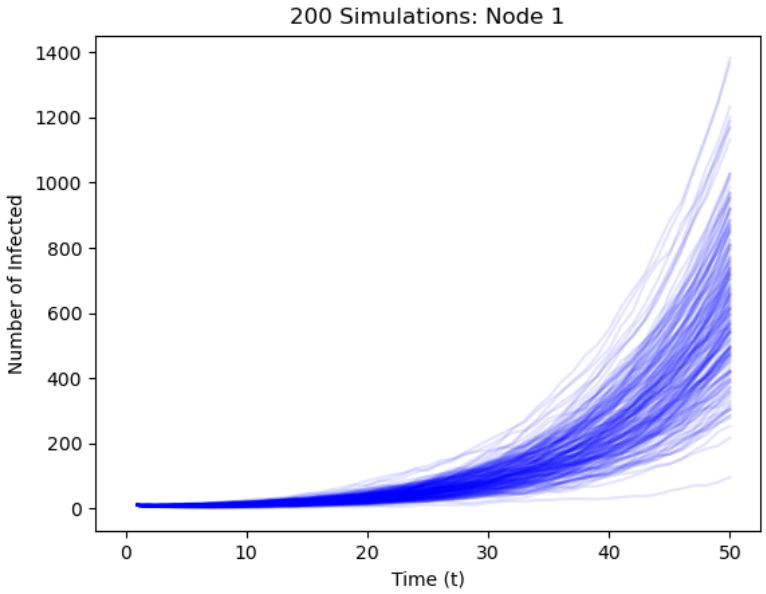

Figure 3: Number of infected for Node 1, 200 simulations

3 Neural Network Models

In this section we give an overview of several different neural network architectures. We begin with an outline of recurrent neural networks and long short-term memory networks, before turning to versions of each of those which incorporate graph convolutional operations.

3.1 Baseline Model

To establish a baseline with which to compare the performance of our neural network models, we first describe a classical approach to predicting future data. With this approach, we infer the values of parameters β, α, γ from the given data in each node and then use forward Euler method to solve the ordinary differential equations for SEIR model numerically.

Recall from Section 1 that the deterministic SEIR model for a single region is described by the following system of ODEs:

$$\frac{dS}{dt} = -\frac{\beta IS}{N}$$
$$\frac{dE}{dt} = \frac{\beta IS}{N} - \alpha E$$
$$\frac{dS}{dt} = \alpha E - \gamma I$$
$$\frac{dR}{dt} = \gamma I.$$

Given $\alpha, \beta, \gamma, S(t), E(t), I(t)$, and $R(t)$, we can compute the right-hand side of each of the above equations. We can then estimate $S(t + \Delta t), E(t + \Delta t), I(t + \Delta t)$, and $R(t + \Delta t)$ via linear approximation:

$$S(t + \Delta t) \approx S(t) + \frac{dS}{dt}\Delta t$$
$$E(t + \Delta t) \approx E(t) + \frac{dE}{dt}\Delta t$$
$$I(t + \Delta t) \approx I(t) + \frac{dI}{dt}\Delta t$$
$$R(t + \Delta t) \approx R(t) + \frac{dR}{dt}\Delta t.$$

When the above step is performed iteratively, we can estimate the values of S, E, I, and R for any future time step. These estimations will be more accurate when the "step size" Δt is smaller. This approach to numerically estimating the system of ODEs is known as the forward Euler method.

Our baseline model works by choosing α, β, and γ in each node so that, given the first 20 days of S, E, I, and R data, the difference between the deterministic simulation (produced using the forward Euler method) that results from using these parameters and the actual simulation is minimized. We measure the difference between the two simulations through the sum of the squared errors; if $(\mathbf{a})_i = (\mathbf{a}(1), \ldots, \mathbf{a}(20))$ is one simulation and $(\mathbf{b})_i = (\mathbf{b}(1), \ldots, \mathbf{b}(20))$ is another, then

$$\text{dif}((\mathbf{a})_i, (\mathbf{b})_i) = \sum_{i=1}^{20} \|\mathbf{a}(i) - \mathbf{b}(i)\|^2.$$

The baseline model uses scipy.optimize.minimize to return the values for α, β, and γ that minimize this difference for each node, and from there the forward Euler method is used to predict future values.

