COVID-19 and Recurrent Neural Networks

Dimitrios Pilitsis

Sam Tickle

Level H/6

20 Credit Points

May 22, 2021

Contents

1 Introduction

The rapid spread of COVID-19 worldwide has warranted a response from governments to stop the spreading of the virus, while also trying to save the economy. In order to create a response plan, governments require accurate models to predict COVID-19 cases within their country. SIR models have been used extensively to model COVID-19 cases as seen in [1], [2] and [3], but have many shortfalls. Some of these shortcomings include being too simplistic and making assumptions that do not hold, such as a closed population [4]. Much less research has been done on Long Short Term Memory (LSTM) models, a particular type of Recurrent Neural Network which are much more complex and can theoretically yield better results. In addition, very little research has been done about what information is actually encoded within the weights of the model.

In this paper, we will investigate whether or not LSTMs encode information regarding COVID-19 cases within the parameters of the model, and if so, whether they can be compared with other countries and yield results that mirror the reality of COVID-19 cases worldwide. To do so, a Literature Review of Machine Learning and Neural Networks will be presented, then an explanation of the dataset and model used for the investigation. The results of the investigation will be analyzed to determine what conclusions can be made.

2 Literature Review

Before delving into the specifics of the investigation, one must understand the following ideas to understand what Long Short Term Memory (LSTM) models are and why they are a natural choice for this problem:

- Machine Learning basics
- Neural Networks
- Recurrent Neural Networks
- Long Short Term Memory (LSTM) models
- Bidirectional Neural Networks

2.1 Machine learning basics

2.1.1 Basics

Machine learning is of profound importance to this project. The neural networks used in this paper were created on the foundations of machine learning. The core idea is that machine learning is an approach to learning without having to follow explicit instructions, using various algorithms or statistical models to analyze and infer patterns within data [5]. Hardcoding constraints used to be a feasible option, but in this day and age of Big Data, where more data has been created in the past two years than in the entire history of the human race [6], it is no longer viable.

For this paper, we are interested in supervised learning problems, specifically regression tasks. In this task, the aim is to predict a numerical value given some input. More formally, the aim is to learn a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Setting $y = f(x)$, we provide an input vector x and obtain a prediction of y. In the case of this paper, x will be a vector of time, and y will be the corresponding COVID-19 cases [7].

In order to evaluate the suitability of a model, one must utilize a performance metric to determine how well the model acts with the given dataset. For regression, there are a variety of metrics to use, including but not limited to: Mean Absolute Error (MAE), Mean Squared Error (MSE) and Root Mean Squared Error (RMSE). The performance metric is of utmost importance, as the predicted value from the model will be compared with the ”true” value provided by the dataset, also known as a label or target.

An extremely important part of machine learning is the ability to perform well on new data that the model has not seen before. The ability to perform well on unseen data is known as generalization. Before beginning any learning, one must split the dataset into a ”training” and a ”testing” set. The model learns by using the data within the training set, where the performance is found by computing an error measure using a performance metric and trying to reduce it as much as possible. In essence, we are solving an optimization problem. We then test the performance of the model on unseen data i.e. the testing set, and see how well it performs by using the same performance metric. The aim is for both the training and testing error to be low [8].

This process is integral in understanding the two main challenges of machine learning: overfitting and underfitting. Overfitting is when the gap between training error and testing error is too large i.e. the model learns the training data very well but fails to generalize to unseen data, practically unusable for prediction. Underfitting is when we are not even able to obtain a low training error as the model is struggling to learn anything from the dataset [5]. Figure 1 captures visually these definitions.

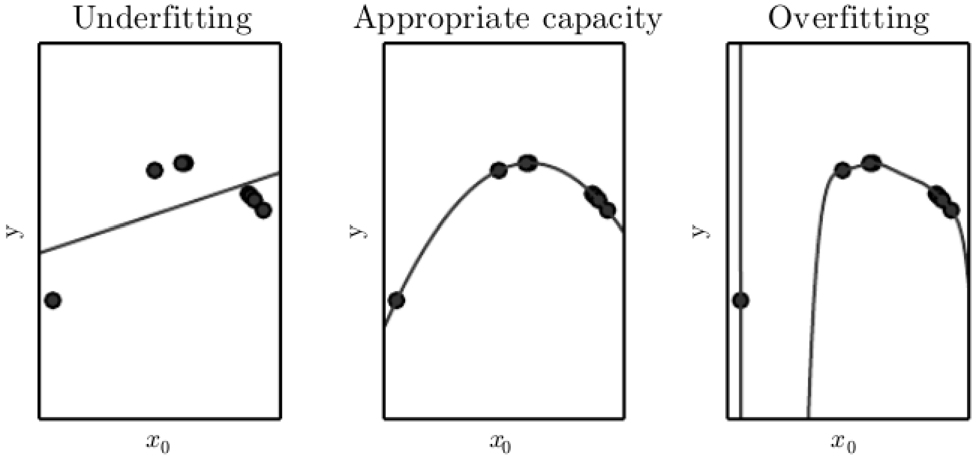

Figure 1: Diagram of a synthetic dataset with a model that underfits (left), overfits (right) and fits well (middle) the dataset [7].

