Understanding the Infection Dynamics of COVID-19 through a SuEIR Model and Deep Learning

Chloe You[1] **and Mateusz Faltyn**[2]

[1]University of British Columbia, Department of Statistics, Vancouver, V6T 1Z2, Canada
[2]University of British Columbia, Department of Mathematics, Vancouver, V6T 1Z2, Canada

ABSTRACT

In this work, we first examine and explain the COVID-19 SuEIR Model presented in Zou et al 2020. Then, we numerically solve the SuEIR Model via a 1) Runge-Kutta method and 2) an artificial neural network ODE solver (Chen et al 2020) to generate "ground truth data" via numerical simulation using parameters and initial conditions from the literature. Next, we generate two sets (ode45 vs NeuroDiffEq) of 10 simulations of 4 time series (S, E, I, R) by selectively perturbing the ground truth parameters. Finally, we reimplement and analyze the parameter learning method presented by Zou et al 2020 on both sets of the 10 simulations to generate R_0 values and compare them to our ground truth.

1 Introduction

With the rapid spread of a new coronavirus (SARS-CoV-2; COVID-19) from Wuhan to the vast majority of the world in 2020, researchers across disciplines paused existing projects and turned their attention to investigating some aspects of the pandemic.[1] Appropriately, many mathematical biology and adjacent research groups attempted to model and forecast the transmission of the virus through epidemic models. Two of the most widely known epidemic models for their simplicity and utility are the Susceptible-Infected-Resistant (SIR) and Susceptible-Exposed-Infected-Removed (SEIR) models. Both of these models are systems of ordinary differential equations (ODEs) where the parameters characterize rates relevant to the pandemic's spread.[2,3]

The successes of machine learning techniques over the past decade have inspired researchers to attempt to blend this modeling method with existing paradigms such as epidemic models.[4] In their 2020 preprint, Zou et al[5] combine a novel epidemic model (SuEIR) with machine learning to forecast the spread of COVID-19 within the United States. The goals of this work are to 1) examine COVID-19 SuEIR Model presented in Zou et al, 2) run two sets of simulations with a traditional and deep learning approach, and 3) estimate transmission parameters using a curve-fitting method to R_0 values and compare them against ground truth.

2 COVID-19 SuEIR Model

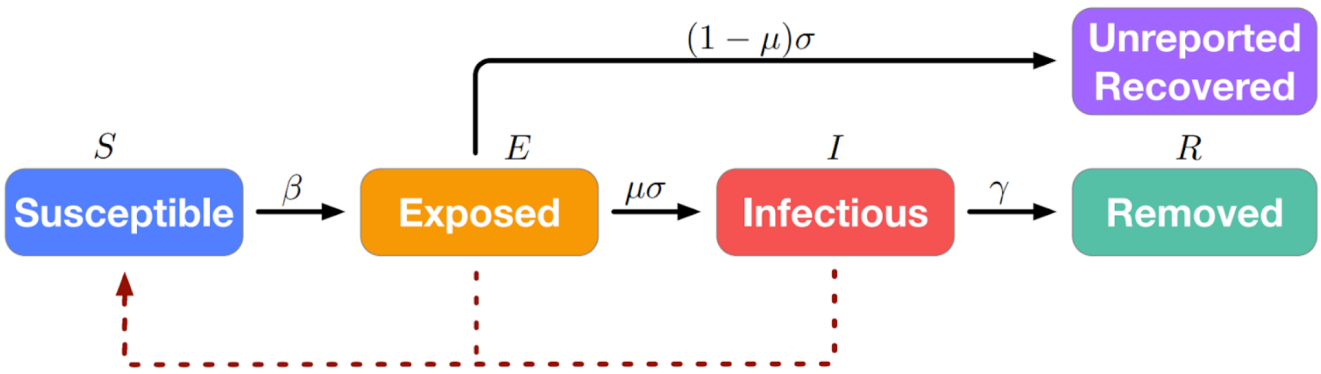

Figure 1. COVID-19 SuEIR Model; Solid lines represent the transitions of individuals and dashed lines represent the routes of infection.[5]

Figure 1 displays the SuEIR compartmental model proposed by Zou et al .[5]. In this model, S, E, I, R represent the susceptible, exposed, infectious, and recovered groups, respectively. N is the total population $N = S+E+I+R$. β is the contact rate between the susceptible and infected groups (E and I). σ is the transition rate from E to I or u (unreported recovered). μ is the discovery rate of infected cases. γ is the transition rate from I to R. $\beta, \sigma, \mu, \gamma$ are all between 0 and 1. From this model, we arrive at the following system of ODEs:

$$\frac{\partial S_t}{\partial t} = -\frac{\beta(I_t + E_t)S_t}{N}, \tag{1}$$

$$\frac{\partial E_t}{\partial t} = \frac{\beta(I_t + E_t)S_t}{N} - \sigma E_t, \tag{2}$$

$$\frac{\partial I_t}{\partial t} = \mu\sigma E_t - \gamma I_t, \tag{3}$$

$$\frac{\partial R_t}{\partial t} = \gamma I_t. \tag{4}$$

Furthermore, the basic reproduction number R_0 based on the above ODE system can be computed using the next generation matrix[6]. We summarize the calculation proposed in Zou et al.[5] as follows. First, let $\mathbf{x} = (x_1, x_2, x_3, x_4)^T$ denote the number of infected individuals in compartment S, E, I, R. Then the ODE system can be expressed as $\frac{d\mathbf{x}}{dt} = F(\mathbf{x}) - V(\mathbf{x})$, where

$$F(\mathbf{x}) = \begin{bmatrix} 0 \\ \frac{\beta(x_2+x_3)x_1}{N} \\ 0 \\ 0 \end{bmatrix}, V(\mathbf{x}) = \begin{bmatrix} \frac{\beta(x_2+x_3)x_1}{N} \\ \sigma x_2 \\ \gamma x_3 - \mu\sigma x_2 \\ -\gamma x_3 \end{bmatrix}.$$

