NEURAL NETWORK-BASED MODELING AND SOLVING OF EPIDEMIC SPREADS

Thesis

Ákos Soós
Mathematics BSc

Supervisor:
Imre Fekete
Assistant Professor
Department of Applied Analysis and Computational Mathematics

Eötvös Loránd University, Faculty of Science, Institute of Mathematics
Budapest, 2025

Acknowledgements

I owe my deepest gratitude to Imre Fekete, who not only shaped this thesis but was always there to help with anything I needed.

I am thankful to all of my teachers who strengthened my love for mathematics, especially Péter Barczi, without whose lessons I wouldn't be where I am today.

Last but not least, I am forever thankful to my family and friends, whom I could always count on.

Contents

Introduction

The spread of diseases has been studied for centuries but the recent pandemic has brought the topic to the forefront of public attention. The mathematical modeling of epidemics has been a key tool in understanding the spread of diseases and in developing strategies to control them. As research in machine learning and deep learning continues to rise in popularity, more and more applications are being developed in fields which previously have not been associated with these technologies.

In this thesis, we will focus on the modeling, solving and predicting of epidemic spreads using various types of neural networks.

In the first chapter, we define some key concepts of the fields of differential equations and epidemic models. Then, in the second chapter, we provide an introduction to the theory of neural networks, with particular emphasis on their relationship to differential equations and their capability to approximate the solutions of such systems. In the third chapter, we introduce different models and neural network architectures to test them on synthetic and real datasets in the fourth chapter.

Chapter 1

Differential Equations in Epidemiology

1.1 Differential Equations

Differential equations are equations that describe a connection between an unknown function and its derivatives. They are natural tools for modeling the behavior of systems in physics, biology, economics, and many other fields. We will focus on ordinary differential equations (ODEs), which are used to describe the evolution of phenomena such as epidemics over time, using what are known as compartmental models.

Let $d \in \mathbb{N}$, $I \subset \mathbb{R}$ be an open interval, $\boldsymbol{\Omega} \subset I \times \mathbb{R}^d$ the domain, $\boldsymbol{f} \in \mathcal{C}(\boldsymbol{\Omega}, \mathbb{R}^d)$.

Definition 1.1. Given initial $t_0 \in I$ and $\boldsymbol{x_0} \in \mathbb{R}^d$ values such that $(t_0, \boldsymbol{x_0}) \in \boldsymbol{\Omega}$, we can consider the following ODE with an initial condition (IC) called an *initial value problem (IVP)* for the unknown function $\boldsymbol{x} \in \mathcal{C}^1(I, \mathbb{R}^d)$:

$$\begin{cases} \boldsymbol{x}'(t) = \boldsymbol{f}\big(t, \boldsymbol{x}(t)\big), \\ \boldsymbol{x}(t_0) = \boldsymbol{x_0}. \end{cases} \tag{1.1}$$

Definition 1.2. A function $\boldsymbol{x} \in \mathcal{C}^1(I, \mathbb{R}^d)$ is called a *solution of the IVP* (1.1) if the following conditions hold:

(i) $\{(t, \boldsymbol{x}(t)) : t \in I\} \subset \boldsymbol{\Omega}$,

(ii) $\boldsymbol{x}'(t) = \boldsymbol{f}\big(t, \boldsymbol{x}(t)\big), \quad \forall t \in I$,

(iii) $t_0 \in I \quad \text{and} \quad \boldsymbol{x}(t_0) = \boldsymbol{x_0}$.

If our goal is to approximate the solution of the IVP (1.1), we should first verify that the problem has a solution and that the solution is unique. Early results show that a unique solution exists for all ICs, provided the function $\boldsymbol{f}$ satisfies certain conditions (see the details in [1] and [2]).

Definition 1.3. A function $\boldsymbol{f} : \boldsymbol{\Omega} \to \mathbb{R}^d$ is called *Lipschitz continuous in its second variable* if there exists a constant $L > 0$ such that

$$\|\boldsymbol{f}(t, \boldsymbol{x}) - \boldsymbol{f}(t, \boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|, \quad \forall (t, \boldsymbol{x}), (t, \boldsymbol{y}) \in \boldsymbol{\Omega}.$$

Theorem 1.4. (Picard-Lindelöf) *Let $\boldsymbol{f} : \boldsymbol{\Omega} \to \mathbb{R}^d$ be a continuous function that is Lipschitz continuous in its second variable. Then the IVP* (1.1) *has a unique solution in a neighborhood of t_0.*

For data generation, we will use numerical methods to solve an IVP (1.1). One of the most popular methods is the *classical four-stage Runge-Kutta method (RK4)* as it is easy to implement and provides good results for most problems (for further details see [3]).

First, we formulate the general form of one-step methods on the equidistant grid on the interval $[a, b]$ with step size h:

$$\omega_h = \{t_i = a + ih \mid i = 0, 1, \dots, n;\; h = (b-a)/n\}. \tag{1.2}$$

We consider the numerical method on this grid defined by a given $\boldsymbol{\Phi}$ function of the form

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{\Phi}(h, t_i, \boldsymbol{y}_i, \boldsymbol{y}_{i+1}); \qquad (i = 0, 1, \dots, n-1). \tag{1.3}$$

Let $h > 0$ be the step size, ω_h the equidistant grid (1.2), we aim for $\boldsymbol{y}_i \approx \boldsymbol{x}(t_i)$ for every $t_i \in \omega_h$. The *RK4 method* has the form

$$\begin{aligned}
\boldsymbol{k}_1 &= h\boldsymbol{f}(t_i, \boldsymbol{y}_i), \\
\boldsymbol{k}_2 &= h\boldsymbol{f}\left(t_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{\boldsymbol{k}_1}{2}\right), \\
\boldsymbol{k}_3 &= h\boldsymbol{f}\left(t_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{\boldsymbol{k}_2}{2}\right), \\
\boldsymbol{k}_4 &= h\boldsymbol{f}\left(t_i + h, \boldsymbol{y}_i + \boldsymbol{k}_3\right), \\
\boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + \frac{\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4}{6}, \qquad (i = 0, 1, \dots, n-1).
\end{aligned} \tag{1.4}$$

This is an *explicit* method since $\boldsymbol{y}_{i+1}$ uses only the previous $\boldsymbol{y}_i$ value. In order to analyze the quality of the numerical solution, we need to define some key notions related to the method's error.

Definition 1.5. A numerical method $\boldsymbol{\Phi}$ is called *consistent of order* $p > 0$ if its local error satisfies

$$\left|\left|\boldsymbol{x}(t_i + h) - \Big(\boldsymbol{x}(t_i) + h\boldsymbol{\Phi}\big(h, t_i, \boldsymbol{x}(t_i), \boldsymbol{x}(t_i + h)\big)\Big)\right|\right| = \mathcal{O}(h^{p+1})$$

for any $t_i \in \omega_h$. This means that the error introduced in a single step of the method decreases proportionally to h^{p+1} as $h \to 0$.

Definition 1.6. A numerical method $\boldsymbol{\Phi}$ is called *convergent of order* $p > 0$ at the fixed point $t^\star \in [a, b]$ if its global error satisfies

$$\big|\big|\boldsymbol{y}_i - \boldsymbol{x}(t^\star)\big|\big| = \mathcal{O}(h^p).$$

The numerical method $\boldsymbol{\Phi}$ is said to be convergent of order $p > 0$ if it is convergent at every point $t^\star \in [a, b]$. This ensures that the numerical solution approaches the exact solution as the step size h decreases, at a rate proportional to h^p as $h \to 0$.

We now state the convergence theorem for one-step methods, including the RK4 method (1.4), as it can be written in the form of the general method (1.3). For further details see [4].

Theorem 1.7. (Theorem 2.3.5, [4]) *Assume that the numerical method $\mathbf{\Phi}$ has the following properties:*

- *it is consistent of order p,*
- *the function $\mathbf{\Phi}$ satisfies the Lipschitz condition with respect to its third and fourth variables, i.e. there exist constants $L_3 \geq 0$ and $L_4 \geq 0$ such that for any $\mathbf{s}_1, \mathbf{s}_2, \mathbf{p}_1, \mathbf{p}_2$ and for all $t_i \in \omega_h$ and $h > 0$, the following inequality holds:*

$$\|\mathbf{\Phi}(h, t_i, \mathbf{s}_1, \mathbf{p}_1) - \mathbf{\Phi}(h, t_i, \mathbf{s}_2, \mathbf{p}_2)\| \leq L_3 \|\mathbf{s}_1 - \mathbf{s}_2\| + L_4 \|\mathbf{p}_1 - \mathbf{p}_2\| .$$

Then the numerical method $\mathbf{\Phi}$ is convergent of order p on the interval $[a, b]$.

Corollary 1.8. *The RK4 method (1.4) is consistent and convergent of order 4.*

The Theorem 1.7 provides great precision even for relatively large step sizes.

1.2 Epidemic Models

In 1927, William Ogilvy Kermack and Anderson Gray McKendrick proposed a new model for describing the spread of infectious diseases, which became the most influential and widely recognized ODE-based epidemic model, known as the SIR model [5]. The model divides the population into three compartments: susceptible (S), infectious (I), and recovered (R). It is assumed that the total population size (N) remains constant and that the disease is transmitted from infectious to susceptible individuals.