There are a multitude of ways to deal with overfitting. One technique is to use regularization, a method of penalizing large weights so as to keep the model relatively simple. Providing as much data as possible also helps in allowing the model to learn better. In addition, you can control the behavior of the model by tinkering with the settings of the model, also known as hyperparameters. In this case, a validation set should be used in-between training and testing the model so that one can analyze the hyperparameters and determine if they are optimal for the issue at hand [7].

From a statistical standpoint, we are creating an estimator $\hat{\theta}$ of the true parameter θ for the model used to predict future values. For any estimator, we care about the sources of error. The two primary sources of error for our estimator are bias and variance. Bias measures the average deviation from the true value of the function, while variance is a measure of deviation from the average value obtained from the model that any particular sample of the data is likely to cause [5]. There exists an inherent trade-off between bias and variance which can be seen in Equation 1 [7]. Having a model that is too simple i.e. too few parameters, will result in a model that has high bias and low variance. To counteract this, we can increase the number of parameters of the model, in a sense ”injecting” bias into the model in order to suitably decrease the variance. Trying to find this equilibrium between bias and variance is extremely important to minimize the total error.

$$\text{MSE} = \mathbb{E}[(\hat{\theta} - \theta)^2] = \text{Bias}(\hat{\theta})^2 + \text{Var}(\hat{\theta}) \tag{1}$$

2.1.2 Stochastic Gradient Descent

Now that we have seen the basics of machine learning, it is time to understand the learning algorithm itself. This extremely powerful learning algorithm considered for this investigation is known as Stochastic Gradient Descent (SGD), and is the key to understanding how machine learning models learn. But before learning about SGD, one must learn about its predecessor Gradient Descent [9].

The core idea is to minimize a function $f(x)$ known as the cost function, i.e. find x such that $x = \text{argmin} f(x)$. The principle idea is to make changes to the weights of the model in order to find the global minimum of the cost function. Many times though it is not possible to obtain the global minimum as the model gets stuck within a local minimum [7].

But in order to arrive at a local minimum, we need to know which direction we must go towards. Let $\nabla_x f(x)$ be the gradient of f containing all the partial derivatives of $f(x)$. The directional derivative in direction u, where u is a unit vector, is the slope of function f pointing in the direction of u [7]. Combining the two, we can create Equation 2 so that we can minimize the equation to find the direction in which f decreases the quickest.

$$\min_{u, u^T u = 1} u^T \nabla_x f(x) = \min_{u, u^T u = 1} \|u\|_2 \|\nabla_x f(x)\|_2 \cos\theta \tag{2}$$

where θ is the angle in-between u and the gradient. By substituting $\|u\|_2 = 1$ and ignoring factors not related to u, we obtain $\min_u \cos\theta$. This expression is minimized when u is pointing in the opposite direction of the gradient [7]. This basic technique is known as Gradient Descent [9]. Hence, we can decrease our function f optimally by moving in the direction opposite of the gradient. So, by taking this into consideration, the new point after calculating the derivative is Equation 3.

$$x' = x - \alpha \nabla_x f(x) \tag{3}$$

where α is the learning rate. The learning rate allows us to control how fast we move in-between each step, in essence, controlling the size of each step. Gradient descent converges when every single element of the gradient is equal to zero or practically close to zero [8].

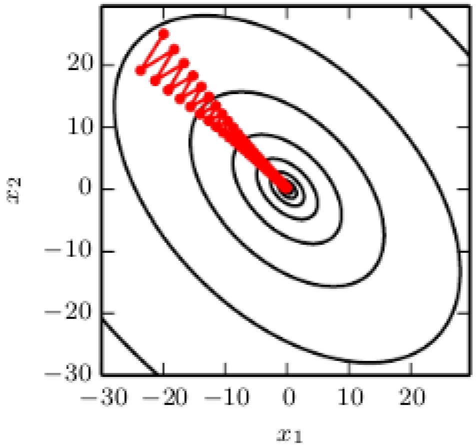

Figure 2: Diagram of gradient descent applied on a quadratic function [7].

One major issue with machine learning algorithms is the amount of data required for these models to learn. In order for a model to generalize, very large amounts of data are needed for training. This not only makes it harder as you need to obtain more data, but you need more computation to complete the task. Another issue is that gradient descent doesn't take into consideration the curvature of the curve, as seen in Figure 2 [5]. Stochastic Gradient Descent attempts to solve this issue. A common cost function for machine learning, say $J(\cdot)$, is usually of the form of a summation which takes all training examples and calculates their individual loss, as seen in Equation 4, where $L(\cdot)$ is the per observation loss, $\boldsymbol{x}^{(i)}$ is the ith input vector, $y^{(i)}$ is the true value of the ith observation, $\boldsymbol{\theta}$ is the parameter that includes all the variables for the calculation of the loss function and n is the number of observations [7].