3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of artificial neural networks that are able to process variable-length input sequences. This makes them incredibluy useful for predicting on time-series data, and so we investigate their usefulness in predicting on our simulated epidemic data. RNNs are

especially useful for processing sequential data because they maintain what is known as a *hidden state*. When a sequence is inputted into an RNN, each element is processed in order; as the sequence is fed into the network in this fashion, the hidden state , which takes the form of a one-dimensional array or vector, is updated. Additionally, the network produces an output each time an element of the sequence is fed into it. Over the course of training, the network learns to encode information about each element in a sequence into the hidden state and produce outputs based on this. Thus, if our sequence has the following form:

$$(\mathbf{x})_n = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n,$$

then the RNN will process $\mathbf{x}_1$ to begin, it will update its hidden state, then it will process $\mathbf{x}_2$, update its hidden state once again, and so on. By the time the network processes $\mathbf{x}_n$, the hidden state has been updated based on all n elements in the sequence, and so in theory the hidden state still contains information about each element of the sequence. Each element in a sequence to be processed by a standard RNN is a vector. Recall that our sequential data takes the form of a matrix; a single simulation J can be represented as follows:

$$J = (X(0), X(1), \ldots, X(50)),$$

where

$$X(t) = \begin{bmatrix} S_1(t) & E_1(t) & I_1(t) & R_1(t) \\ & & \vdots & \\ S_{10}(t) & E_{10}(t) & I_{10}(t) & R_{10}(t) \end{bmatrix} \in \mathbb{R}^{10\times 4}.$$

In order to use our simulation data as input to a standard RNN, we must first *vectorize* each element of our simulation sequence. After this step, our new simulation J' has the following form:

$$J' = (\mathbf{X}(0), \mathbf{X}(1), \ldots, \mathbf{X}(50)),$$

where

$$\mathbf{X}(t) = (S_1(t), E_1(t), I_1(t), R_1(t), S_2(t), \ldots, R_{10}(t))^\top \in \mathbb{R}^{40}.$$

With each simulation in the above format, subsequences can be sampled from each simulation and used for training; this process will be explained in more detail in Section 4.2.

The architecture of an RNN is described in the following formulas:

$$\begin{aligned} \mathbf{h}_t &= \tanh(W(\mathbf{h}_{t-1} \mid \mathbf{X}(t)) + \mathbf{b}) \\ \mathbf{o}_t &= V\mathbf{h}_t + \mathbf{c}, \end{aligned}$$

where $\mathbf{h}_t$ is the hidden state at time t and $\mathbf{o}_t$ is the output at time t; $(\cdot \mid \cdot)$ represents vector concatenation, i.e. if $\mathbf{a} = (a_1, \ldots, a_n)^\top$ and $\mathbf{b} = (b_1, \ldots, b_m)^\top$, then

$$(\mathbf{a}|\mathbf{b}) = (a_1, \ldots, a_n, b_1, \ldots, b_m)^\top.$$

Thus, each time an element $\mathbf{X}(t)$ of the sequence is inputted into the RNN, it is first combined with the hidden state from the previous step via vector concatenation (the hidden state is initialized to all zeros for the first element). Then an affine transformation is applied to the concatenated vector, followed by a non-linear tanh (applied element-wise). This results in the updated hidden state; the output at this step is generated through an affine transformation of the hidden state. The matrices W and V contain what are known as the *weights* of the network, and $\mathbf{b}$ and $\mathbf{c}$ contain the *biases*. All elements in these components are trainable parameters, i.e. W, V, $\mathbf{b}$, and $\mathbf{c}$ are updated as the RNN is trained. Given $\mathbf{X}(t-s), \ldots, \mathbf{X}(t)$, we will find it useful to refer to the final output $\mathbf{o}_t$ as

$$\mathbf{o}_t = F_{\text{RNN}}(\mathbf{X}(t-s), \ldots, \mathbf{X}(t)),$$

where the subscript gives relevant information about the network.

It should be noted that the hidden state can be of arbitrary dimension, and that an increased hidden dimension necessarily leads to an increased number of parameters and therefore to an increase in computational complexity. Analyzing the changes in performance that result from different hidden dimensions will be one of the topics covered in the next section.