Let $\mathbf{F}$ and $\mathbf{V}$ be the partial Jacobian matrices of functions $F(\mathbf{x})$ and $V(\mathbf{x})$ with respect to x_i, $i = 1, 2, 3, 4$.

$$\mathbf{F} = \begin{bmatrix} \frac{\partial F_2(\mathbf{x}^*)}{\partial x_2} & \frac{\partial F_2(\mathbf{x}^*)}{\partial x_3} \\ \frac{\partial F_3(\mathbf{x}^*)}{\partial x_2} & \frac{\partial F_3(\mathbf{x}^*)}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \beta & \beta \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{V} = \begin{bmatrix} \frac{\partial V_2(\mathbf{x}^*)}{\partial x_2} & \frac{\partial V_2(\mathbf{x}^*)}{\partial x_3} \\ \frac{\partial V_3(\mathbf{x}^*)}{\partial x_2} & \frac{\partial V_3(\mathbf{x}^*)}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \sigma & 0 \\ -\mu\sigma & \gamma \end{bmatrix}.$$

Then the next-generation matrix can be computed as $\mathbf{G} = \mathbf{F}\mathbf{V}^{-1} = \begin{bmatrix} \frac{\beta}{\sigma} + \frac{\beta\mu}{\gamma} & \frac{\beta}{\gamma} \\ 0 & 0 \end{bmatrix}$, and R_0 is given by the largest eigenvalue of next generation matrix $\mathbf{G}$:

$$\mathcal{R}_0 = \frac{\beta}{\sigma} + \frac{\beta\mu}{\gamma}.$$

3 Fundamentals of Neural Networks

3.1 Preliminaries

To begin, a neural network is a network made up of *neurons* - a cell in a graph that receives an input (usually a real number) and generates an output (usually another real number). When an input enters a neuron, it is scaled (i.e., multiplied) by a *weight* - a scalar that determines the significance of the input. *Bias* is an additional scalar that is multiplied together with the weight and the input. Once all of the linear components (i.e., input, weight, and bias) are multiplied together, a non-linear function called the *activation function* is applied to the input. The specific form of the activation function is determined by the neural network architecture. Once the activation function is applied, the transformed input is sent out of the neuron as an output. The objective of a neural network is to approximate an unknown function that maps to a specific *objective function* - the specific modeling task or goal.

Within a neural network, there are three types of layers: the input layer, the hidden layer, and the output layer. First, the *input layer* is the layer of neurons that initially receives input from the data. Next, the *hidden layers* are layers of neurons which perform the "learning" or processing of the data. The notion of *depth* refers to the number of hidden layers in a neural network (e.g., a deep neural network has more than one hidden layer). Finally, the *output layer* is the last layer of neurons that generates an output or prediction based on the objective function of the neural network. It is important to note that we are only able to easily examine the input and output layers of a neural network.

3.2 Definitions

Definition 3.1. (Feedforward Neural Network)[7] A *feedforward neural network* is a directed acyclic graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ over its edges. Every node in the graph corresponds to a *neuron* and is modeled by a scalar function $\sigma : \mathbb{R} \to \mathbb{R}$ called the *activation function*. Every edge connects the output of some neuron to the input of another. The input of a neuron is the weighted sum of all of the outputs of all of the neurons connected to it. Unless stated otherwise, the use of the term *neural network* refers to a feedforward neural network.

Definition 3.2. (Neural Network Layers)[7] A neural network is organized in *layers*. Each layer consists of the union of nonempty and disjoint subsets of nodes, $V = \cup_{t=0}^{T} V_t$, such that every edge in E connects some node in V_{t-1} to some node in V_t, for some $t \in [T]$ where T denotes the *depth* or number of layers in the network (excluding V_0). The size of the network is $|V|$ while the width of the network is $\max_t |V_t|$. The layers of a neural network can be grouped into three sections: the *input layer* V_0 with $n+1$ neurons where n is the dimensionality of the input space, *hidden layers* $V_1, \ldots, V_{T-1}$ where computation occurs, and the output layer V_T.

Definition 3.3. (Neural Network Computation)[7] Suppose we have a neural network with $n+1$ neurons in V_0. For every $i \in [n]$, the output of neuron i in V_0 is simply x_i. The last neuron in V_0 is the constant neuron, which always outputs 1. We denote by $v_{t,i}$ the ith neuron of the tth layer and by $o_{t,i}(x)$ the output of $v_{t,i}$ when the network is fed with the input vector x. Therefore, for $i \in [n]$ we have $o_{0,i}(x) = x_i$ and for $i = n+1$ we have $o_{0,i}(x) = 1$. We now proceed with the calculation in a layer by layer manner. Suppose we have calculated the outputs of the neurons at layer t. Then, we can calculate the outputs of the neurons at layer $t+1$ as follows. Fix some $v_{t+1,j} \in V_{t+1}$. Let $a_{t+1,j}(x)$ denote the input to $v_{t+1,j}$ when the network is fed with the input vector x. Then,

$$a_{t+1,j}(x) = \sum_{r:(v_{t,r}, v_{t+1,j}) \in E} w((v_{t,r}, v_{t+1,j}))o_{t,r}(x),$$