$$J(\theta) = \mathbb{E}_{x,y} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L(\boldsymbol{x}^{\boldsymbol{(i)}}, y^{(i)}, \boldsymbol{\theta}) \tag{4}$$

For example, if we chose as the loss function the mean square error, Equation 4 would become

Equation 5.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{5}$$

where $\hat{y}^{(i)} = f(\boldsymbol{x^{(i)}})$ is the predicted value of $y^{(i)}$. If we chose as the loss function the mean absolute error, Equation 4 would become Equation 6.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}| \tag{6}$$

If we wanted to combine the Mean Square Error and Mean Absolute Error loss functions together, we could use the Huber loss function [10]. Let S be the set of indices such that $|x| \leq \delta$ holds. Then Equation 4 would become Equation 7.

$$J(\theta) = \frac{1}{n}\left(\sum_{i\in S}\frac{1}{2}x^2 + \sum_{i\notin S}\delta\Big(|x| - \frac{1}{2}\delta\Big)\right) \tag{7}$$

where δ is a hyperparameter.

For these cost functions, gradient descent requires all n datapoints from the training set, which, using big o notation, takes $O(n)$ time [7].

The trick of SGD is to take advantage of the fact that the gradient is an expectation. Instead of using all the data as a single batch, we can break it down into multiple batches, also known as "mini batches", in order to estimate the expectation. The beauty of this technique is that the cost per SGD update is independent of the size of the training set - it only depends on the size of the minibatch which you can choose. This way, each estimate of the gradient is based on each minibatch instead of using the entire dataset each time [7]. Of course, as the size of each minibatch increases, the training error will decrease, but in many cases when you are dealing with millions or billions of observations, you cannot use barebones gradient descent, so SGD provides us with an excellent balance between time complexity and accuracy [8].

2.2 Feedforward Neural Networks

Now that we have a basic understanding of Machine Learning, we can proceed with understanding what neural networks are. Inspired by the biological neural network structure of the brain, neural networks are a collection of artificial neurons known as nodes used for modelling purposes [5]. A neural network is feedforward when data flows from the inputs through to intermediary functions to obtain an output as seen in Figure 3. There are no feedback connections for which data can be fed back into the model [7]. Models with feedback connections are known as Recurrent Neural Networks and will be inspected in Section 2.3.

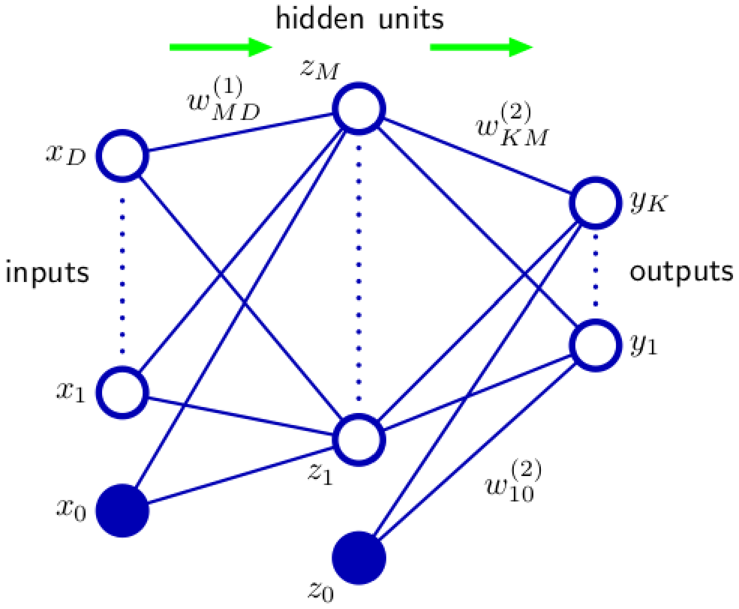

Figure 3: Diagram of a feedforward neural network. $\boldsymbol{x}$ is the input vector, $\boldsymbol{z}$ is the hidden layer vector and $\boldsymbol{y}$ is the output vector. Values x_0 and z_0 are the biases [5].

As we are concerned with supervised learning problems, the aim of a feedforward neural network is to approximate a function f, as seen in Section 2.1.1. The structure of a neural network is of utmost importance in order to understand how it works. A feedforward Neural Network is a Directed Acyclic Graph which encodes the functions that are used and how they are composed [7]. Using Figure 3 as an example, our inputs are $x_1, \ldots, x_D$, our outputs are $y_1, \ldots, y_k$, and any layer in-between the input and output layers are known as a hidden layer. In this example, we only have one hidden layer with neurons $z_1, \ldots, z_M$. Each layer is fully connected with its following layer; these connections are known as weights. They are the parameters of the model which need to be optimized [8].

Continuing with our example, we will now see how data flows through the neural network. We start off by using the input vector $\boldsymbol{x}$ and propagating it through our network. Initially, we must obtain a linear combination for each input vector which results in M variables of the form of Equation 8 [8].