We want an RNN that predicts $\mathbf{X}(t+1)$ given $\mathbf{X}(t-s), \ldots, \mathbf{X}(t)$; to do this, we train the neural network to minimize the *loss function* L:

$$\begin{aligned} L &= \sum \ell(\mathbf{o}_t, \mathbf{X}(t+1)) \\ &= \sum \ell(F_{\text{RNN}}(\mathbf{X}(t-s), \ldots, \mathbf{X}(t)), \mathbf{X}(t+1)), \end{aligned}$$

where ℓ is a function that measures the discrepancy between the "prediction" $\mathbf{o}_t$ and the true value $\mathbf{X}(t+1)$, and the sum is taken over all outputs for all simulations in our set of training simulations (more on this in the next section). Training the RNN amounts to finding the W, V, $\mathbf{b}$ and $\mathbf{c}$ which minimize L. The particular algorithm for this process is known as backpropagation through time, a gradient-based optimization technique that is outlined in [5].

One of the limitations of standard RNNs is that they have a limited "memory"; because the hidden state is updated at every step (through concatenation with the current element, an affine transformation, and the nonlinear tanh function), information about the first few elements in a sequence is gradually eroded as new elements are inputted into the network. The architecture introduced in the next section aims to alleviate some of these difficulties.

Figure 4 illustrates an unfolded recurrent network that has connections between hidden units, mapping an input sequence $\boldsymbol{x}_j, \boldsymbol{x}_{j+1}, \ldots, \boldsymbol{x}_{j+\ell-1}$ to a single output $\boldsymbol{y}_{j+\ell}$. The loss function L measures the discrepancy between the prediction $\hat{\boldsymbol{y}}_{j+\ell}$ and its target $\boldsymbol{y}_{j+\ell}$.

3.3 Long Short-Term Memory Networks

Long Short-term Memory Networks (LSTMs) introduce several modifications to the standard RNNs described in the previous subsection and were introduced by Sepp Hochreiter and Jürgen Schmidhuber[7]. These modifications enable the prediction at one time step to utilize information from received in the more distant past. The primary modification is the introduction of a *cell state* in addition to the hidden state. A cell state is akin to a hidden state in that it encodes information

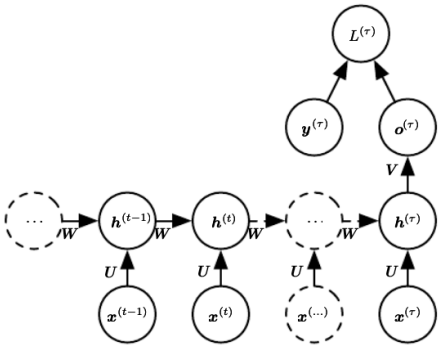

Figure 4: Unfolded Structure of a Recurrent Neural Network

about previously inputted elements in a sequence, but it is different in how it is updated. Whereas the hidden state in a standard RNN is updated at each time step via an affine transformation and nonlinear activation function, the degree to which the cell state in an LSTM is updated may vary from input to input, and the amount that the cell state should change based on any given input is learned during training. Practically speaking, this means that an input to the LSTM which is deemed "unimportant" by the network will not cause the cell state to significantly update, whereas an input that is deemed especially important will cause the cell state to update significantly.

LSTMs utilize three gates to control the flow of information in the network: a forget gate, an input gate, and an output gate. The forget gate determines which parts of the cell state to keep, and which parts to discard; the input gate determines what new information should be added to the cell state, and the output gate determines what information to output from the cell state. The equations detailing the architecture of an LSTM are:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(W_f(\mathbf{X}(t) \mid h_{t-1}) + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(W_i(\mathbf{X}(t) \mid h_{t-1}) + \mathbf{b}_i) \\
\tilde{\mathbf{c}}_t &= \tanh(W_c(\mathbf{X}(t) \mid h_{t-1}) + \mathbf{b}_c) \\
\mathbf{p}_t &= \sigma(W_p(\mathbf{X}(t) \mid h_{t-1}) + \mathbf{b}_p) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{p}_t \odot \tanh(\mathbf{c}_t) \\
\mathbf{o}_t &= V\mathbf{h}_t + \mathbf{c}
\end{aligned}
$$

where $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ is the input gate, $\tilde{\mathbf{c}}_t$ gives *candidate values* for the new cell state, $\mathbf{p}_t$ is the output gate, $\mathbf{c}_t$ is the cell state, $\mathbf{h}_t$ is the hidden state, and $\mathbf{o}_t$ gives the final output for the LSTM. Note that σ represents the sigmoid function and $\odot$ element-wise multiplication.