and

$$o_{t+1,j}(x) = \sigma(a_{t+1,j}(x)).$$

4 Fundamentals of Deep Learning

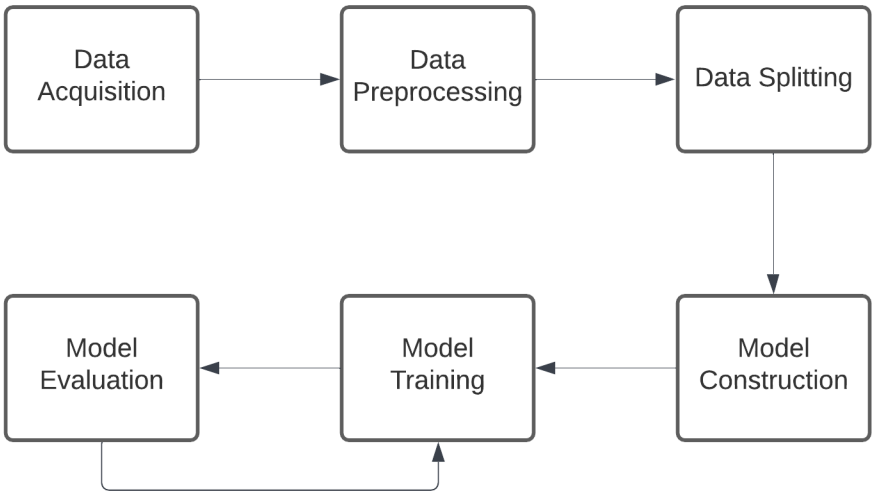

Figure 2. Deep Learning Workflow.

In this section, we provide an overview of the deep learning workflow in an academic context using the typical example of predicting whether an image is a cat or a dog. See Figure 2 for a visual summary of the workflow.

4.1 Data Acquisition

The first step in all deep learning research projects is to acquire the appropriate dataset that could be used to create a deep learning model which performs some prediction. This step is typically the most difficult because most deep learning approaches require a large amount of *labelled data* - raw data that has undergone the process of a human or machine learning model adding one or more meaningful annotations to it. In tabular datasets, data is often labelled by its column or row name. Returning to our example, a folder containing several jpegs of cats and dogs is not useful in itself for creating a deep learning model that predicts the type of animal shown in the image. Instead, the jpegs must be annotated in some way to distinguish cats and dogs. This is commonly done in image recognition tasks by moving all photos of cats into one folder labelled "Cats" and all photos of dogs into another folder labelled "Dogs".

4.2 Data Preprocessing

Once the labelled dataset is built, the data must be cleaned and scaled. *Cleaning data* refers to the removal or filling-in of outliers or missing data. *Scaling data* refers to ensuring that all of the data is homogeneous in all aspects except for the ones which matter for the deep learning task. Scaling often involves normalizing data (transforming all data to have a value between 0 and 1) and standardizing data (transforming all data to have a mean of 0 and a variance of 1). In order to scale data, all data must first be converted into real-valued numbers. Data preprocessing is extremely important in improving the performance of the deep learning model; however, it must be done with care in order to not introduce biases or errors. Returning to our example, a good start to data preprocessing would be to transform all of the images to the same dimensions (e.g., 256×256 pixels), enforce the same color model (e.g., RGB), and convert the images into real-valued numbers.

4.3 Data Splitting

Once the data has been appropriately cleaned and scaled, it is time to split the dataset into two subsets: a training set and a test set. A *training set* is the subset of a dataset that is used to train a deep learning model. *Training* refers to the concept of a neural network adjusting its weights using labelled data as inputs to minimize a specific loss function (explained in the next section). A *test set* is the subset of a dataset that is used to evaluate the deep learning model using specified metrics. $10-30\%$ of a dataset is typically used for testing, while the remainder is used for training. The *train-test split* refers to the percentage of data used for training. Returning to our example, we could select a random train-test split of 80% which means that our image recognition deep learning model would be trained on 80% of the dataset and tested on the other 20%.

4.4 Model Construction

Model construction refers to the selection and coding of the architecture that performs best for the specific deep learning task at hand. There are many different types of neural network architectures; for example, convolutional neural networks (CNNs) tend to perform very well on image recognition tasks while LSTM-RNNs tend to perform very well on time series regression tasks. It is at this point where we select a specific *loss function* - some function that determines how far off the model's guess is from the labelled data. Additionally, we select the number of hidden units (i.e., hidden layers) at this stage in the workflow as well as the metrics that will be used to evaluate our model. Once these selections are made, we begin training the model. Returning to our example, we will select and code the CNN architecture for our binary classification task (i.e., is the image a cat or a dog). We can select binary cross-entropy loss as our loss function, choose 256 hidden units, and select a confusion matrix as our primary metric for evaluation.

4.5 Model Training

Once we have selected an architecture and training criteria, it is time to start training! As a reminder, training refers to the neural network adjusting its weights using labelled data as inputs to minimize a specific loss function. Training is usually done in *batches* of data - equally-sized chunks of a dataset - to improve training efficiency. We refer to each training iteration as an *epoch*. The *learning rate* is the descent rate of the loss function. Returning to our example, our deep learning model will run through all of the training data trying to predict whether an image is a cat or a dog while attempting to minimize the loss function via gradient descent.

4.6 Model Evaluation

Finally, we can evaluate the performance of our model against a primary metric or an entire set of metrics. The choice of which metric to use is specific to the deep learning task and the architecture. Returning to our example, we can select a confusion matrix as our primary metric to evaluate our binary classification model performance as it shows us true/false positives and true/false negatives for both cats and dogs.

5 Solving the Model: Runge-Kutta vs Deep Learning

5.1 MATLAB ode45: Fourth-Order Runge-Kutta

One of the most efficient and widely-used numerical methods for solving ODEs is the Fourth-Order Runge-Kutta method.[8] The algorithm, shown in Algorithm 1, has a time complexity of $O(n)$ and a auxillary space of $O(1)$ where $n = \frac{x-x_0}{h}$. The MATLAB default ODE suite, including ode45, uses the Dormand-Prince pair method which is a member of the Runge-Kutta family of ODE explicit solvers.