Definition 1.9. Under the *SIR model*, we understand the following system of ordinary differential equations:

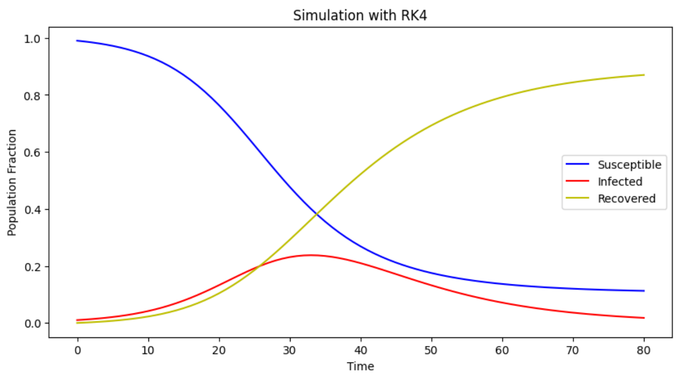
$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta}{N} SI, \\ \dfrac{dI}{dt} = \dfrac{\beta}{N} SI - \gamma I, \\ \dfrac{dR}{dt} = \gamma I, \end{cases} \tag{1.5}$$

where $\beta > 0$ is the *transmission rate*, $\gamma > 0$ the *recovery rate*, and $N \in \mathbb{Z}^+$ the *total population size.*

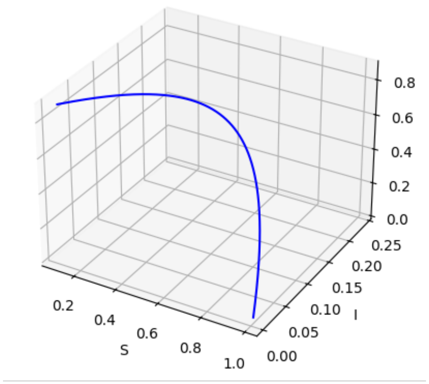
The Figures 1.1a and 1.1b show a numerical solution of the SIR model to illustrate the dynamics of the compartments.

It is easy to see that

$$\frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0,$$



(a) Plot of a solution of the SIR model.



(b) Phase space of the SIR model.

which means that the total population indeed remains constant, i.e.

$$S(t) + I(t) + R(t) = S(0) + I(0) + R(0) = N, \qquad \forall t \in \mathbb{R}^+.$$

The parameter β represents the rate at which susceptible individuals become infected upon contact with infectious individuals. Similarly, γ represents the rate at which infectious individuals recover and move to the recovered compartment.
An important quantity derived from these parameters is the basic reproduction number, R_0, which is defined as $R_0 = \frac{\beta}{\gamma}$. It represents the average number of secondary infections produced by a single infectious individual in a completely susceptible environment. If $R_0 > 1$, the infection will spread in the population, while if $R_0 < 1$, it will gradually disappear.

Although the model is nonlinear, nearly analytical solutions can be found by evaluating non-elementary integrals [6]. However, in practice, it is more common to use numerical methods (such as (1.4)) to solve the system.

There are many variations of compartmental models which use different compartments and different rules for transitions between them. One could create a new model by adding or removing compartments, or even combining multiple models to create a new one.

The SIS model is similar to the SIR model, but it does not have a recovered compartment, meaning that individuals return to the susceptible compartment after recovery.

Definition 1.10. Under the *SIS model*, we understand the following system of ODEs:

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta}{N} SI + \gamma I, \\ \dfrac{dI}{dt} = \dfrac{\beta}{N} SI - \gamma I, \end{cases}$$

where $\beta > 0$ is the *transmission rate*, $\gamma > 0$ the *recovery rate*, and $N \in \mathbb{Z}^+$ the *total population size.*

The SEIRVD model is an extension of the SIR model where a new compartment is added for exposed, deceased and vaccinated individuals. The exposed compartment represents individuals who have been infected but are not yet infectious. The deceased compartment represents individuals who have died from the disease. The vaccinated compartment represents individuals who have been vaccinated against the disease.

Definition 1.11. Under the *SEIRVD model*, we understand the following system of ODEs:

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta}{N} SI - \nu S, \\ \dfrac{dE}{dt} = \dfrac{\beta}{N} SI - \sigma E, \\ \dfrac{dI}{dt} = \sigma E - \gamma I - \delta I, \\ \dfrac{dR}{dt} = \gamma I, \\ \dfrac{dV}{dt} = \nu S, \\ \dfrac{dD}{dt} = \delta I, \end{cases}$$

where $\beta > 0$ is the *transmission rate*, $\sigma > 0$ the *incubation rate*, $\gamma > 0$ the *recovery rate*, $\nu > 0$ the *vaccination rate*, $\delta > 0$ the *disease-induced mortality rate*, and $N \in \mathbb{Z}^+$ the *total population size.*

In practice, the parameters of the models are not known exactly, and they can vary over time, making them difficult to predict. Neural networks are often used to estimate and learn these parameters from data.

Chapter 2

Neural Networks

2.1 Introduction to Neural Networks

A neural network is a parameterized function is inspired by the way biological neural networks in the human brain work. Its greatest advantage is that it can learn from data; as the algorithm adjusts its parameters, it refines its predictions over time during a process called *training*.

We will remain within the framework of supervised learning, thus our training data consists of pairs of elements from the input $\mathcal{X} \subset \mathbb{R}^n$ and the output $\mathcal{Y} \subset \mathbb{R}^m$ spaces:

$$\boldsymbol{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2),\ \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\} \qquad (\boldsymbol{x}_i \in \mathcal{X},\, \boldsymbol{y}_i \in \mathcal{Y},\, i = 1, \ldots, N).$$

Every neural network naturally can be represented as a graph, where the nodes are called neurons. These neurons are organized into *layers*, the "zeroth one" is called the input layer, the last one is the output layer, and in between them lay the hidden layers. The number of neurons in a layer is referred to as the *width* of that layer. The neurons in the input layer receive the input data, and the output of the network is in the output layer. The Figure 2.1 shows a simple neural network with 4 layers.

In the most common case, this graph is a directed acyclic graph (DAG), and the network is known as a *feedforward* neural network. We will assume this structure throughout, unless stated otherwise.

The way the input data is transformed into the output data is called the *forward pass*. Each layer can be thought of as a function that takes the output of the previous layer as its input. Then, each neuron calculates a weighted sum of the neurons of the previous layer it is connected to, with the weights corresponding to the edges of the graph. The whole layer's step can be represented as a linear transformation, i.e. a matrix multiplication. Then, a bias term is added to the weighted sum, and a nonlinear activation function is applied to the result. These weights and biases are the *learnable* parameters of the network, meaning they are adjusted during training.

In a fully connected layer, the neurons of the previous layer are connected to all neurons of the next layer.

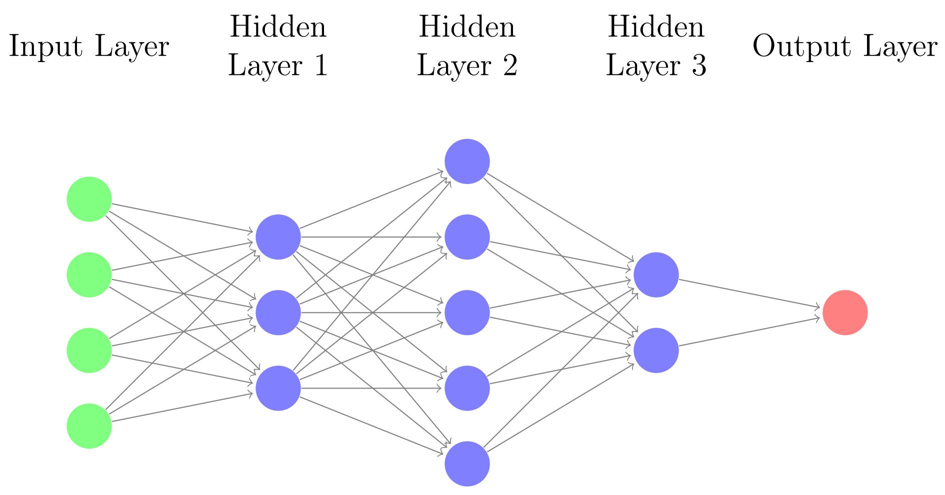


Figure 2.1: A neural network with 4 layers.

Definition 2.1. A *fully connected (FC) layer* is a function $\mathbf{FC} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ of the form

$$\mathbf{FC}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}),$$

where $\boldsymbol{x} \in \mathbb{R}^{d_1}$ is the input vector, $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is called the *weight* matrix, $\boldsymbol{b} \in \mathbb{R}^{d_2}$ is the *bias* vector, and $\boldsymbol{\phi} : \mathbb{R}^{d_2} \to \mathbb{R}^{d_2}$ is the *activation function.*

As we will see later, it is beneficial to use almost everywhere differentiable, non-linear activation functions. We list below the most common ones. A visual comparison can be found in Figure 2.2.

- The *sigmoid* activation function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

- The *hyperbolic tangent* activation function is defined as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

- The *rectified linear unit (ReLU)* activation function is defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

A simple yet still effective network is just the composition of multiple fully connected layers called a Multilayer Perceptron (MLP), which is described in detail in Section 3.2.

Now let us focus on how neural networks are trained. The goal of training is to find optimal parameters of the network, which minimize a chosen error function called the loss function.