Each of the three gates is a vector whose elements are real numbers in the interval $[0, 1]$. The gates are multiplied be the other relevant vectors and determine which information to retain. The forget gate is multiplied by the cell state, the input gate is multiplied by the candidate values for the new cell state, and the output gate is multiplied by the cell state itself to determine what information to pass to the hidden state. The last equation, $\mathbf{o}_t = V\mathbf{h}_t + \mathbf{c}$, gives the final output of the LSTM.

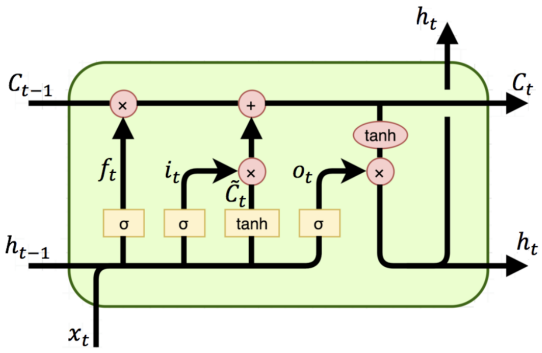

Figure 5: An LSTM Cell

The loss function used for training an LSTM is identical to that used for the standard RNN:

$$\begin{aligned} L &= \sum \ell(\mathbf{o}_t, \mathbf{X}(t+1)) \\ &= \sum \ell(F_{\text{LSTM}}(\mathbf{X}(t-s), \ldots, \mathbf{X}(t)), \mathbf{X}(t+1)), \end{aligned}$$

and training uses the same backpropagation through time algorithm that RNNs use.

Compared to standard RNNs, LSTMs have more trainable parameters; W_f, W_i, W_c, W_p, V, $\mathbf{b}_f$, $\mathbf{b}_i$, $\mathbf{b}_c$, $\mathbf{b}_p$, and $\mathbf{c}$ make up the entire set in an LSTM, while only W, V, $\mathbf{b}$ and $\mathbf{c}$ form the set of trainable parameters in an RNN. Thus, while the cell state and gates used in an LSTM enable the network to have a longer working "memory," this does come at the cost of increased computational complexity.

3.4 Graph Convolutional RNNs and LSTMs

One of the potential drawbacks of using standard RNNs and LSTMs as described in the previous section is that information about the underlying graph structure is nowhere utilized; the inputted data contain all relevant information about the number of individuals within each compartment in every node on the graph, but nothing about the graph itself, and this information is also not encoded into the network architectures. However, for real-world data, this information is usually at least partially accessible. For instance, if real-world epidemic data was obtained for each of the fifty United States, then the underlying graph structure may be estimated based on the geographic distances between states.

The architectures of the next two models incorporate the structure of the underlying graph which was used to produce the data. To allow for this, we replace all matrix multiplication operations in the equations for RNNs and LSTMs with graph convolution, as outlined in [10]. This operation utilizes the graph structure and is parameterized by far fewer parameters than matrix multiplication. We refer to the models outlined in the equations below as Graph Convolutional Recurrent Neural Networks (GCRNNs) and Graph Convolutional Long Short-Term Memory Networks (GCLSTMs), respectively.