Algorithm 1 Fourth-Order Runge-Kutta

Require: Differential equation $\frac{\partial y}{\partial x} = f(x, y)$, initial condition $y(x_0) = y_0$.
Calculate successively:
$k_1 = hf(x_0, y_0)$.
$k_2 = hf(x_0 + \frac{h}{2}, y_0 + \frac{k_1}{2})$.
$k_3 = hf(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2})$.
$k_4 = hf(x_0 + h, y_0 + k_3$.
Find:
$k = \frac{1}{6}(k_1 + 2k_2 + +2k_3 + k_4)$.
Return $y_1 = y_0 + k$ and $x_1 = x_0 + h$.

5.2 Python NeuroDiffEq: Deep Learning

Created by Chen et al,[9] NeuroDiffEq is a Python library built on top of PyTorch that provides tools for solving ODE and PDE systems using neural networks. As neural networks are universal function approximators, researchers have demonstrated their ability to solve complicated differential equation systems.[9] NeuroDiffEq provides two building blocks out of the box: a simple Fully-Connected Neural Network and a Residual Neural Network. Using only two simple Fully-Connected Neural Networks (one for each variable), one can accurately solve the basic Lokta-Volterra predator-prey equations[10] as demonstrated in Figure 3.

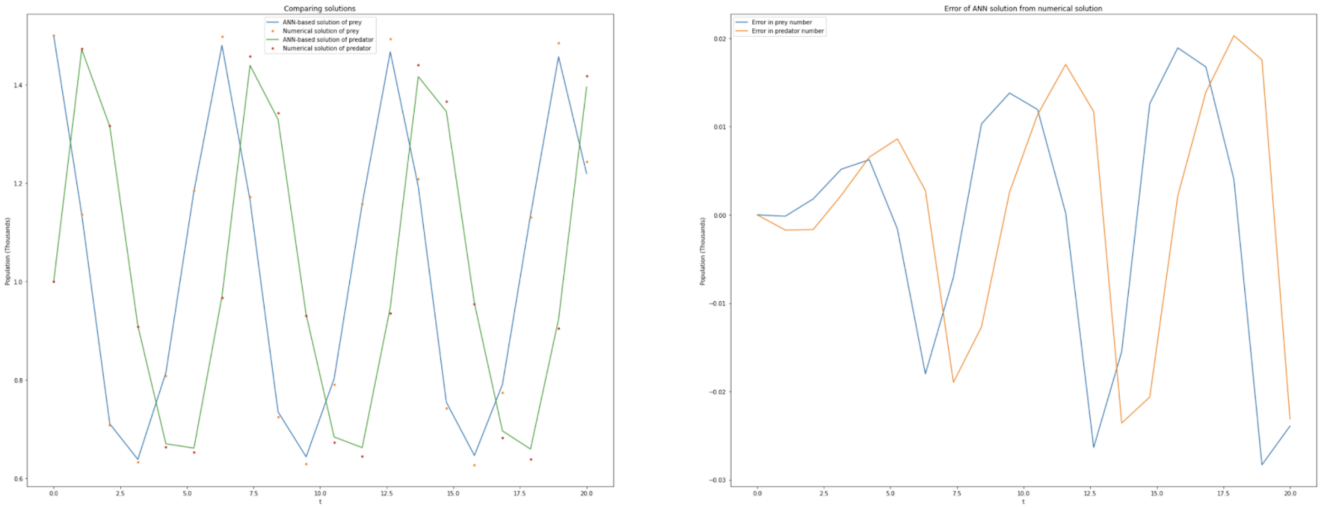

Figure 3. Solving the Lotka-Volterra Equations using NeuroDiffEq.

5.3 Simulation Setup

The three primary objectives of our simulation approach were to 1) generate a simulation to be used as "ground truth", 2) generate two sets of 10 simulations selectively perturbed from the ground truth parameters to compare the NeuroDiffEq library solvers against the ode45 solver, and 3) use these two sets of simulations as data for parameter estimation. Table 1 displays the parameters used for our "ground truth" model as well as the parameter perturbations for 10 simulations. Ground truth parameters were taken from Mwalili et al.[11] All 10 simulations are implemented twice: once in MATLAB using the ode45 solver and once in Python using the NeuroDiffEq library. All MATLAB computation was done locally on a 12th Gen Intel(R) Core(TM) i9-12900 2.40 GHz Processor with 32.0 GB RAM. All Python computation was done on Google Colab using an NVIDIA A100-SXM4-40GB Tensor Core GPU.

Simulation Number	N	S_0	E_0	I_0	R_0	β	μ	σ	γ	Explanation
0	1000	800	100	50	50	0.25	0.075	0.09	0.12	Ground truth
1	1000	800	100	50	50	0.5	0.05	0.09	0.12	Varying β and low μ
2	1000	800	100	50	50	0.4	0.05	0.09	0.12	Varying β and low μ
3	1000	800	100	50	50	0.3	0.05	0.09	0.12	Varying β and low μ
4	1000	800	100	50	50	0.2	0.05	0.09	0.12	Varying β and low μ
5	1000	800	100	50	50	0.1	0.05	0.09	0.12	Varying β and low μ
6	1000	800	100	50	50	0.5	0.1	0.09	0.12	Varying β and high μ
7	1000	800	100	50	50	0.4	0.1	0.09	0.12	Varying β and high μ
8	1000	800	100	50	50	0.3	0.1	0.09	0.12	Varying β and high μ
9	1000	800	100	50	50	0.2	0.1	0.09	0.12	Varying β and high μ
10	1000	800	100	50	50	0.1	0.1	0.09	0.12	Varying β and high μ

Table 1. Simulation Parameters (ode45 vs NeuroDiffEq).

5.4 Simulation Results

All data and code can be found in the math561-gp-SuEIR GitHub repository. All ode45 simulations had a runtime of under 0.2 seconds. All NeuroDiffEq simulations had a training time of under 1 minute 59 seconds. Figures 4 and 5 display the mean absolute error (MAE) and mean absolute percentage error (MAPE) of all four compartments for Simulations 1-10 using NeuroDiffEq versus ode45, respectively. Figures 6 and 7 show the sum of MAE and MAPE of all four compartments for Simulations 1-10 using NeuroDiffEq versus ode45, respectively. From these figures, the following general trend appears: NeuroDiffEq solvers tend to better approximate ode45 solvers as β decreases for either higher or lower μ.