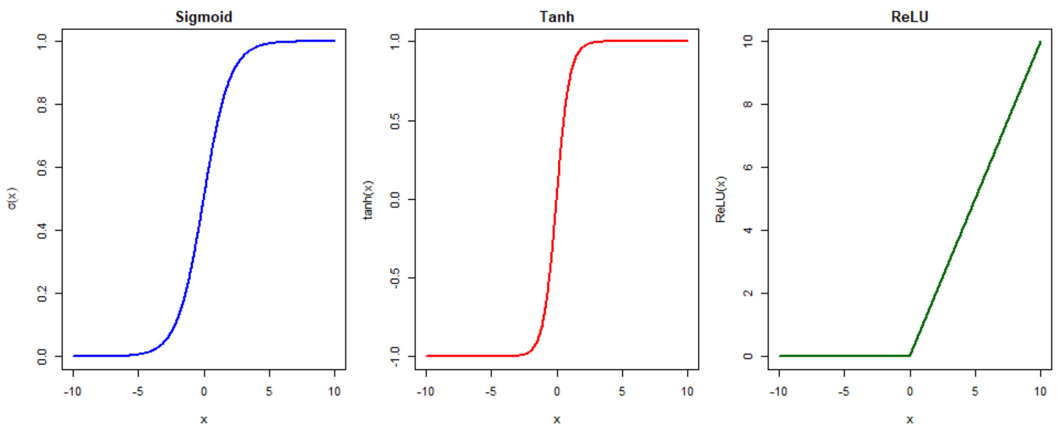


Figure 2.2: Plot of the most common activation functions.

Definition 2.2. A *loss function* $\mathcal{L} : \mathcal{Y}\times\mathcal{Y} \to \mathbb{R}$ is a function which measures the difference between the predicted and desired output of the network.

Let $\boldsymbol{y}$ be the desired output, and $\hat{\boldsymbol{y}}$ the predicted output. We list below the most common loss functions used in practice.

- The Squared Error (SE) loss function is defined as

$$\mathcal{L}_{SE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 .$$

- The Absolute Error (AE) loss function is defined as:

$$\mathcal{L}_{AE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_1 .$$

- The Cross-Entropy loss function is defined as

$$\mathcal{L}_{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{j=1}^{m} \left(y_j \log(\hat{y}_j) + (1 - y_j)\log(1 - \hat{y}_j)\right) .$$

Let $\mathcal{N}_{\boldsymbol{\theta}}$ be a neural network with parameters $\boldsymbol{\theta}$. Now, our task is to minimize this loss in the parameter space ($\boldsymbol{\theta} \in \mathbb{R}^d$ for some large $d \in \mathbb{N}$), with respect to the underlying distribution of the data $\mathcal{P}$. We are faced with the following optimization problem:

$$\min_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{P}} \left[\mathcal{L}\left(\boldsymbol{y}, \mathcal{N}_{\boldsymbol{\theta}}(\boldsymbol{x})\right)\right] = \int_{\mathcal{X}\times\mathcal{Y}} \mathcal{L}\left(\boldsymbol{y}, \mathcal{N}_{\boldsymbol{\theta}}(\boldsymbol{x})\right) \, d\mathcal{P}(\boldsymbol{x}, \boldsymbol{y}).$$

Since the true data distribution $\mathcal{P}$ is generally unknown, we cannot evaluate the expected loss directly. Instead, we approximate it using the empirical distribution defined by the training dataset. This leads to the empirical risk minimization (ERM) principle, where the expected loss is replaced by the average loss over the observed samples. Thus, we can rewrite the optimization problem as

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(\boldsymbol{y}_i, \mathcal{N}_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\right).$$

This process is usually done by iteratively updating the parameters using an algorithm called an *optimizer*. When the model has processed all the training data once, we say it has finished an *epoch*. Models are usually trained for multiple epochs, and in the case of small datasets, the optimizer steps after each completed epoch. Among the various optimization algorithms, gradient descent is the most fundamental, performing updates along the direction of the negative gradient, which corresponds to the steepest descent direction of the loss function in parameter space.

Definition 2.3. A *gradient descent* step updates the parameters of the network ($\boldsymbol{\theta}$) the following way:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}),$$

where $\eta > 0$ is the learning rate, $\boldsymbol{y}$ is the desired output, $\hat{\boldsymbol{y}}$ is the predicted output, and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the gradient of the loss function with respect to the parameters.

The learning rate η controls the size of the steps taken toward a minimum of the loss function. It is a *hyperparameter*, meaning that it is not learned during training, but rather set beforehand.

The Adam (for "Adaptive Moment Estimation") method became a popular option for optimizers after the paper [7], as it combines the ideas and advantages of previous optimizers, thus often performs better than other options in practice.

Definition 2.4. The *Adam* optimizer updates the parameters of the network as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon},$$

where $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$ are the bias-corrected first and second moment estimates, respectively, $\epsilon > 0$ is a small constant to prevent division by zero, and all operations are understood elementwise.

The first moment estimate $\boldsymbol{m}$ and the second moment estimate $\boldsymbol{v}$ are updated as:

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}), \\ \boldsymbol{v} &\leftarrow \beta_2 \boldsymbol{v} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})^2, \end{aligned}$$

where $\beta_1, \beta_2 \in (0, 1)$ are exponential decay rates. Then the bias-corrected estimates in the t-th iteration are

$$\begin{aligned} \hat{\boldsymbol{m}} &= \frac{\boldsymbol{m}}{1 - \beta_1^t}, \\ \hat{\boldsymbol{v}} &= \frac{\boldsymbol{v}}{1 - \beta_2^t}. \end{aligned}$$

The gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ is usually calculated with an algorithm called *backpropagation*, which uses the chain rule during what is called the *backward pass*.

2.2 Universal Approximation Theorem

To show the power of neural networks, researchers have developed theorems called Universal Approximation Theorems (UATs), which state that a neural network with a particular architecture can approximate any continuous function arbitrarily well. We will use this universal function approximator property to approximate the solution of the SIR model.

Cybenko was among the first ones to prove such a theorem in 1989 [8]. His theorems state that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well on the unit cube of $\mathbb{R}^n$, under mild assumptions on the activation function.

We define two important properties of activation functions to state the theorem.

Let $I_n = [0,1]^n$ denote the n-dimensional unit cube, $\mathcal{C}(I_n, \mathbb{R})$ the space of continuous real-valued functions on I_n, and $M(I_n)$ the space of finite signed Borel measures on I_n.

Definition 2.5. A function $\sigma : \mathbb{R} \to \mathbb{R}$ is called *sigmoidal* if

$$\sigma(t) \to \begin{cases} 1, & \text{as } t \to +\infty, \\ 0, & \text{as } t \to -\infty. \end{cases}$$

Definition 2.6. A function $\sigma : \mathbb{R} \to \mathbb{R}$ is called *discriminatory* if for a $\mu \in M(I_n)$

$$\int_{I_n} \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)\, d\mu(\boldsymbol{x}) = 0$$

for all $\boldsymbol{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ implies that $\mu = 0$.

Theorem 2.7. (UAT, [8]) *Suppose σ is a continuous sigmoidal function. Then, for any $f \in \mathcal{C}(I_n, \mathbb{R})$ and any $\varepsilon > 0$, there exists a sum of the form*

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i \sigma(\boldsymbol{w}_i^T \boldsymbol{x} + b_i), \tag{2.1}$$

where $\alpha_i \in \mathbb{R}$, $\boldsymbol{w}_i \in \mathbb{R}^n$, and $b_i \in \mathbb{R}$, such that

$$\sup_{\boldsymbol{x} \in I_n} |f(\boldsymbol{x}) - F(\boldsymbol{x})| < \varepsilon.$$

In other words, the sums of the form $F(\boldsymbol{x})$ are dense in $\mathcal{C}(I_n, \mathbb{R})$.

Note that this is an existence theorem, meaning that it does not guarantee that an algorithm such as backpropagation actually finds the weights and biases of the network.

For the proof, we will use two classical results from the field of functional analysis. The theorems are stated in a form suitable for our purposes rather than in the most general form possible. The proof for these can be found in any advanced textbook on the subject, such as [9].

Firstly, a consequence of the Hahn-Banach theorem is that the closure of a linear subspace can be characterized using continuous linear functionals.

Theorem 2.8. (Corollary of the Hahn-Banach Theorem [10]) *Let V be a normed linear space, U a subspace of V, and $a \in V$. Let $\overline{U}$ denote the closure of U, and V' the space of bounded/continuous linear functionals on V. Then*

$$a \in \overline{U} \quad \textit{if and only if} \quad \textit{there does not exist } L \in V' : L(u) = 0 \quad \forall u \in U \quad \textit{and} \quad L(a) \neq 0.$$

Secondly, Riesz's representation theorem connects bounded linear functionals to measure theory.

Theorem 2.9. (Riesz Representation [11] [12]) *Let $L : \mathcal{C}(I_n, \mathbb{R}) \to \mathbb{R}$ be a bounded linear functional. Then there exists a unique μ on I_n such that*

$$L(f) = \int_{I_n} f(\boldsymbol{x})\, d\mu(\boldsymbol{x}), \quad \forall f \in \mathcal{C}(I_n, \mathbb{R}).$$

Moreover, we will need the following lemma from Cybenko's original paper [8].

Lemma 2.10. *Any bounded, measurable, sigmoidal function is discriminatory.*

Now we're ready to prove the Approximation Theorem 2.7.