GCRNN equation:

$$H_t = \sigma(g_H \star (X(t) \mid H_{t-1}) + B_H)$$

GCLSTM equations:

$$\begin{aligned}
F_t &= \sigma(g_F \star (X(t) \mid H_{t-1}) + B_F) \\
I_t &= \sigma(g_I \star (X(t) \mid H_{t-1}) + B_I) \\
\tilde{C}_t &= \tanh(g_C \star (X(t) \mid H_{t-1}) + B_C) \\
P_t &= \sigma(g_P \star (X(t) \mid H_{t-1}) + B_P) \\
C_t &= F_t \odot C_{t-1} + I_t \odot \tilde{C}_t \\
H_t &= P_t \odot \tanh(C_t) \\
O_t &= \text{unvec}(V \,\text{vec}(H_t) + \mathbf{c})
\end{aligned}$$

In the above equations, $(\cdot \mid \cdot)$ represents matrix concatenation and $g_A \star B$ represents a graph convolution of a signal $B \in \mathbb{R}^{n \times F}$ with a filter parameterized by A (see [10]). The last equation represents a standard fully-connected layer without an activation function (vec and unvec represent matrix vectorization and un-vectorization, respectively).

An important distinction between the architectures described above and the standard ones explained in the previous section concerns the domain of the relevant data elements. In standard RNNs and LSTMs, every gate and hidden state (including the cell state) are vectors in $\mathbb{R}^h$, where h is the hidden dimension. In GCRNNs and GCLSTMs, all gates and hidden states are matrices in $\mathbb{R}^{n \times h}$, where n is the number of nodes and h is the hidden dimension. For our experimental setup, this means that a GCRNN (or GCLSTM) has a hidden state with ten times the dimensionality (since we use 10 nodes) of a standard RNN (or LSTM) with the same hidden dimension. For this reason, we specify the complexity of the networks by referring to the total number of hidden *units*; this number is the same as the hidden dimension for standard RNNs and LSTMs, but is greater than the hidden dimension by a factor of 10 for GCRNNs and GCLSTMs.

4 Training and Results

4.1 Generating Training Data

All neural networks tested were trained on sequences of length 21 that were sampled from the 100 training simulations, i.e. each training sequence is of the form

$$(\mathbf{X}(t))_{t=s}^{t=s+20} = (\mathbf{X}(s), \mathbf{X}(s+1), \ldots, \mathbf{X}(s+10))$$

for some starting time s. (This is for the standard RNNs and LSTMs. For their graph convolutional counterparts, each $\mathbf{X}$ is replaced by X.)

From each of the 100 training simulations, six sequences of length 21 were sampled randomly, for a total of 600 training sequences. The same sampling was performed on the validation simulations, for a total of 300 validation sequences. Before training, every sequence was also normalized by

dividing each element by the total population within each node, 50,000. We found that this step, which ensures all compartmental values are in the interval $[0, 1]$, significantly decreased the required training time.

4.2 Training Details

Let $\mathcal{X}_{\text{train}}$ denote our set of training sequences, and let

$$(x)_i = (x_0, x_1, \ldots, x_{20})$$

represent a training sequence in $\mathcal{X}_{\text{train}}$. Each element of the training sequences used for RNNs and LSTMs will be of the form $\mathbf{X}(t)$, while each element of the training sequences used for GCRNNs and GCLSTMs will be of the form $X(t)$. Then we define our total training loss L_{train} for each model F_{model} to be the average square error of the model's prediction *for each of the 20 outputs in every sequence*, across all training sequences; thus

$$L_{\text{train}}(F_{\text{model}}) = \frac{1}{20 \cdot 600} \sum_{(x)_i \in \mathcal{X}} \sum_{i=1}^{20} \|x_i - F_{\text{model}}(x_0, \ldots, x_{i-1})\|^2.$$

All model weights were initialized using Xavier Initialization, described in [4], and all hidden and cell states were zero-initialized. The Adam optimizer from [9] was used for training with $\beta_1 = 0.9, \beta_2 = 0.999$, and $\varepsilon = 1 \times 10^{-8}$. The learning rate was set to 0.001 for the first epoch of training and decayed by a factor of 0.9 for each subsequent epoch. Training was performed until the validation loss L_{val}, calculated the same was as the training loss L_{train} above, stabilized. We found this to occur at or before epoch 60 for all models tested. Our set of training sequences was shuffled prior to each epoch during training, and we used a batch size of one for all models.

The training and validation loss for a GCLSTM with 200 hidden units is plotted in Figure 6; we note that the losses have been un-normalized to approximately reflect average absolute errors in compartmental number predictions.