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{8}$$

where $j = 1, \dots, M$ and the subscript (1) signifies that we are dealing with the variables from the first layer of the neural network. The vector w_{ji}, for each j and i, are known as the weights of the model, while w_{j0} are known as the biases (in Figure 3 the input bias is x_0). The results we get from Equation 8 are known as activations. These activation values are then transformed using a differentiable activation function, say $h(\cdot)$, as seen in Equation 9, which is normally non-linear.

$$z_j = h(a_j) \tag{9}$$

The fact that the activation function is non-linear is extremely significant, because this way we are able to introduce non-linearity into our models which greatly expand the capabilities of neural networks. There are a variety of activation functions that are used, including but not limited to: the sigmoid function, hyperbolic tangent function, rectified linear unit function. For this project, we will concern ourselves with the sigmoid and hyperbolic tangent functions, which can be seen in Equations 10 and 11 respectively [5].

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{10}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{11}$$

These z values obtained from Equation 9 are known as hidden units. We then proceed to use these z values by propagating them through the network using Equation 12 to obtain the output unit activations.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{12}$$

where $k = 1, \ldots, K$ and K is the output size. We finally transform a_k using another activation function to obtain the actual output values y_k. After updating the weights of the network, this single cycle of training the neural network with all the training data is known as an epoch [7]. A forward and backward pass are counted as a single pass.

The aforementioned is an example of how a feedforward neural network works. It's complexity can be increased by adding more hidden layers and utilizing different activation functions. The feedforward neural network has proven to be an excellent model for problems, including document recognition [11] and value-at-risk based Asset Allocation [12], to more recent accomplishments such as image classification [13]. However, it struggles with time series analysis as it is unable to retain information, which is extremely important for time series analysis. This is why Recurrent Neural Networks were created.

2.3 Recurrent Neural Networks

When humans think, they do not restart their trail of thought after every second. When humans read text, they read each word and use the previous words they have read to understand the sentence they are reading. Humans don't disregard the previous understanding they have formed from reading earlier words of a sentence. This is the essence of persistent memory. Feedforward neural networks do not have any form of persistent memory [14]. Feedforward neural networks were adapted to have feedback connections so that they could deal with situations which require persistent memory. These modified models are known as Recurrent Neural Networks (RNN) [15].

RNNs are a family of neural networks special for handling sequential data. In order to understand the purpose of RNNs, we need to explore the concept of parameter sharing. Parameter sharing is quite simply the ability to share parameters with different parts of the model; in this context, various time positions. Sharing parameters with different parts of the model allows the model to use the understanding it has gained so far in order to solve the task at hand. Let us look at an example to see the significance of this feature [7].

If we consider the sentences "John went to Italy in 2019" and "In 2019, John went to Italy", one would want the model to read each sentence and extract who went where and when. In this case, we want the model to recognize that the year is 2019 and that John went to Italy. We want the model to recognize this relevant information regardless of the position of the information. If we were to use feedforward neural networks, then we would first need to set the sentence length to a fixed number as they can't handle varying sentence lengths. Then, the feedforward neural network would have separate parameters for each sentence, so it would need to learn all the rules of the English language separately at each position of each sentence [14]. By comparison, a RNN would share the same weights across the time stamps, making it much easier to solve the task at hand. The added benefit is that with RNNs, we can apply the model to varying length sequences which feedforward neural networks can't do [7].

Let us first define important notation and points. The time step index normally refers to some position in a sequence of text or the passage of time. Moreover, RNNs usually operate on batches of sequences, but for the equations presented in this paper, minibatch notation is excluded for simplicity. Lastly, we assume that an RNN operates on a sequence of data $\boldsymbol{x}^{(t)}$ where $t \in \{1, \ldots, \tau\}$. Before learning more about the intricate details of RNNs, one must first understand what a computation graph is, specifically those with cycles [7].

Computation graphs are a way to formalize the structure of multiple computations. They are the main visualizations we will see in order to understand how RNNs work. Another important concept is unrolling, the act of exposing the recursive nature of a graph. For example, by looking at a dynamical system $s^{(t)} = f(s^{(t-1)}; \boldsymbol{\theta})$, where $s^{(t)}$ is known as the state of the system, we can unroll the system by applying the same definition multiple times. So, for $\tau = 3$, we have $s^{(3)} = f(s^{(2)}; \boldsymbol{\theta}) = f(f(s^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})$. Almost any function can be considered a feedforward NN, and in similar fashion, almost any function involving some form of recurrence can be considered a RNN. Many RNNs use Equation 13 to define the values of a hidden unit of a RNN, and are drawn as Figure 4 [7].