Proof. Let $S \subset \mathcal{C}(I_n, \mathbb{R})$ be the set of functions of the form $F(\boldsymbol{x})$ in (2.1). Naturally, S is a subset of $\mathcal{C}(I_n, \mathbb{R})$. Let $R = \overline{S}$ be the closure of S in the supremum norm. If $R = \mathcal{C}(I_n, \mathbb{R})$, then S is dense in $\mathcal{C}(I_n, \mathbb{R})$, and we are done.
Assume that $R \neq \mathcal{C}(I_n, \mathbb{R})$. Then, by the corollary of the Hahn-Banach Theorem 2.8, there exists a nonzero $L \in \mathcal{C}(I_n, \mathbb{R})'$ that vanishes on R. By the Riesz Representation Theorem 2.9, we know that

$$L(f) = \int_{I_n} f(\boldsymbol{x})\, d\mu(\boldsymbol{x}), \quad \forall f \in \mathcal{C}(I_n, \mathbb{R})$$

for some $\mu \in M(I_n)$. Now, for every $\boldsymbol{w} \in \mathbb{R}^n$, and $b \in \mathbb{R}$, the function

$$F(x) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

is in R, and therefore $L(F) = 0$. By setting $f = F$ in the representation formula, we get

$$0 = L(F) = \int_{I_n} F(\boldsymbol{x})\, d\mu(\boldsymbol{x}) = \int_{I_n} \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)\, d\mu(\boldsymbol{x}).$$

As a continuous sigmoidal function is measurable and bounded, by the Lemma 2.10, it is also discriminatory. Therefore by Definition 2.6 $\mu = 0$, which contradicts the assumption that $L \neq 0$.

□

As of today, several types of approximation theorems exist for different activations, assumptions and architectures. Cybenko's theorem is called the bounded depth or arbitrary width case, as it fixes the depth of the network to two, but has no regulations for the width of the hidden layer. Thanks to Maiorov and Pinkus [13], we even have results for the case of bounded depth and width.

We state Cybenko's and Hornik's further work on the bounded depth case, characterizing the activation function σ ([8], [15] and [16]).

Theorem 2.11. (UAT, Bounded Depth) *Let $\sigma \in \mathcal{C}(\mathbb{R}, \mathbb{R})$, and let $\sigma(\boldsymbol{x})$ denote σ applied to each coordinate of $\boldsymbol{x}$. Then σ is not polynomial if and only if for all $n, m \in \mathbb{N}$, compact $\boldsymbol{K} \subset \mathbb{R}^n$, $\boldsymbol{f} \in \mathcal{C}(\mathbb{R}^n, \mathbb{R}^m)$ and $\varepsilon > 0$, there exists a function*

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{W}_2 \cdot \sigma(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}),$$

where $k \in \mathbb{N}, \boldsymbol{W}_1 \in \mathbb{R}^{k \times n}, \boldsymbol{b} \in \mathbb{R}^k, \boldsymbol{W}_2 \in \mathbb{R}^{m \times k}$, such that

$$\sup_{\boldsymbol{x} \in \boldsymbol{K}} \|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{F}(\boldsymbol{x})\| < \varepsilon.$$

As an outlook, let us also mention a fresh and completely different approach to neural networks. Kolmogorov-Arnold Networks (KANs) are based on the Kolmogorov-Arnold representation theorem, which states that any multivariate continuous function can be represented as a finite composition of univariate functions and additions. KANs have no linear weights at all, every weight parameter is replaced by parametrized univariate functions, making them more interpretable. In theory and practice, KANs seem to outperform MLPs in many cases, as they tend to achieve better accuracy on small-scale tasks and have faster scaling properties. For further details about this new and promising alternative, see [14].

2.3 Physics Informed Neural Networks

Physics informed neural networks (PINNs) were first introduced by Raissi et al. in 2019 [17] for solving ordinary and partial differential equations. They are neural networks that incorporate physical laws and constraints into the training process. This is achieved by designing a loss function that typically includes multiple terms.

Recall the form of an ODE with an initial condition from (1.1). Suppose we have a dataset of N observed pairs of values $(t_i, \boldsymbol{x}_i)$, where t_i is the time and $\boldsymbol{x}_i$ is the observed value of the solution at that time. We would like to train the neural network $\mathcal{N}$ as an approximation of the solution $\boldsymbol{x}$. Then the PINN loss function can be defined as a weighted sum of the following terms:

- the *data loss*, which measures the difference between the predicted and observed values:
$$\mathcal{L}_{data} = \frac{1}{N} \sum_{i=1}^{N} \ell_{data}\big(\boldsymbol{x}_i, \mathcal{N}(t_i)\big);$$
- the *residual/physics/ODE loss*, which measures how well the predictions satisfy the underlying differential equations locally:
$$\mathcal{L}_{res} = \frac{1}{N} \sum_{i=1}^{N} \ell_{res}\Big(\mathcal{N}'(t_i), \boldsymbol{f}\left(t_i, \mathcal{N}(t_i)\right)\Big);$$

- the *initial loss*, which measures how well the predictions satisfy the initial conditions:

$$\mathcal{L}_{init} = \ell_{init}\big(\boldsymbol{x}_0, \mathcal{N}(t_0)\big);$$

  where ℓ_{data}, ℓ_{res} and ℓ_{init} are loss functions (e.g. SE, AE, etc.) for the data, residual and initial loss, respectively.

The total loss function is

$$\mathcal{L}_{total} = \omega_{data}\mathcal{L}_{data} + \omega_{res}\mathcal{L}_{res} + \omega_{init}\mathcal{L}_{init}, \tag{2.2}$$

where $\omega_{data}, \omega_{res}, \omega_{init} \in \mathbb{R}^+$ are the weights.

This is a powerful tool for solving differential equations, as it allows us to incorporate prior knowledge about the system into the training process. In a non-rigorous sense, the data loss "fits the model to the data", and the residual loss "tells the model how to interpolate between the data points".

Of course, an optimal loss function can change drastically from task to task, as it depends on the gathered data, the dynamics of the system and the constraints and conditions.
In the case of PDEs, it can be extended to include additional terms, such as boundary conditions or other constraints like conservation laws.

Example 2.12. A MLP with only residual loss is trained to solve the simple IVP:

$$\begin{cases} x'(t) = x(t), \\ x(0) = 1. \end{cases} \tag{2.3}$$

With only 2 hidden layers, the model easily learns the solution in 100 epochs and less than a second.

The Figures 2.3 and 2.4 show the initialized and the trained network's predictions compared to the function $x(t) = e^t$, the true solution of the IVP (2.3). The change of the value of the loss function during training is shown in Figure 2.5.

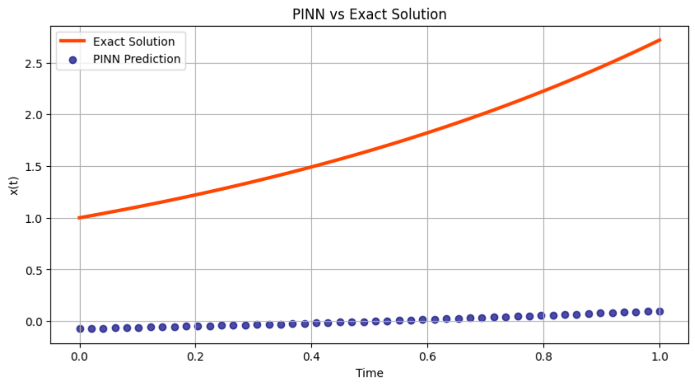


Figure 2.3: Initial predictions of the network.

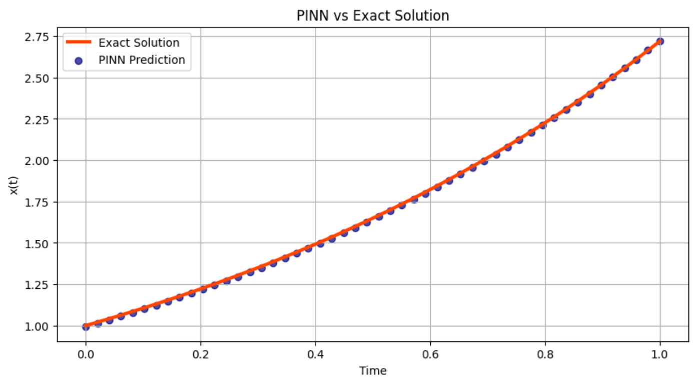


Figure 2.4: The trained network's predictions.

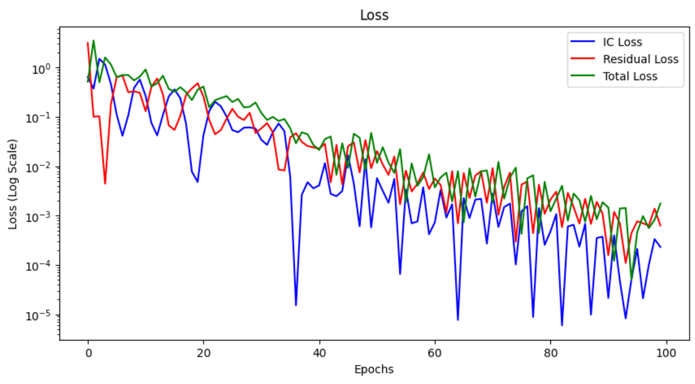


Figure 2.5: The change of loss during training.

In our case, we will use PINNs to fit the SIR model (1.5) to our data and potentially predict the progression of the epidemic over the next few days.

Chapter 3

Models

In this chapter, we define different models and neural network architectures in detail, which we will use in the next chapter to train, test and compare on different datasets.

Before we dive into neural network-based modeling, let us take a look at a more classical modeling method.