4.3 Results

4.3.1 Evaluation Metric

We evaluate all models by comparing their predictions for the number of infected individuals at time $t = 50$ with the ground truth, across all nodes and all testing simulations. We denote model F_{model}'s predicted number of infected individuals at time $t = 50$ in node i for simulation $J = (X(0), \ldots, X(50))$ by $\hat{I}_{i,J,\text{model}}$. The predicted values are obtained via Algorithm 4, and for our evaluation metric we simply take the mean absolute error across all nodes and simulations:

$$L_{\text{eval}}(\text{model}) = \frac{1}{10|\mathcal{T}|} \sum_{J \in \mathcal{T}} \sum_{i=1}^{10} |I_{i,J} - \hat{I}_{i,J,\text{model}}|,$$

where $\mathcal{T}$ denotes our set of 50 testing simulations.

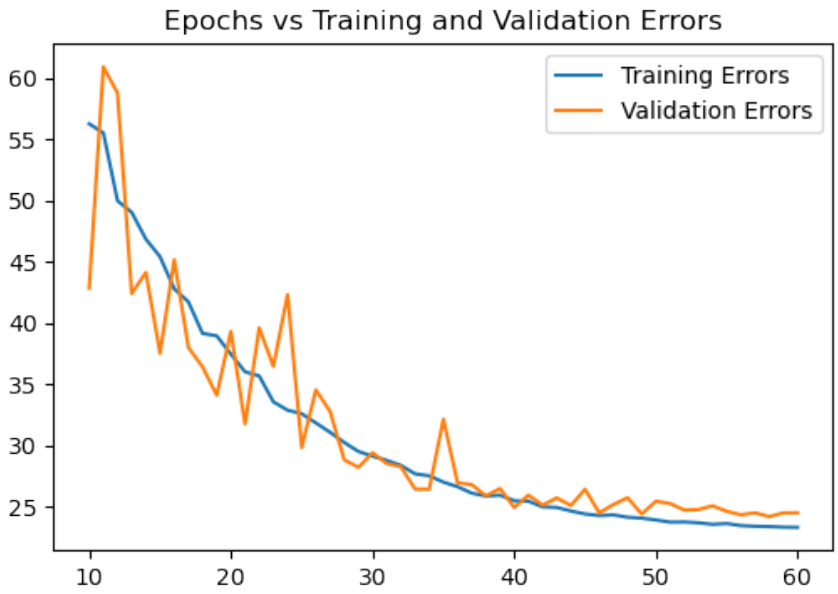

Figure 6: Training and validation error for the GCLSTM with 200 hidden units

Algorithm 4: Prediction for $I_{1,j}(50), \ldots, I_{10,j}(50)$ by model F_{model}

Input: A trained model F_{model} and steps $X(0), \ldots, X(19)$ of simulation $J = (X(0), \ldots, X(50))$

Output: An array P giving predicted values for $I_{1,J}(50), \ldots, I_{10,J}(50)$

function predictFuture($f_{\text{model}}, X(0), \cdots, X(50)$)
    Initialize array $H = [X(0), \ldots, X(19)]$;
    for $i = 20, \ldots, 50$ **do**
        Generate $\hat{X}(i) = F_{\text{model}}(H)$;
        Remove $H[0]$ from H and add $\hat{X}(i)$ to H;
    Sample $I_{1,J}(50), \ldots, I_{10,J}(50)$ from $H[-1]$;
    return $I_{1,J}(50), \ldots, I_{10,J}(50)$

4.3.2 Model Summaries and Analysis

Each of the 12 models used are summarized in Table 1; for each of the discussed architectures, we tested models with 50, 100, and 200 hidden units. The table also gives the evaluation metric of each model (with standard deviations).