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{13}$$

With Equation 13, and as seen in Figure 4, we are able to draw an RNN in two different manners, where their only difference is the compactness of the diagrams. The compact method contains one

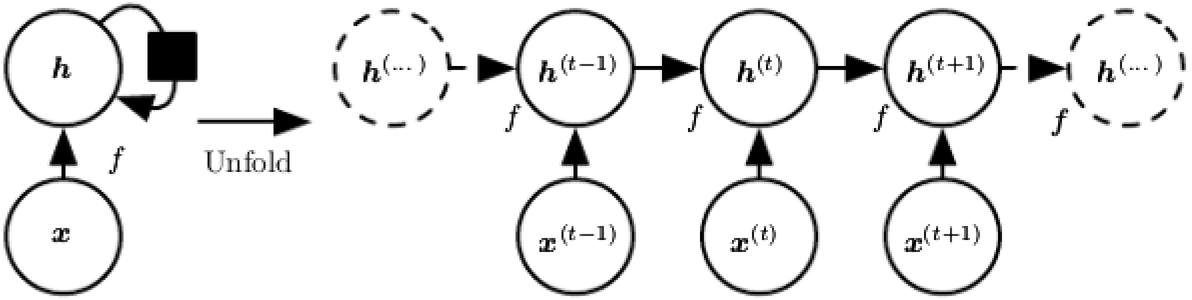

Figure 4: Diagram of a basic RNN. It does not contain any output neurons [7].

node for each type of component that we care about. For an RNN, that would include the input node, hidden node, output node and the true value node (which is used to calculate the loss). A black square in the diagram signifies that an interaction takes place with a time delay of one i.e. from time t to time $t+1$. The other method is to "unfold" the computation graph, where each type of node has a separate node for each time step, representing the state of the component at time t [7]. The size of the unfolded graph depends on the size of the sequence. Unfolding is the act of turning the compact diagram into the version with all the individual components for each time step [5].

We can then represent the unfolded recurrence after t steps with the function $g^{(t)}$ as seen in Equation 14.

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x^{(t)}}, \boldsymbol{x^{(t-1)}}, \ldots, \boldsymbol{x^{(2)}}, \boldsymbol{x^{(1)}}) = f(\boldsymbol{h^{(t)}}, \boldsymbol{x^{(t)}}; \boldsymbol{\theta}) \tag{14}$$

The function $g^{(t)}$ takes an entire sequence up to time t as input in order to produce the current state. Utilizing the fact that the network can be unfolded, we can factorize $g^{(t)}$ into a repeated application of function f. This offers us two key benefits. Firstly, we will always have the same input size, regardless what the sequence length is. This holds as we specified in terms of transitioning from one state to the next rather than variable length inputs. Secondly, we are able to use the same function f for every time step, which means we can use the same parameters. This greatly simplifies the process, as now, we only need to learn a single model f instead of learning a model for each time step. With this single model, we are able to generalize to unseen data a lot better than if we had to make a model per time step [7].

Knowing these important points, we can now look at Figure 5 to see both types of computational diagrams for a classic RNN.

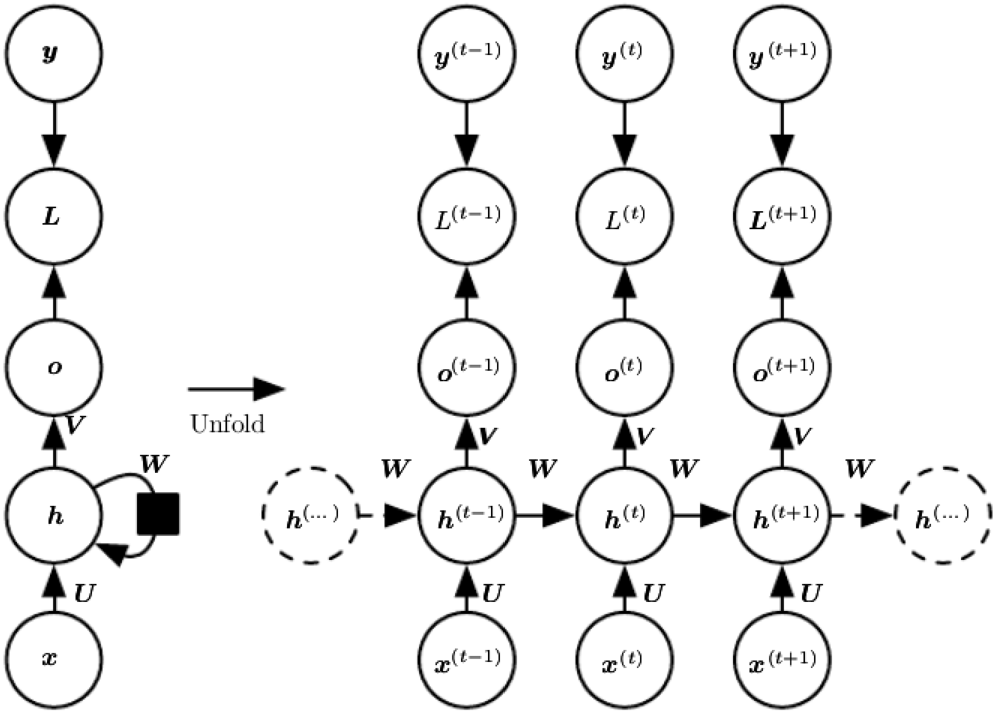

Figure 5: Diagram of a RNN. On the left is the compact version and on the right is the unfolded version [7].