3.1 SINDy

SINDy (Sparse Identification of Nonlinear Dynamical Systems) is a classical machine learning method for identifying the governing equations of a system from data [18]. It works by constructing a library of candidate functions, and then using sparse regression techniques to select the most relevant terms to describe the dynamics of the system. Consider an autonomous system of ODEs of the form:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}(t)),$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the state vector, and $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ describes the motion of the system. Let $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ be the matrix of m measured state vectors, and let $\dot{\boldsymbol{X}} \in \mathbb{R}^{m \times n}$ be the corresponding matrix of time derivatives. If measurements for the derivatives are not available, then numerical approximations may be used instead. Our data is then given by the matrices

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^\top(t_1) \\ \mathbf{x}^\top(t_2) \\ \vdots \\ \mathbf{x}^\top(t_m) \end{bmatrix} \quad \text{and} \quad \dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^\top(t_1) \\ \dot{\mathbf{x}}^\top(t_2) \\ \vdots \\ \dot{\mathbf{x}}^\top(t_m) \end{bmatrix}.$$

We define a nonlinear feature library

$$\boldsymbol{\Theta}(\boldsymbol{X}) = \begin{bmatrix} \boldsymbol{\theta}_1(\boldsymbol{X}) & \boldsymbol{\theta}_2(\boldsymbol{X}) & \cdots & \boldsymbol{\theta}_p(\boldsymbol{X}) \end{bmatrix} \in \mathbb{R}^{m \times p}.$$

Each column of $\boldsymbol{\Theta}$ corresponds to a candidate basis function evaluated at all measured states. This usually includes polynomial terms, trigonometric functions, and other non-linear functions of the state variables.

We then seek a sparse coefficient matrix $\Xi = \begin{bmatrix} \boldsymbol{\xi}_1 & \boldsymbol{\xi}_2 & \cdots & \boldsymbol{\xi}_n \end{bmatrix} \in \mathbb{R}^{p \times n}$ such that

$$\dot{\boldsymbol{X}} \approx \boldsymbol{\Theta}(\boldsymbol{X})\Xi.$$

Since we are looking for a sparse representation, the solution is usually obtained by LASSO (Least Absolute Shrinkage and Selection Operator) regression. It is also known as regression with L_1 regularization, because it penalizes the number of non-zero coefficients in Ξ in the L_1 norm. Thus, we want to solve the following optimization problem for each column $\boldsymbol{\xi}_k$ of Ξ:

$$\boldsymbol{\xi}_k = \arg\min_{\boldsymbol{\xi}'_k} \left\| \dot{\mathbf{X}}_k - \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\xi}'_k \right\|_2^2 + \lambda \left\| \boldsymbol{\xi}'_k \right\|_1,$$

where $\lambda > 0$ is the hyperparameter of the strength of the regularization, that controls the sparsity of the solution.

Due to the non-differentiability of the L_1 term, the solution is typically computed using iterative optimization algorithms, such as subgradient methods.

3.2 MLP

As mentioned in Section 2.1, the simplest neural network architecture is the multilayer perceptron.

Definition 3.1. A *multilayer perceptron (MLP)* is a function $\mathbf{MLP} : \mathcal{X} \to \mathcal{Y}$ of the form

$$\mathbf{MLP}(\boldsymbol{x}) = \mathbf{FC}_\ell \circ \mathbf{FC}_{\ell-1} \circ \ldots \circ \mathbf{FC}_1(\boldsymbol{x}), \qquad i = 1, \ldots, \ell,$$

where ℓ is the number of layers, also called the *depth* of the network, and each $\mathbf{FC}_i$ is a fully connected layer.

MLPs are capable of achieving high accuracy on the MNIST dataset. For instance, a well-configured MLP can reach a test accuracy of 98.1% [19], meaning that it can correctly classify 98.1% of the handwritten digits in an unseen test set.

3.3 ResNet

A Residual Network (ResNet) is a type of neural network architecture that uses skip/residual connections, or shortcuts, to jump over some layers. This architecture was introduced by He et al. in 2015 [20] to address the vanishing gradient problem in deep networks, allowing for the training of very deep networks with hundreds or even thousands of layers. Residual connections were developed for computer vision, in the context of convolutional neural networks (CNNs), but they can be applied to any type of neural network, making the training process more efficient and stable.

A residual block is the building block of a ResNet, which consists of two or more layers with a skip connection that bypasses one or more layers.

Definition 3.2. A *residual block* is a function $\boldsymbol{R} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ of the form

$$\boldsymbol{R}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{P}\boldsymbol{x},$$

where $\boldsymbol{x} \in \mathbb{R}^{d_1}$ is the input vector, $\boldsymbol{f}$ is a shallow neural network, $\boldsymbol{P} \in \mathbb{R}^{d_2 \times d_1}$ is a learnable linear projection matrix.

Remark 3.3. The learnable matrix $\boldsymbol{P}$ is used to match the dimensions of the input and output, in the case of $d_1 = d_2$ it is fixed and chosen to be the identity matrix most of the time.

Around the same time, another technique was developed to improve the stability of training of deep networks, called normalization. First introduced by Ioffe and Szegedy in 2015 [21], *batch normalization (BN)* is a technique that normalizes the inputs across the batch dimension.

Following that, another normalization method called layer normalization (LN) was introduced by Ba et al. in 2016 [22], which normalizes the inputs across the feature dimension. We now define layer normalization formally, because it will be the normalization method used in our ResNet model, as it is more suitable for our task than batch normalization.

Definition 3.4. *Layer normalization* is a function $\mathbf{LN} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_1}$ defined as

$$\mathbf{LN}(\boldsymbol{x}) = \boldsymbol{\gamma} \cdot \frac{\boldsymbol{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \boldsymbol{\beta},$$

where $\boldsymbol{x} \in \mathbb{R}^{d_1}$ is the input vector,

$$\mu = \frac{1}{d_1} \sum_{i=1}^{d_1} x_i$$

is the mean of the input,

$$\sigma^2 = \frac{1}{d_1} \sum_{i=1}^{d_1} (x_i - \mu)^2$$

is the variance, $\epsilon > 0$ is a small constant added for numerical stability, and $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{d_1}$ are learnable scaling and shifting parameters. All operations are understood elementwise.

The architecture of a ResNet model later used in this thesis, visualized with the ONNX (Open Neural Network Exchange) package, is shown in Figure 3.1.

3.4 RNN

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data, where the output at each time step depends on the previous time steps, meaning that it is not a feedforward network. They have a hidden state that is updated at each time step, functioning as a memory of previous time steps. Therefore RNNs are particularly useful for tasks such as time series prediction, natural language processing, and speech recognition, where the order of the data matters.

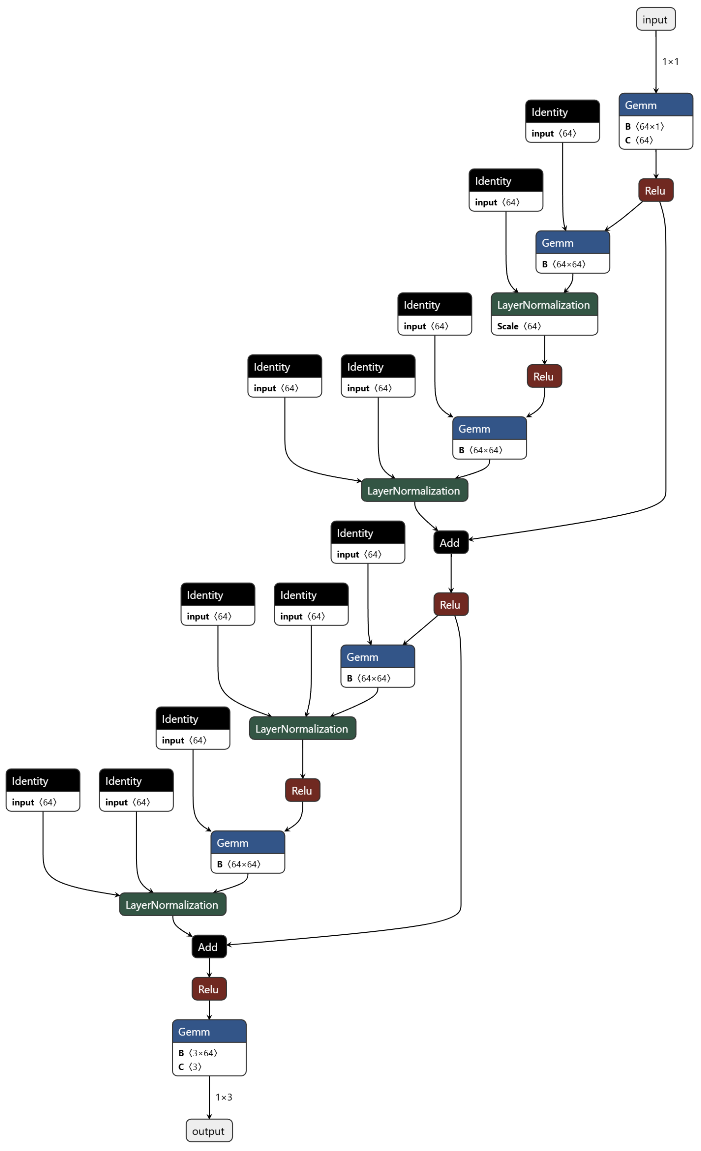


Figure 3.1: ResNet model architecture.