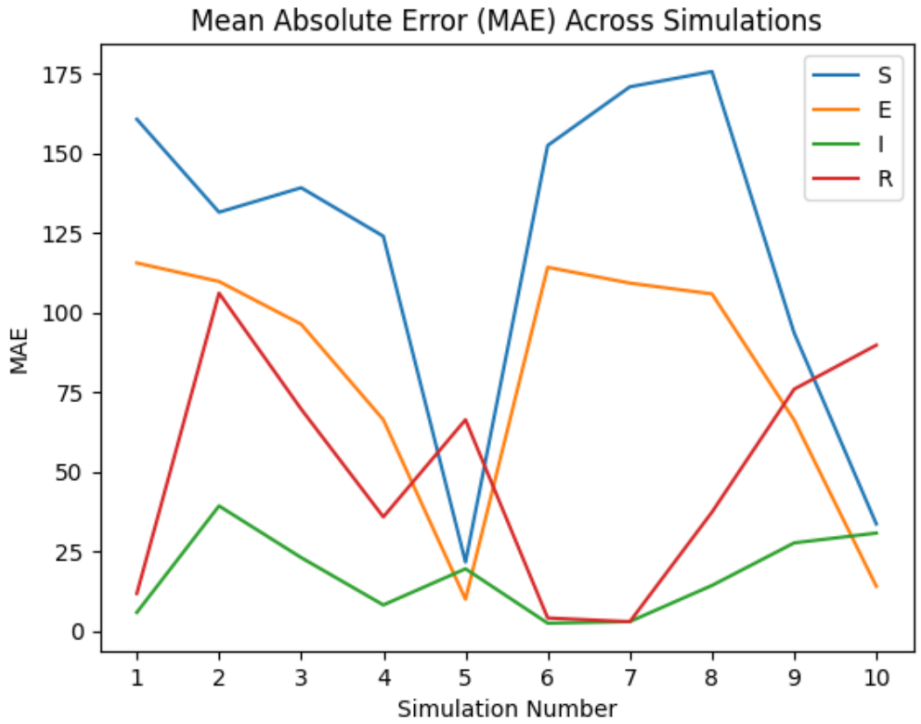

Figure 4. Mean Absolute Error (MAE) Results for All Simulations.

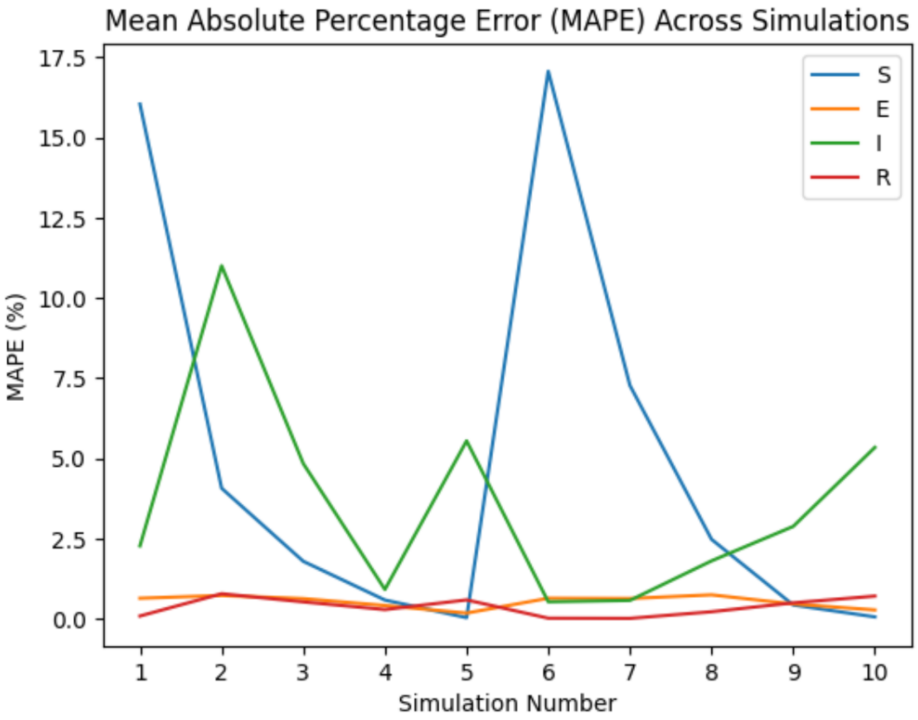

Figure 5. Mean Absolute Percentage Error (MAPE) Results for All Simulations.