The key components of the RNN are the input node $\boldsymbol{x}$, the hidden unit $\boldsymbol{h}$, the output unit $\boldsymbol{o}$, the true value $\boldsymbol{y}$ and the loss value $\boldsymbol{L}$. The weights are updated after each time step as seen by the black box next to the hidden unit. The loss $\boldsymbol{L}$ measures the deviation of $\boldsymbol{o}$ from $\boldsymbol{y}$. $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ are all weight matrices. The benefit of this RNN is that it can encapsulate any information it wants to be transferred to the next state within the hidden unit $\boldsymbol{h}$ [7].

We now present the forward propagation equations for a RNN with an input sequence length equal to the output sequence length, seen in Equations 15 to 18.

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \tag{15}$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}) \tag{16}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \tag{17}$$

$$\hat{\boldsymbol{y}}^{(t)} = \tanh(\boldsymbol{o}^{(t)}) \tag{18}$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$, and the weight matrices are $\boldsymbol{U}, \boldsymbol{V}$ and $\boldsymbol{W}$. $\boldsymbol{U}$ is the weight matrix for input to hidden connections, $\boldsymbol{V}$ is the weight matrix for hidden to output connections, and $\boldsymbol{W}$ is the weight matrix for hidden to hidden connections. Equation 18 uses the hyperbolic tangent for regression tasks, while a softmax function, seen in Equation 19 [7], is used for classification tasks.

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \tag{19}$$

It was seen in Section 2.1.1 that a dataset is split into training, validating and testing sets so that the model can train, hyperparameters can be fine tuned and then the model's performance can be determined. For RNNs it works slightly differently. Firstly, you still train using a training dataset and validate your hyperparameters using the validation set. However, you then retrain the model using the validation set. The reason for this is because the validation set contains the data closest to the last recorded observation; these datapoints can provide immense power for predicting future cases. If you do not retrain the model, then you lose out in using those datapoints to improve the model. Due to this nuance, it is quite normal to forgo a test set completely, and just test the model on future data.

The main appeal of RNNs is the ability to have persistent memory. In many cases, we only need to look at recent information in order to understand the current situation. For example, let us consider a language model whose aim is to predict the next word given a phrase. If we wanted to predict the last word in the phrase ”the color of this tomato is ...”, we would know with a high probability that it would be the word ”red”. In order to come to this conclusion, the model needs to be able to understand that we are interested in the color of a tomato i.e. learn what the task at hand is and then remember it so that we can make our prediction. However, there are situations where more context is needed, and this context could be in separate sentences or even paragraphs. For example, we may want to predict the next word of the phrase ”I grew up in the UK, ..., I speak fluent ...". By looking at the last words of the sentence, we understand that our prediction should be a language, but in order to know the precise language, we would need to look earlier in the sentence or even before to see that the person grew up in the UK [14].

Researchers began to understand that as this gap between relevant information increases, the difficulty in retaining this information increases exponentially [16] [17]. A primary issue is that as you propagate gradients through a network, especially over long periods of time, they usually begin to vanish or in some rarer case explode, causing destruction to any attempt of solving the optimization problem. Long term values usually have very small weights compared to short term values. This is extremely problematic as it becomes practically impossible to learn, as these small weights can be misinterpreted as small fluctuations arising from short term dependencies [7].

To deal with this issue, LSTMs [18] were created and explicitly designed to deal with long term dependencies.

2.4 Long Short Term Memory (LSTM) models

Gates are a method of determining what information is allowed to pass. All the RNNs that use gates are known as Gated RNNs, and LSTMs are a type of Gated RNN. While other techniques such as Leaky Units are not good at forgetting an old state, LSTMs actively learn when it is time to forget old information that is no longer needed [7]. This is a huge improvement as it no longer requires one to manually decide when to forget the old state, the LSTM simply learns on it's own when it is time to [14].

The core concept that makes LSTMs so effective in handling long term dependencies is the fact that they contain self loops that allow gradients to flow for long periods of time. This limits the magnitude of gradients and ensures they do not explode or vanish, making them usable for learning about an entire sequence of data. After its creation [18], LSTMs were improved so that the weights of the self loop depend on the context of the sequence, rather than being fixed [19]. This brought about even more improvements because the self loop is gated, so the time scale can change dynamically instead of being fixed. Thus, it can change based on the input sequence, remarkably improving the performance of LSTMs [7].

Knowing the fundamentals of LSTMs, we can now look at Figure 6 to understand in more detail the individual components of a LSTM model.