Definition 3.5. A *recurrent neural network* (RNN) is a function $\mathbf{RNN} : \mathcal{X} \times \mathbb{R}^k \to \mathcal{Y} \times \mathbb{R}^k$ of the form

$$\mathbf{RNN}(\boldsymbol{x}_t, \boldsymbol{h}_t) = (\boldsymbol{y}_t, \boldsymbol{h}_{t+1}),$$

where $\boldsymbol{x}_t \in \mathcal{X}$ is the input at time t; $\boldsymbol{y}_t \in \mathcal{Y}$ is the output at time t; $\boldsymbol{h}_t$ and $\boldsymbol{h}_{t+1} \in \mathbb{R}^k$ are the hidden states at time t and $t+1$, respectively; and k is the size of the hidden state.

A consequence of the definition is that if we take the appropriate computational graph of the RNN model, it is easy to notice that an RNN is not a DAG anymore, because it has a directed cycle in it, therefore backpropagation cannot be used directly. However, we can still use the backpropagation algorithm by unfolding the RNN through time, which means that we treat the RNN as a feedforward network with a number of layers equal to the number of time steps. This method, called *backpropagation through time (BPTT)*, is illustrated in Figure 3.2.

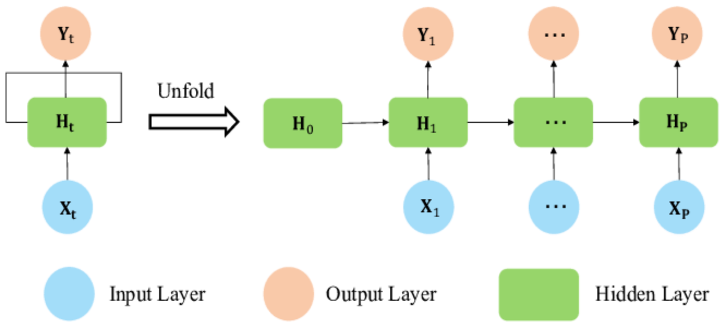


Figure 3.2: Unfolding an RNN through time. Source: [23].

The first RNN architecture which became widely used in practice was the Long Short-Term Memory (LSTM) network, which was introduced by Hochreiter and Schmidhuber in 1997 [24].

An LSTM cell consists of three main components: an input gate, a forget gate, and an output gate, and several variables, which make up these gates. The cell's update rules are defined by the following equations:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{b}_i),\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f\boldsymbol{x}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{b}_f),\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_{t-1} + \boldsymbol{b}_o),\\
\tilde{\boldsymbol{c}}_t &= \tanh(\boldsymbol{W}_c\boldsymbol{x}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1} + \boldsymbol{b}_c),\\
\boldsymbol{c}_t &= \boldsymbol{i}_t * \tilde{\boldsymbol{c}}_t + \boldsymbol{f}_t * \boldsymbol{c}_{t-1},\\
\boldsymbol{h}_t &= \boldsymbol{o}_t * \tanh(\boldsymbol{c}_t),\\
\boldsymbol{y}_t &= f(\boldsymbol{h}_t),
\end{aligned}$$

where $\boldsymbol{W}_i, \boldsymbol{W}_f, \boldsymbol{W}_o, \boldsymbol{W}_c \in \mathbb{R}^{k\times n}$ are the input weights, $\boldsymbol{U}_i, \boldsymbol{U}_f, \boldsymbol{U}_o, \boldsymbol{U}_c \in \mathbb{R}^{k\times k}$ are the recurrent weights, $\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o, \boldsymbol{b}_c \in \mathbb{R}^k$ are the biases, $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t, \boldsymbol{c}_t, \boldsymbol{h}_t \in \mathbb{R}^k$ are the input, forget, output, cell and hidden states, respectively, f is an activation function, usually tanh or ReLU and $*$ denotes elementwise multiplication.

In Figure 3.3 we can see a visualization of a LSTM cell.

GRUs (Gated Recurrent Units) are a simpler version of LSTMs, which were introduced by Cho et al. in 2014 [26]. They have only two gates, an update gate and a reset gate, and a single hidden state, resulting in fewer parameters and often faster training than LSTMs.

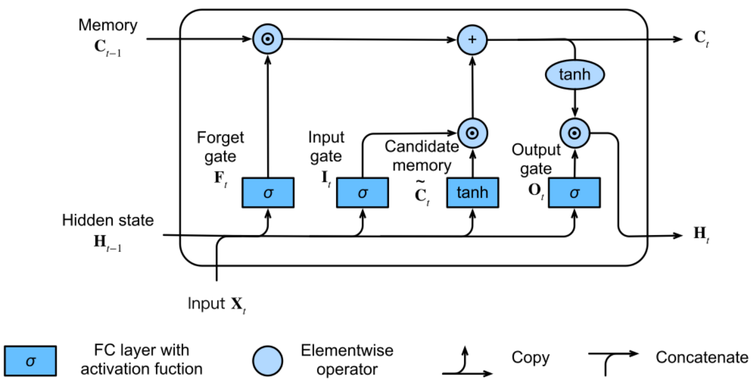


Figure 3.3: The architecture of an LSTM cell. Source: [25].

The update rules of a GRU cell are defined by the following equations:

$$\begin{aligned}
\boldsymbol{z}_t &= \sigma(\boldsymbol{W}_z\boldsymbol{x}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1} + \boldsymbol{b}_z),\\
\boldsymbol{r}_t &= \sigma(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r),\\
\tilde{\boldsymbol{h}}_t &= \tanh(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{U}_h(\boldsymbol{r}_t * \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h),\\
\boldsymbol{h}_t &= (1-\boldsymbol{z}_t) * \tilde{\boldsymbol{h}}_t + \boldsymbol{z}_t * \boldsymbol{h}_{t-1},\\
\boldsymbol{y}_t &= f(\boldsymbol{h}_t),
\end{aligned}$$

where $\boldsymbol{W}_z, \boldsymbol{W}_r, \boldsymbol{W}_h \in \mathbb{R}^{k\times n}$ are the input weights, $\boldsymbol{U}_z, \boldsymbol{U}_r, \boldsymbol{U}_h \in \mathbb{R}^{k\times k}$ are the recurrent weights, $\boldsymbol{b}_z, \boldsymbol{b}_r, \boldsymbol{b}_h \in \mathbb{R}^k$ are the biases, $\boldsymbol{z}_t, \boldsymbol{r}_t \in \mathbb{R}^k$ are the update and reset gates, respectively, $\boldsymbol{h}_t \in \mathbb{R}^k$ is the hidden state, f is an activation function, usually tanh or ReLU, and $*$ denotes elementwise multiplication.

In Figure 3.4 we can see a visualization of a GRU cell.

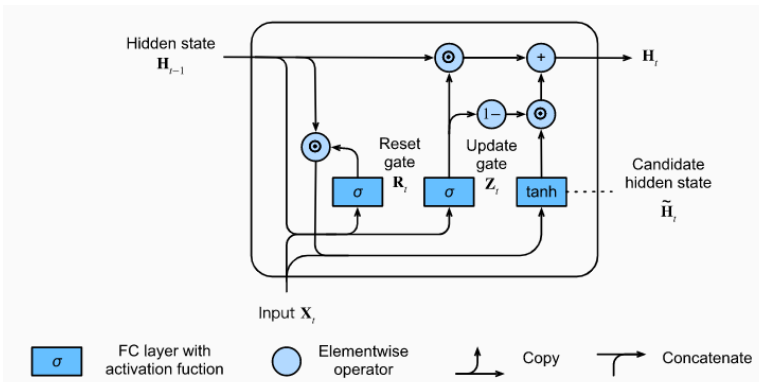


Figure 3.4: The architecture of a GRU cell. Source: [27].

Chapter 4

Results

In this chapter, we present the results of our models on two different datasets. We compare their performance based on accuracy and their ability to generalize by predicting future values.

4.1 Datasets

We use two different datasets to train and test our models. Both of them are based on the SIR model (1.5), and they contain the population fraction of the susceptible, infected and recovered individuals, respectively. One unit of time corresponds to one day.

The first one is a synthetic dataset generated via the RK4 method (1.4) solving the SIR model numerically with time step $h = 1$, and with added noise from a normal distribution with mean 0 and standard deviation 0.005. The dataset depicts a large-scale epidemic with a total of 80 days, where the first 40 days are used for training and the last 40 days for testing.

In Figure 4.1 we can see the plot of the synthetic dataset.

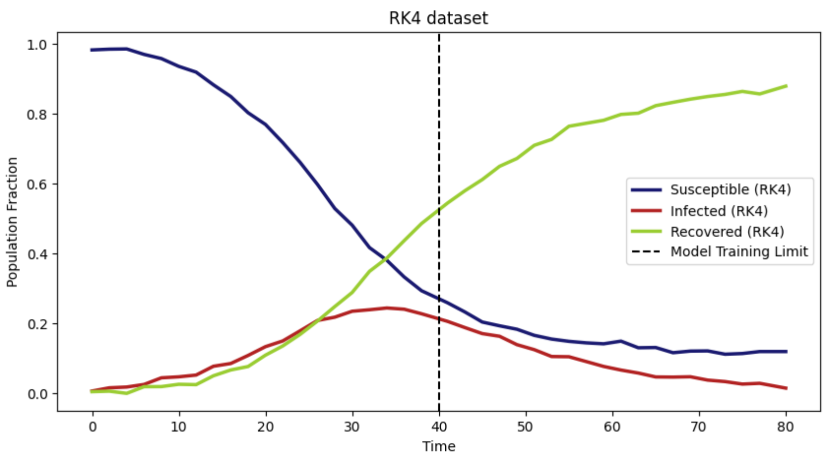


Figure 4.1: Plot of the synthetic RK4 dataset.