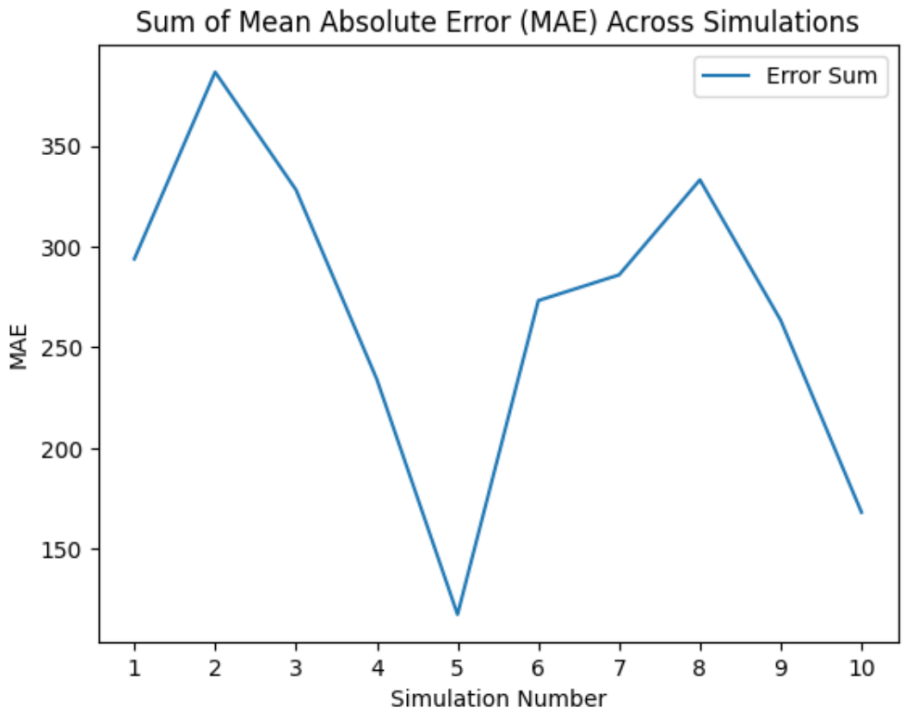

Figure 6. Sum of Mean Absolute Error (MAE) Results for All Simulations.

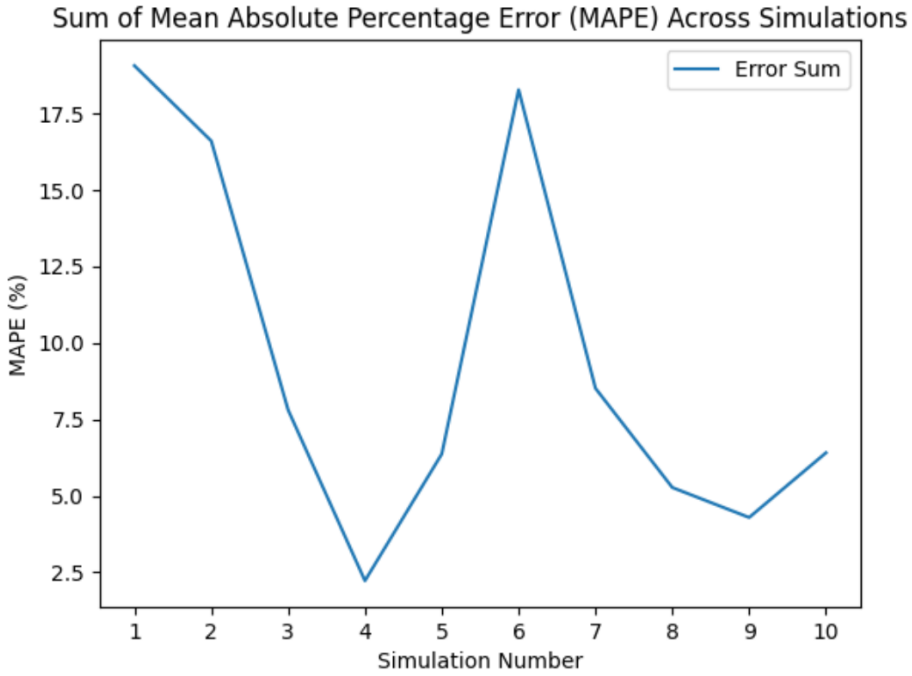

Figure 7. Sum of Mean Absolute Percentage Error (MAPE) Results for All Simulations.

We can plot the NeuroDiffEq solution versus the ode45 solution for Simulations 5 and 6 (see Figures 8 and 9, respectively) to get a better sense of the variance of the deep learning solver performance. In Simulation 5, the ode45 time series and their derivatives for all four compartments are relatively unchanging across all timepoints. However, in Simulation 6 we see that S resembles rapid exponential decay while E exhibits a sharp rise followed by slow exponential decay. More complicated functions such as S and E may be more difficult to approximate for neural networks, particularly when using simple networks such as fully-connected or residual networks.

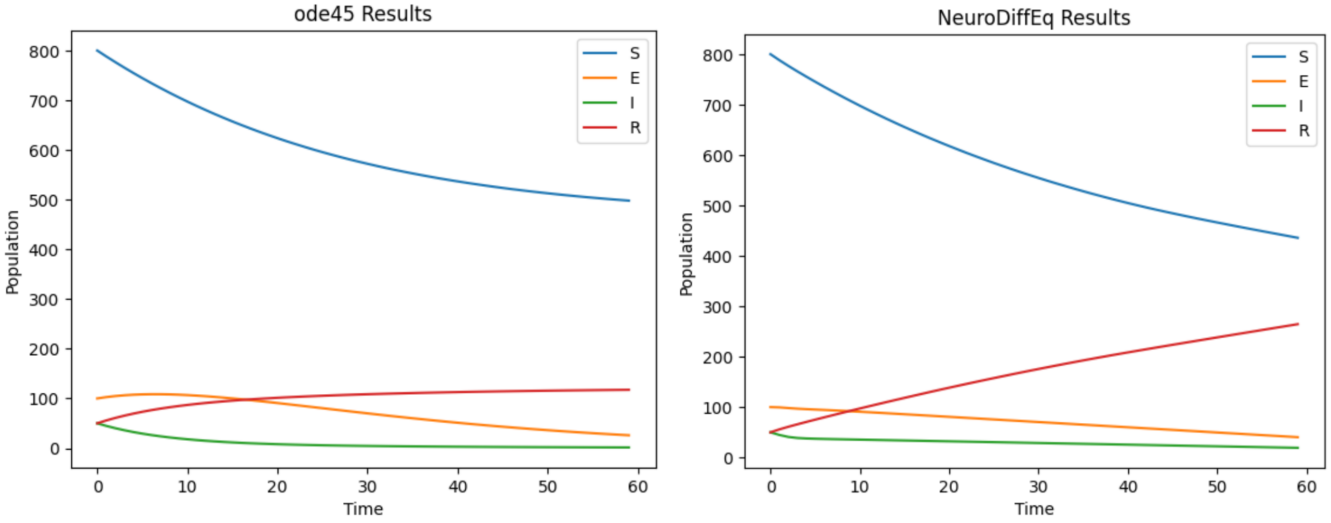

Figure 8. Simulation 5 Comparison.