Table 1: Model Summaries and Performance

Architecture	Hidden Units	Trainable Parameters	L_{eval}
Baseline	-	-	223.1 ± 27.4
RNN	50	6,590	75.3 ± 17.4
RNN	100	18,140	49.5 ± 14.2
RNN	200	56,240	79.3 ± 23.3
LSTM	50	20,240	47.8 ± 13.6
LSTM	100	60,440	52.6 ± 15.7
LSTM	200	200,840	49.2 ± 16.4
GCRNN	50	2,090	233.1 ± 68.3
GCRNN	100	4,190	115.6 ± 25.3
GCRNN	200	8,540	85.6 ± 28.5
GCLSTM	50	2,240	71.0 ± 22.9
GCLSTM	100	4,640	64.4 ± 17.5
GCLSTM	200	10,040	46.7 ± 17.4

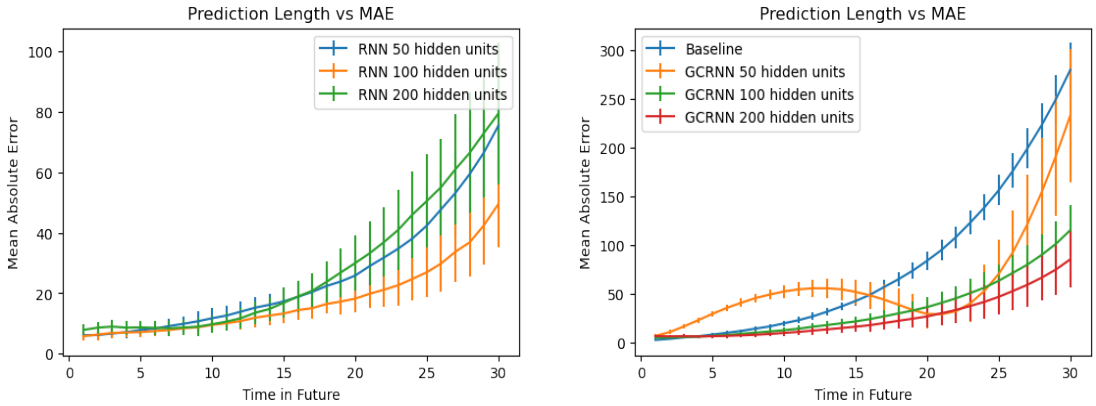

Figure 7: Comparison of RNN and GCRNN models

Figures 7 and 8 show the mean absolute errors of all tested models for different forecast lengths, across all testing simulations (note that the error at 30 future steps represents our evaluation metric). These errors were produced by inputting the first 20 time steps of data from each testing simulation into each model and using Algorithm 4 to extrapolate predictions into the further future. All models perform very well for predictions made within the first few future time steps, likely due

to the relatively smooth nature of simulations.

Interestingly, the standard RNN and LSTM models show no noticeable difference in performance for the different numbers of hidden units. This may be indicative of their inability to learn to utilize all hidden "memory" during training. We also note that, as expected, the LSTM and GCLSTM models perform better than their RNN and GCRNN counterparts, likely since their architectures permit the more long-term storage of memory.

When comparing the graph convolutional models, on the other hand, significant improvement in performance can be seen when an increased number of units are used in the architectures. This may be a sign that the graph convolutional models are better at utilizing all hidden units for prediction. We can also see the errors given by the baseline model plotted alongside the different GCRNN models in Figure 7; the baseline errors are only present in this plot as the GCRNN model with 50 hidden units was the only model with comparably large errors.