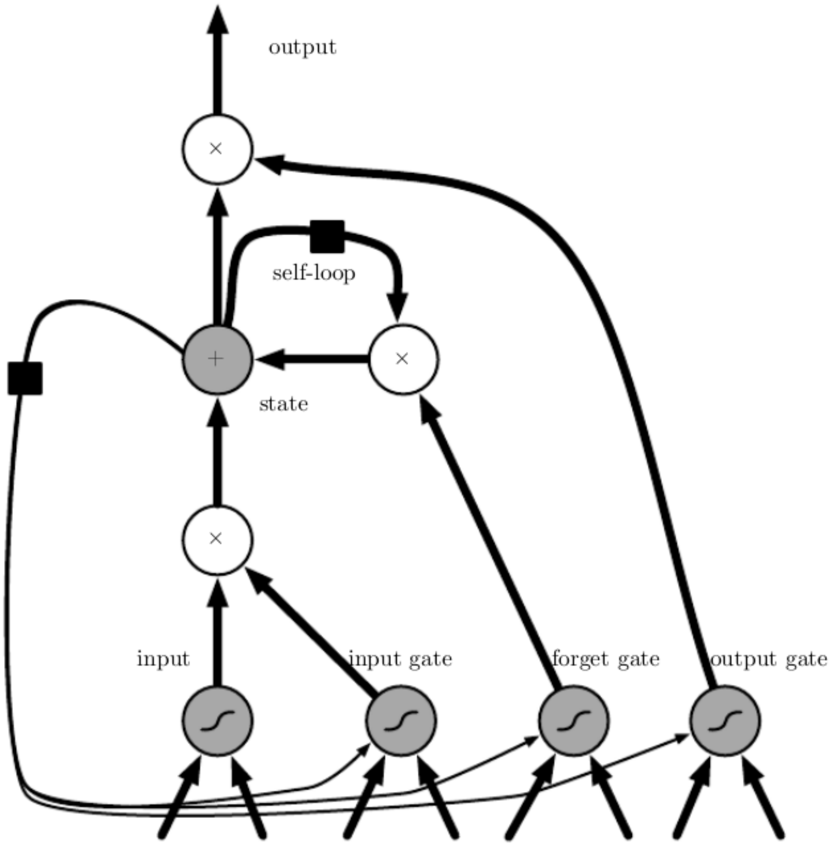

Figure 6: Diagram of a LSTM "cell" [7].

Cells replace the conventional hidden units of regular RNNs and are connected recurrently to one another, as seen in Figure 7. Just like a RNN, each cell has an input and output unit, but also has gates that control the flow of information.

To understand the aforementioned more rigorously, we must inspect the forward propagation equation seen in Equations 20 to 23 [7].

We begin with the forget gate, the job of which is to determine what information is not needed from the cell state anymore. The most important component of the model is the state unit $s_i^{(t)}$ which has a linear self loop that is controlled by the forget gate $f_i^{(t)}$ as seen in Equation 20, where σ is the sigmoid function, $\boldsymbol{b}^f$ is the bias of the forget gate, $\boldsymbol{U}^f$ is the input weights of the forget gate, $\boldsymbol{W}^f$ is the recurrent weights for the forget gate, $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector [7].

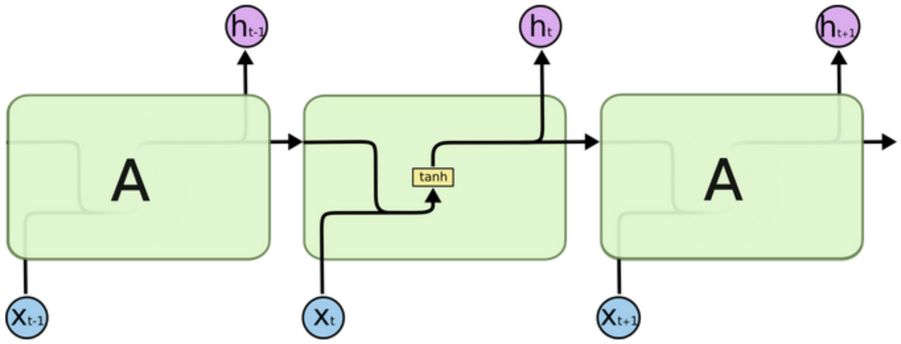

Figure 7: Diagram of the recurrent nature of a LSTM model [14].

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{20}$$

Then, the cell's internal state $s_i^{(t)}$ is updated with a conditional self loop weight $f_i^{(t)}$ as seen in Equation 21, where $\boldsymbol{b}$ is the bias of the cell, $\boldsymbol{U}$ is the input weights of the cell and $\boldsymbol{W}$ is the recurrent weights of the cell [7].

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \tag{21}$$

The external input gate $g_i^{(t)}$ determines which values are updated. Its computation is akin to the forget gate, however using its own parameters as seen in Equation 22 [7].

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{22}$$

Then, the output cell $h_i^{(t)}$ of the cell is a value calculated based on the current state. The output gate determines what parts of the output should be kept. The output cell is controlled by the output gate $q_i^{(t)}$ as seen in Equations 23 and 24 [7].

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right) \tag{23}$$

$$h_i^{(t)} = \tanh(s_i^{(t)})q_i^{(t)} \tag{24}$$

where $\boldsymbol{b}^o$ is the bias for the output gate, $\boldsymbol{U}^o$ is the matrix of input weights for the output gate and $\boldsymbol{W}^o$ is matrix of recurrent weights for the output gate [7].

The aforementioned equations are the forward propagation equations for a barebones LSTM; many variations exists, such as the one seen in Figure 6 which would need another three equations to be complete, as the cell state $s_i^{(t)}$ acts as an extra input for the three gates [7].