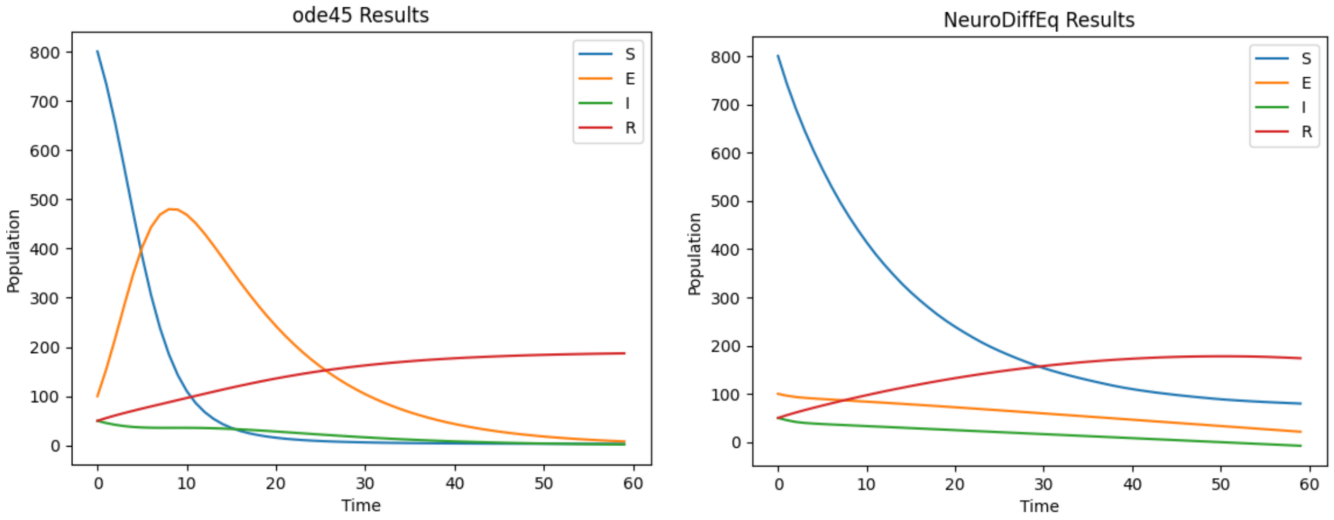

Figure 9. Simulation 6 Comparison.

6 Parameter Estimation

In this section, we reimplement and analyze the parameter learning method presented by Zou et al.[5]. We introduce the objective function that we optimize to obtain estimates of $\widehat{\boldsymbol{\theta}} = (\widehat{\beta}, \widehat{\sigma}, \widehat{\gamma}, \widehat{\mu})$ for each simulation. $\hat{R}_0$ is obtained for each simulation based on the next-generation matrix method introduced in section 2. Last but not least, we compare the R_0 estimation results and discuss the ability of the different ODE solvers to recover reasonable estimates, along with the limitations of the methods.

6.1 Objective Function and Optimization

Zou et al. proposed to learn the model parameter $\widehat{\boldsymbol{\theta}} = (\widehat{\beta}, \widehat{\sigma}, \widehat{\gamma}, \widehat{\mu})$ by minimizing the following logarithmic-type mean square error (MSE):

$$L(\boldsymbol{\theta}; \mathbf{I}, \mathbf{R}) = \frac{1}{T} \sum_{t=1}^{T} \left[\left(\log\left(\widehat{I}_t + p\right) - \log\left(I_t + p\right) \right)^2 + \left(\log\left(\widehat{R}_t + p\right) - \log\left(R_t + p\right) \right)^2 \right],$$

where $\mathbf{I} = \{I_t\}_{t=1}^{T}, \mathbf{R} = \{R_t\}_{t=1}^{T}$ with I_t and R_t denote the reported numbers of infected cases and removed cases (including both recovered cases and fatality cases) at time t (i.e., date), and p is the smoothing parameter used to ensure numerical stability. The pseudocode below shows the algorithm that solves for $\widehat{\boldsymbol{\theta}} = (\widehat{\beta}, \widehat{\sigma}, \widehat{\gamma}, \widehat{\mu})$:

Algorithm 2 Parameter Learning

Require: smoothing parameter p,
initial conditions S_0, E_0, I_0, R_0,
initial parameter estimates for $\beta, \sigma, \gamma, \mu$
"Reported" I_t, R_t values from simulations in Section 5.
Do:
Integrate the ODE system to get $\widehat{I}_t$, $\widehat{R}_t$
$(\widehat{\beta}, \widehat{\sigma}, \widehat{\gamma}, \widehat{\mu}) = \text{argmin}_{\beta,\sigma,\gamma,\mu \in [0,1]} L(\boldsymbol{\theta}; \mathbf{I}, \mathbf{R})$
return $(\widehat{\beta}, \widehat{\sigma}, \widehat{\gamma}, \widehat{\mu})$

We use the standard gradient-based optimizer mentioned in the paper to solve for the parameters. Namely, we will use L-BFGS-B, which is a second-order algorithm for bound-constrained optimization - an extension of the BFGS algorithm[12]. It is a local search algorithm, intended for convex optimization problems with a single optimum. A limitation of this, which is not mentioned in the paper, is that the objective function is likely nonconvex, therefore it is not guaranteed that the optimizer is able to converge to an optimal solution.