The second one is a real dataset from the Covid-19 pandemic, made available by the Our World in Data [28] project. It contains the daily number of confirmed cases, deaths and vaccinations in the United States over the period of one wave of the epidemic, from which the population fractions of the S, I and R compartments are calculated. The period from December 2021 to March 2022 is covered, using the first 45 days for training and the last 45 days for testing.

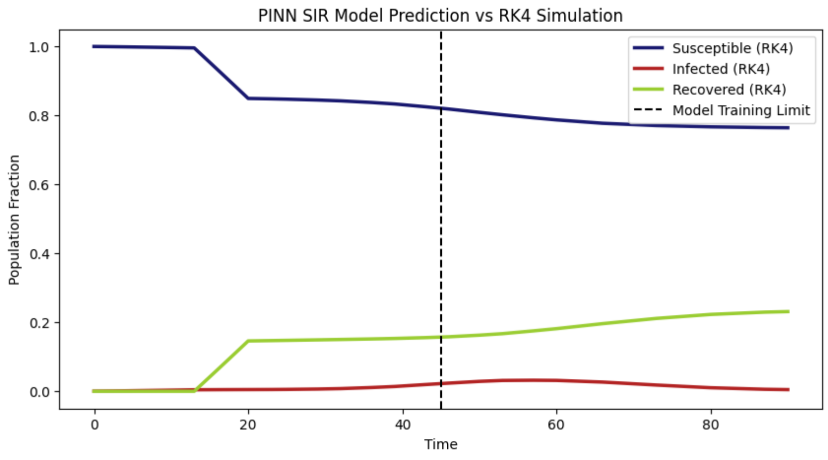


Figure 4.2: Plot of the real Covid-19 dataset.

4.2 Models

The models defined in Chapter 3 are trained and tested on both datasets with the following specifications:

- The SINDy model with a library of candidate functions $\boldsymbol{\Theta}$ of polynomial terms up to degree two, as the SIR model is a polynomial system of degree two.
- A MLP with four hidden layers, each with 64 neurons, and ReLU activation function.
- A Residual Network with two residual blocks, each with two hidden 64-neuron layers and layer normalization, as shown in Figure 3.1.
- A LSTM cell with hidden size 64 followed by a linear layer transforming to the output size.
- A GRU cell with hidden size 64 followed by a linear layer transforming to the output size.

4.3 Methodology

Each model is trained and tested on both datasets. The predictions are compared on a shorter 10 day period, and the whole 40-45 day test set as well.

The following settings and considerations were applied during the training of the neural networks:

- Every neural network is trained with the PINN-style loss function (2.2), with all of the parts weighted being Mean Squared Error (MSE) functions, e.g. Squared Error functions averaged out on the training dataset.
- The β and γ parameters of the SIR model (1.5) are not known initially, they are learnable parameters of the network as well.
- The parameters of the models are initialized randomly from different normal and uniform distributions according to PyTorch's default initialization methods, except for the ResNet model, where the weights of layers are initialized from a normal distribution with mean 0 and standard deviation 0.01, and the biases are initialized to zero.
- The hyperparameters of the models were optimized manually, the best performing ones are presented in the results.
- As the training data consists of at most 45 days, there was no need to divide the data into batches during training, therefore the number of epochs might be relatively high.
- The Adam optimizer in Definition 2.4. was used in all cases with the help of a learning rate scheduler, which reduces the learning rate gradually during training.
- Early stopping was used to prevent overfitting.

Every line of code was written in Python 3.11.12 in a Google Colaboratory Jupyter Notebook, utilizing the T4 GPU for faster computing capacity. The models were built using the PySINDy and PyTorch frameworks. The code is publicly available at GitHub [29].

4.4 Results on the synthetic dataset

Firstly, we present the results of the models on the synthetic dataset.

As we can see in Figure 4.3a, the SINDy model fitted the training data very well, but it failed to generalize to the test data, as the test predictions are not close to the true values, and the equations of the SIR model are not present.

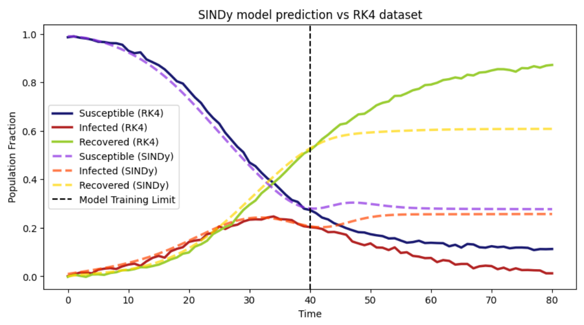
Following that, we can see the results of the MLP model in Figure 4.3b. It also fitted the training data well, but was not capable of generalizing to the test data, resulting in test predictions that were essentially a line. Interestingly, that line's slope was not equal to the derivative at the last training data point, as it might have been expected.

The ResNet model, shown in Figure 4.3c, performed even better at fitting the training data. It managed to capture the dynamics of the susceptible compartment, but not the infected and recovered compartments, falsely predicting the infected compartment to increase instead of decreasing.

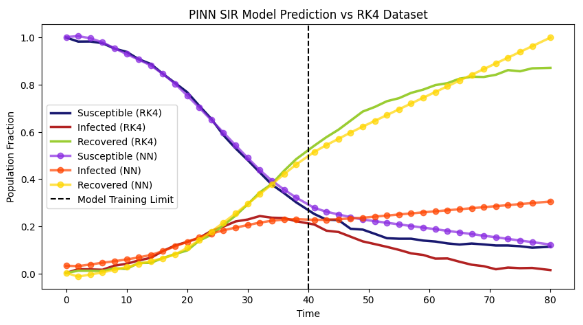
The two recurrent models, LSTM and GRU, performed similarly, as shown in Figures 4.3d and 4.3e. They fitted the first 30 days well, but then steered away from the solution, causing large test error.

The MSE values for the training, test and 10-day test sets can be found in Table 4.1.

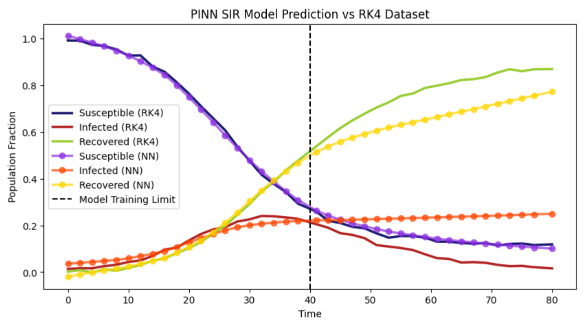
In conclusion, the ResNet model performed clearly best, but still with a significant error.



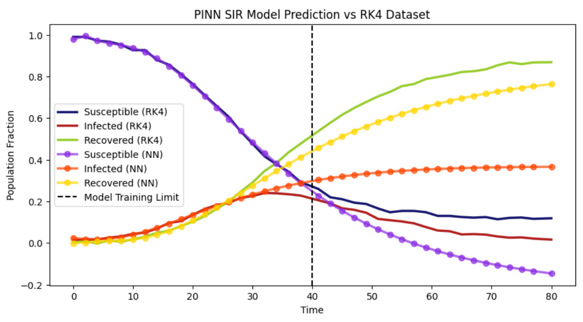
(a) SINDy



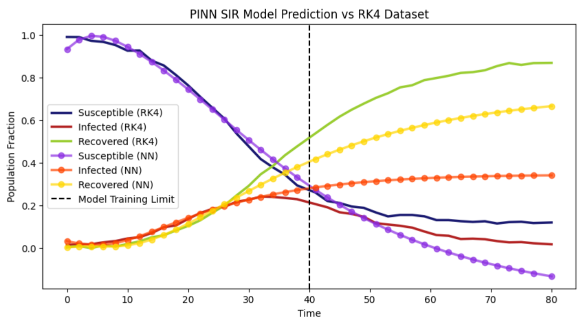
(b) MLP



(c) ResNet



(d) LSTM



(e) GRU

Figure 4.3: Results of models on the synthetic dataset.

Model	Train MSE	Test MSE	10-day Test MSE
SINDy	0.00028695	0.02764623	0.00484633
MLP	0.00020275	0.01446716	0.00283157
ResNet	0.00018742	0.01876398	0.00459437
LSTM	0.04872668	0.15723560	0.15723560
GRU	0.01134293	0.04872668	0.04872668

Table 4.1: MSE of different models on the synthetic dataset.

4.5 Results on the Covid-19 dataset

Secondly, we present the results on the real Covid-19 dataset.

The SINDy model in Figure 4.4a did not manage to fit the training data, predicting constantly zero for the infected compartment and incorrect functions for the other two.

The MLP in Figure 4.4b performed quite nicely, correctly predicting the recovered compartment's nonlinear jump, and closely predicting the infected population fraction.

Following that, the ResNet model shown in Figure 4.4c improved upon the MLP's performance, giving precise predictions for all three compartments.

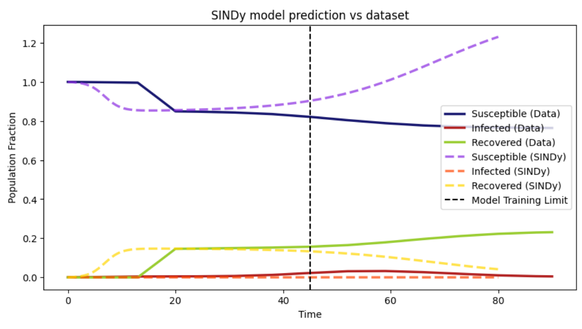
The recurrent networks in Figures 4.4d and 4.4e did not do that well in summary.