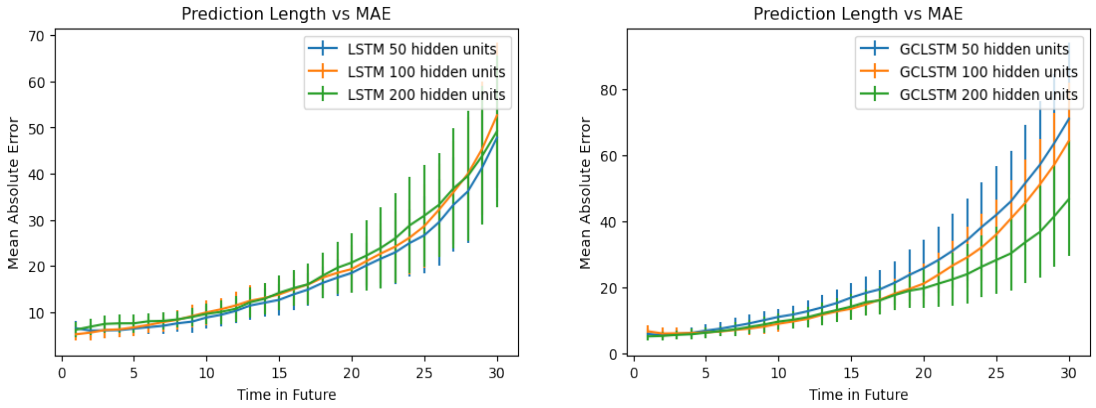

Figure 8: Comparison of LSTM and GCLSTM models

We compare the errors from different model architectures (each having 200 hidden units) to one another in Figure 9. From this plot, it is clear that, for this number of hidden units, the GCRNN's performance is similar to the RNN's, and the GCLSTM's is similar to the LSTM's. This is a very significant result, because the total number of trainable parameters in each graph convolutional model is much smaller than in the standard models; the GCRNN with 200 hidden units has just 8,540 trainable parameters, while the RNN with 200 hidden units has 56,240 trainable parameters—greater by a factor of 6.5. Similarly, the GCLSTM with 200 hidden units has only 10,040 trainable parameters, while the LSTM with 200 hidden units has 200,840 trainable parameters, greater by a large factor of 20.

Because the graph convolutional models achieve the same performance as the standard models while having far fewer parameters, they have the potential to be very useful in learning on spatial-temporal data. The smaller model size would both decrease the required training time to obtain results that match those obtained through standard RNN and LSTM models, and the models themselves would have much smaller storage requirements.

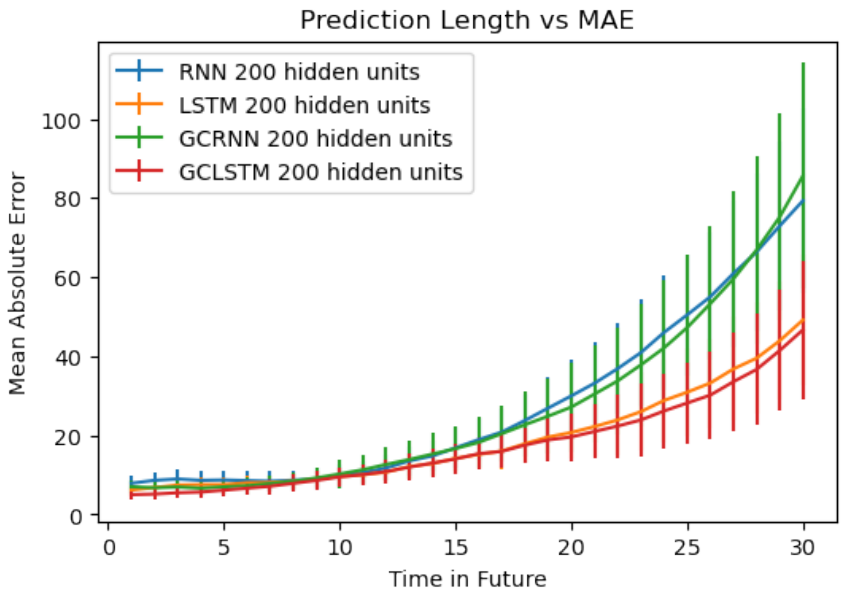

Figure 9: Comparison of different model architectures (200 hidden units)

References

[1] Joan L Aron and Ira B Schwartz. Seasonality and period-doubling bifurcations in an epidemic model. *Journal of theoretical biology*, 110(4):665–679, 1984.

[2] Yuanda Chen. Thinning algorithms for simulating point processes. *Florida State University, Tallahassee, FL*, 2016.

[3] Richard Durrett and R Durrett. *Essentials of stochastic processes*, volume 1. Springer, 1999.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[5] Jiang Guo. Backpropagation through time. *Unpubl. ms., Harbin Institute of Technology*, 40:1–6, 2013.

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[8] David G Kendall. Deterministic and stochastic epidemics in closed populations. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 4: Contributions to Biology and Problems of Health*, pages 149–165. University of California Press, 1956.

[9] Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations*, 2014.

[10] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[11] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[12] Sheldon M Ross, John J Kelly, Roger J Sullivan, William James Perry, Donald Mercer, Ruth M Davis, Thomas Dell Washburn, Earl V Sager, Joseph B Boyce, and Vincent L Bristow. *Stochastic processes*, volume 2. Wiley New York, 1996.