6.2 Parameter Estimation Results

We use two different settings of initial parameter estimates for $\beta, \sigma, \gamma, \mu$. We use a more informed initial parameter (0.15, 0.075, 0.05, 0.1) and a less informed initial parameter (0.5, 0.5, 0.5, 0.05) for the L-BFGS-B optimizer. Furthermore, by assuming

that it takes around 3 to 20 days to recover, we bound the range for γ in the optimization as (0.05, 0.3). We summarize the estimated values for R_0 in the following tables, along with the difference between the estimates and the simulation specific R_0, and the difference between the estimates and the ground truth R_0. The ground truth R_0 from Simulation 0 is 2.9 based on the calculation $\frac{\beta}{\sigma} + \frac{\beta\mu}{\gamma}$.

Table 2. Estimated R_0 and bias from an informed initial parameter (0.15, 0.075, 0.05, 0.1)

	Neural ODE Solver			Traditional ODE Solver		
Simulation	Estimated R_0	Δ simulation specific R_0	Δ ground truth	Estimated R_0	Δ simulation specific R_0	Δ ground truth
1	3.24	-2.52	0.3	2.43	-3.334	-0.5
2	2.28	-2.33	-0.7	4.59	-0.021	1.7
3	1.59	-1.87	-1.3	3.46	0.002	0.5
4	2.12	-0.19	-0.8	2.31	0.004	-0.6
5	1.48	0.33	-1.5	1.15	-0.003	-1.8
6	2.85	-3.12	-0.1	2.51	-3.462	-0.4
7	2.89	-1.89	0	4.76	-0.018	1.8
8	1.6	-1.98	-1.3	3.58	-0.003	0.6
9	1.99	-0.4	-0.9	2.39	0.001	-0.5
10	1.91	0.72	-1	1.19	-0.004	-1.7

Table 3. Estimated R_0 and bias from a non-informed initial parameter (0.5, 0.5, 0.5, 0.05)

	Neural ODE Solver			Traditional ODE Solver		
Simulation	Estimated R_0	Δ simulation specific R_0	Δ ground truth	Estimated R_0	Δ simulation specific R_0	Δ ground truth
1	3.24	-2.52	0.3	2.43	-3.33	-0.5
2	2.28	-2.33	-0.7	4.59	-0.02	1.7
3	1.59	-1.87	-1.3	3.46	0	0.5
4	2.12	-0.19	-0.8	2.31	0	-0.6
5	0.77	-0.38	-2.2	1.16	0.01	-1.8
6	2.85	-3.12	-0.1	2.51	-3.46	-0.4
7	2.89	-1.89	0	4.76	-0.02	1.8
8	1.6	-1.98	-1.3	3.58	0	0.6
9	1.99	-0.4	-0.9	2.39	0	-0.5
10	1.91	0.72	-1	38.12	36.93	35.2

We observed from Table 2 that R_0 estimates using 'reported' values from the traditional ODE solver gave more accurate estimates than the Neural ODE solver in terms of the simulation specific R_0. However, combined with the estimates from Table 3, we observed that Neural ODE solvers reported values that allowed the optimizer to estimate R_0 ranges that are closer to the ground truth 2.9. We can see that for Simulation 10 using the traditional ODE solver, $R_0 = 1.2$ but the estimated R_0 was unrealistically high. Taking a deeper dive into the estimated $\beta, \sigma, \gamma, \mu$, we see that the estimated parameters for was (1, 0.05, 0.026, 0.12) while the real true values were (0.1, 0.1, 0.09, 0.12). The estimated β was on the upper bound and therefore led to a large R_0. Furthermore, uninformed initial parameter estimates are more likely to lead to unrealistic estimates of R_0 since the objective function is nonconvex so the optimizer could get stuck at a local optimum. Overall, we were able to recover reasonable values of R_0 which is helpful in forecasting and simulations of disease outbreaks.

References

1. Chan, J. F.-W. *et al.* A familial cluster of pneumonia associated with the 2019 novel coronavirus indicating person-to-person transmission: A study of a family cluster. *The Lancet* **395**, 514–523, DOI: 10.1016/s0140-6736(20)30154-9 (2020).

2. Hethcote, H. W. The mathematics of infectious diseases. *SIAM Rev.* **42**, 599–653, DOI: 10.1137/s0036144500371907 (2000).

3. Kermack, W. O. & McKendrick, A. G. A contribution to the mathematical theory of epidemics. *Proc. Royal Soc. London. Ser. A, Containing Pap. a Math. Phys. Character* **115**, 700–721, DOI: 10.1098/rspa.1927.0118 (1927).

4. Sarker, I. H. Machine learning: Algorithms, real-world applications and research directions. *SN Comput. Sci.* **2**, DOI: 10.1007/s42979-021-00592-x (2021).

5. Zou, D. *et al.* Epidemic model guided machine learning for covid-19 forecasts in the united states. *medRxiv* DOI: 10.1101/2020.05.24.20111989 (2020).

6. Heffernan, J. M., Smith, R. J. & Wahl, L. M. Perspectives on the basic reproductive ratio. *J. Royal Soc. Interface* **2**, 281–293 (2005).

7. Shalev-Shwartz, S. & Ben-David, S. *Understanding Machine Learning: From Theory to Algorithms* (Cambridge University Press, 2014).

8. Dormand, J. & Prince, P. A family of embedded runge-kutta formulae. *J. Comput. Appl. Math.* **6**, 19–26, DOI: 10.1016/0771-050x(80)90013-3 (1980).

9. Chen, F. *et al.* Neurodiffeq: A python package for solving differential equations with neural networks. *J. Open Source Softw.* **5**, 1931, DOI: 10.21105/joss.01931 (2020).

10. Brauer, F. & Castillo-Chávez, C. *Mathematical models in population biology and Epidemiology* (Springer, 2012).

11. Mwalili, S., Kimathi, M., Ojiambo, V., Gathungu, D. & Mbogo, R. Seir model for covid-19 dynamics incorporating the environment and social distancing. *BMC Res. Notes* **13**, DOI: 10.1186/s13104-020-05192-1 (2020).

12. Byrd, R. H., Lu, P., Nocedal, J. & Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM J. on scientific computing* **16**, 1190–1208 (1995).