The Table 4.2 shows the MSE values for the training, test and 10-day test sets.

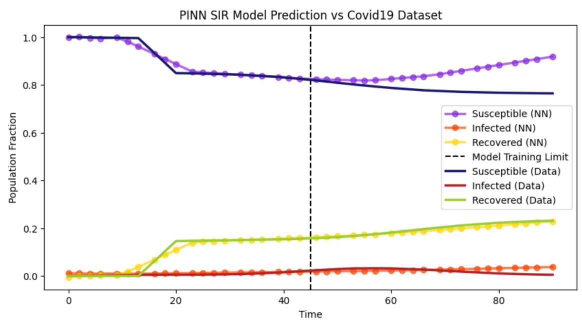
To sum up, the ResNet model has completed the task of successfully predicting the progress of the epidemic.

Model	Train MSE	Test MSE	10-day Test MSE
SINDy	0.00251505	0.03143052	0.00594847
MLP	0.00011404	0.00252494	0.00009680
ResNet	0.00011276	0.00013843	0.00010866
LSTM	0.00044755	0.00141054	0.00032160
GRU	0.00044197	0.00410682	0.00158534

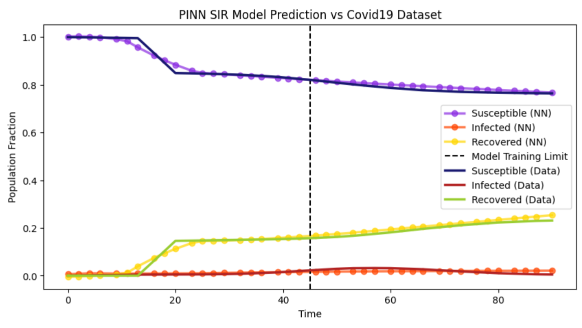
Table 4.2: MSE of different models on the Covid-19 dataset.



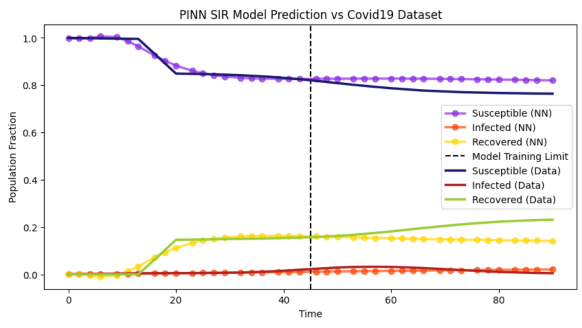
(a) SINDy



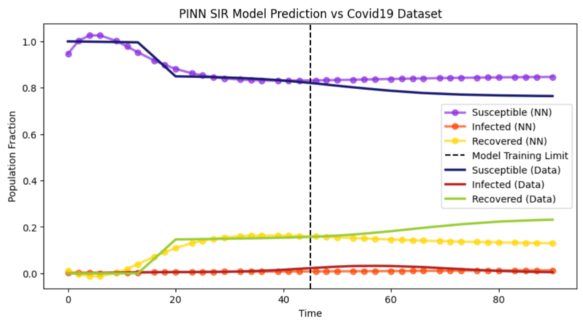
(b) MLP



(c) ResNet



(d) LSTM



(e) GRU

Figure 4.4: Results of models on the Covid-19 dataset.

Conclusions

To sum up, in this thesis, we built up and presented several models for the task of predicting an epidemic based on only the previous days' data with the tools of neural network-based modeling.

In the final chapter, we assume that the Covid-19 pandemic can be modeled using the SIR framework. While this represents a simplification, since professionals typically rely on significantly more complex compartmental models to capture the full dynamics of the epidemic, it serves as a reasonable approximation for our purposes, especially when focusing on just a single wave of the pandemic.

From the performance of the different models, we can see that the architecture of the network has a significant impact on the model's effectiveness. In the future, it would be interesting to investigate this task with more training data from different epidemics, perhaps RNNs and transformers could solve the problem more effectively than anything before.

Bibliography

[1] Picard, É.: *Sur l'application des méthodes d'approximations successives aux équations différentielles*, Journal de Mathématiques Pures et Appliquées, Vol. 4, No. 9, pp. 217–272, 1893.

[2] Lindelöf, E.: *Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires*, Bulletin des Sciences Mathématiques, Vol. 18, pp. 201–202, 1894.

[3] Hairer, E., Nørsett, S. P., Wanner, G.: *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer-Verlag, Berlin, 1993.

[4] Faragó, I.: *Numerical Methods for Ordinary Differential Equations*, Eötvös Loránd University, Budapest, 2012.

[5] Kermack, W. O., and McKendrick, A. G.: *A Contribution to the Mathematical Theory of Epidemics*, Proceedings of the Royal Society A, Vol. 115, No. 772, pp. 700–721, 1927.

[6] Harko, T., Lobo, F. S., and Mak, M. K.: *Exact Analytical Solutions of the Susceptible-Infected-Recovered (SIR) Epidemic Model and of the SIR Model with Equal Death and Birth Rates*, Applied Mathematics and Computation, Vol. 236, pp. 184–194, 2014.

[7] Kingma, D. P., Ba, J.: *Adam: A Method for Stochastic Optimization*, arXiv preprint arXiv:1412.6980, 2014.

[8] Cybenko, G.: *Approximation by Superpositions of a Sigmoidal Function*, Mathematics of Control, Signals, and Systems, Vol. 2, pp. 303–314, 1989.

[9] Rudin, W.: *Functional Analysis*, McGraw-Hill, New York, 1973.

[10] Hahn, H., and Banach, S.: *Über lineare Gleichungssysteme in linearen Räumen*, Journal für die reine und angewandte Mathematik, Vol. 157, pp. 214–229, 1927.

[11] Riesz, F.: *Sur les opérations fonctionnelles linéaires*, Comptes Rendus de l'Académie des Sciences de Paris, Vol. 149, pp. 974–977, 1909.

[12] Markov, A. A.: *On Mean Values and Exterior Measures*, C. R. (Doklady) Acad. Sci. USSR (N.S.), Vol. 55, pp. 475–478, 1947.

[13] Maiorov, V., Pinkus, A.: *Lower Bounds for Approximation by MLP Neural Networks*, Neurocomputing, Vol. 25, No. 1–3, pp. 81–91, 1999.

[14] Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., Hou, T. Y., Tegmark, M.: *KAN: Kolmogorov–Arnold Networks*, arXiv preprint arXiv:2404.19756, 2024.

[15] Hornik, K., Stinchcombe, M., White, H.: *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, Vol. 2, No. 5, pp. 359–366, 1989.

[16] Hornik, K.: *Approximation Capabilities of Multilayer Feedforward Networks*, Neural Networks, Vol. 4, No. 2, pp. 251–257, 1991.

[17] Raissi, M., Perdikaris, P., Karniadakis, G. E.: *Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations*, Journal of Computational Physics, Vol. 378, pp. 686–707, 2019.

[18] Brunton, S. L., Proctor, J. L., Kutz, J. N.: *Discovering Governing Equations from Data by Sparse Identification of Nonlinear Dynamical Systems*, Proceedings of the National Academy of Sciences, Vol. 113, No. 15, pp. 3932–3937, 2016.

[19] Zhou, H.: *MNIST – MLP*, 2022. https://ucla-biostat-203b.github.io/2022winter/slides/15-nn/mnist_mlp/mnist_mlp.html.

[20] He, K., Zhang, X., Ren, S., Sun, J.: *Deep Residual Learning for Image Recognition*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778, 2016.

[21] Ioffe, S., Szegedy, C.: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, Proceedings of the 32nd International Conference on Machine Learning (ICML), pp. 448–456, 2015.

[22] Ba, J. L., Kiros, J. R., Hinton, G. E.: *Layer Normalization*, arXiv preprint arXiv:1607.06450, 2016.

[23] Ye, J., Zhao, J., Ye, K., Xu, C.: *How to Build a Graph-Based Deep Learning Architecture in Traffic Domain: A Survey*, arXiv preprint arXiv:2005.11691, 2020.

[24] Hochreiter, S., Schmidhuber, J.: *Long Short-Term Memory*, Neural Computation, Vol. 9, No. 8, pp. 1735–1780, 1997.

[25] Zhang, A., Lipton, Z. C., Li, M., Smola, A. J.: *Long Short-Term Memory*, Dive into Deep Learning, https://d2l.ai/chapter_recurrent-modern/lstm.html.

[26] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*, arXiv preprint arXiv:1406.1078, 2014.

[27] Zhang, A., Lipton, Z. C., Li, M., Smola, A. J.: *Gated Recurrent Units*, Dive into Deep Learning, https://d2l.ai/chapter_recurrent-modern/gru.html.

[28] Ritchie, H. et al.: *COVID-19 Data*, Our World in Data. https://covid.ourworldindata.org/data/owid-covid-data.csv.

[29] Soós, Á.: *BSc Thesis Code*, GitHub Repository, 2025. https://github.com/akos5858/bsc_thesis_code.

Statement of AI usage

I, *Ákos Soós*, hereby declare that during the preparation of my thesis, I used the AI-based tools listed below to perform the following tasks:

Task	**Tool Used**	**Used For**	**Note**
LaTeX syntax assistance	GPT-4o	-	Preamble settings and figure layout
Data import	GPT-4o	Section 4.1	Loading the data into a Jupyter Notebook
Grammar and spell check	Grammarly	Complete thesis	-
Statement of AI usage table	GPT-4o	Page 34	Creating this table

Apart from the ones listed above, I did not use any other AI-based tools.