LSTMs have proven to be extremely effective in a range of domains, including speech recognition [20] and unconstrained handwriting recognition [21]. This holds especially true for problems which have long term dependencies such as handwriting recognition [22].

2.5 Bidirectional Recurrent Neural Networks

So far we have only looked at RNNs that require information from the past. However, there are many instances when the whole input sequence is needed i.e. information from the future, in order to make an accurate prediction of the present. A key example is when interpreting a sound signal during speech recognition. Many times, in order to make an accurate prediction, one must know the words that follow the current one as well as the previous words, as many words have linguistic dependencies with nearby words [7]. This is even more true when needing to search far away from the current word for context. Bidirectional neural networks were created to deal with this need of requiring the whole input sequence [23].

The core idea of a Bidirectional RNN is that you have two RNNs, one that starts from the beginning of the input sequence and moves forward in time, and one that starts from the end of the input sequence and moves backwards in time. You then combine their results together to make a prediction [7]. Figure 8 displays a classic Bidirectional RNN.

$\boldsymbol{h}^{(t)}$ is the state of the RNN that moves forward in time, while $\boldsymbol{g}^{(t)}$ is the state of the RNN that moves backward in time. Thus $\boldsymbol{o}^{(t)}$ is able to compute a value that depends on both the past and the future.

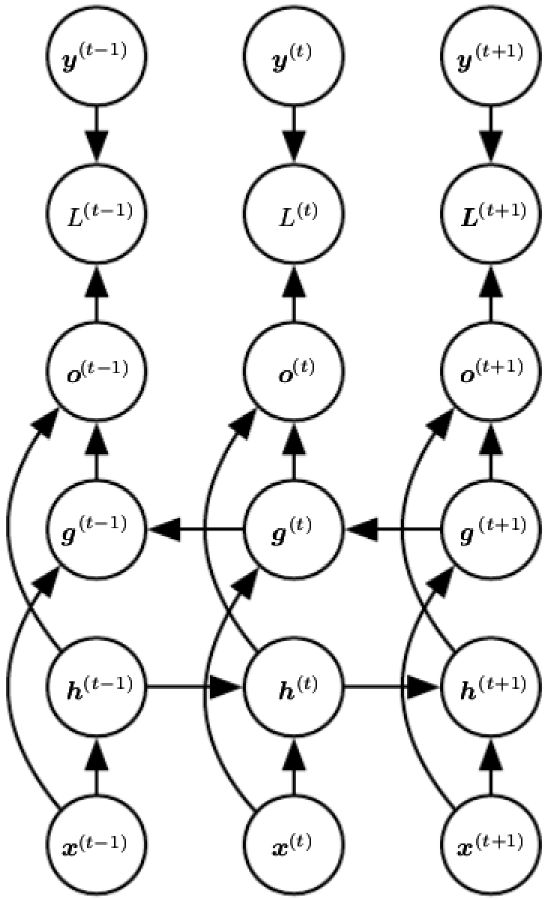

Figure 8: Diagram of a Bidirectional RNN [7].

Bidirectional neural networks have been very successful [24], ranging from handwriting recognition [22] to speech recognition [20]. To deal with problems which require long term memory, we can use a Bidirectional LSTM model instead of a traditional Bidirectional RNN.

3 Dataset

After this discussion on existing neural architectures relevant to situations in which data are sequential, we now understand the theoretical underpinning of the investigation. Now, it is time to look at the dataset that will be used for the problem.

We will be dealing with the cumulative confirmed COVID-19 cases of 173 countries. The dataset is called "COVID-19 Data Repository by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University" [25]. John Hopkins has aggregated data from multiple sources, including but not limited to: the World Health Organization, European Centre for Disease Prevention and Control (ECDC), US Centers for Disease Control and Prevention (CDC).

As we are looking at the COVID-19 cases for each country from the first recorded day, at the time of writing, some countries had to be excluded due to not having cases for over 365 days or even having zero recorded cases. The countries with less than a year of recorded cases are:

- Comoros
- Malawi
- Tajikistan
- South Sudan
- Solomon Islands
- Yemen
- Lesotho
- Vanuatu

The countries that had no recorded cases are:

- Kiribati
- Turkmenistan
- Tonga
- North Korea

Moreover, Kosovo and Timor-Leste were not included as they could not be found in the dataset.

As always, there are drawbacks to using live data. The validity of live data is always questionable, as there can be errors in the systems that upload the data. Additionally, it is important to realize that these recorded cases are a sample of the total number of cases that exist at a point in time. The recorded cases can vary a lot due to the amount of testing that is carried out and how much contact tracing is completed. Moreover, the true number of COVID-19 cases may be hidden to the public due to political factors. Finally, it is important to understand that the confirmed cases of COVID-19 per country is non-stationary.

Figures 9 to 12 display the number of cumulative cases for the United Kingdom, United States of America, Greece and New Zealand from 1st January 2020 to the present day. The first recorded case for the United Kingdom was on 31st January 2020, for the US it was on 22nd January 2020, for Greece it was on 26th February 2020, and for New Zealand it was on 28th February 2020.

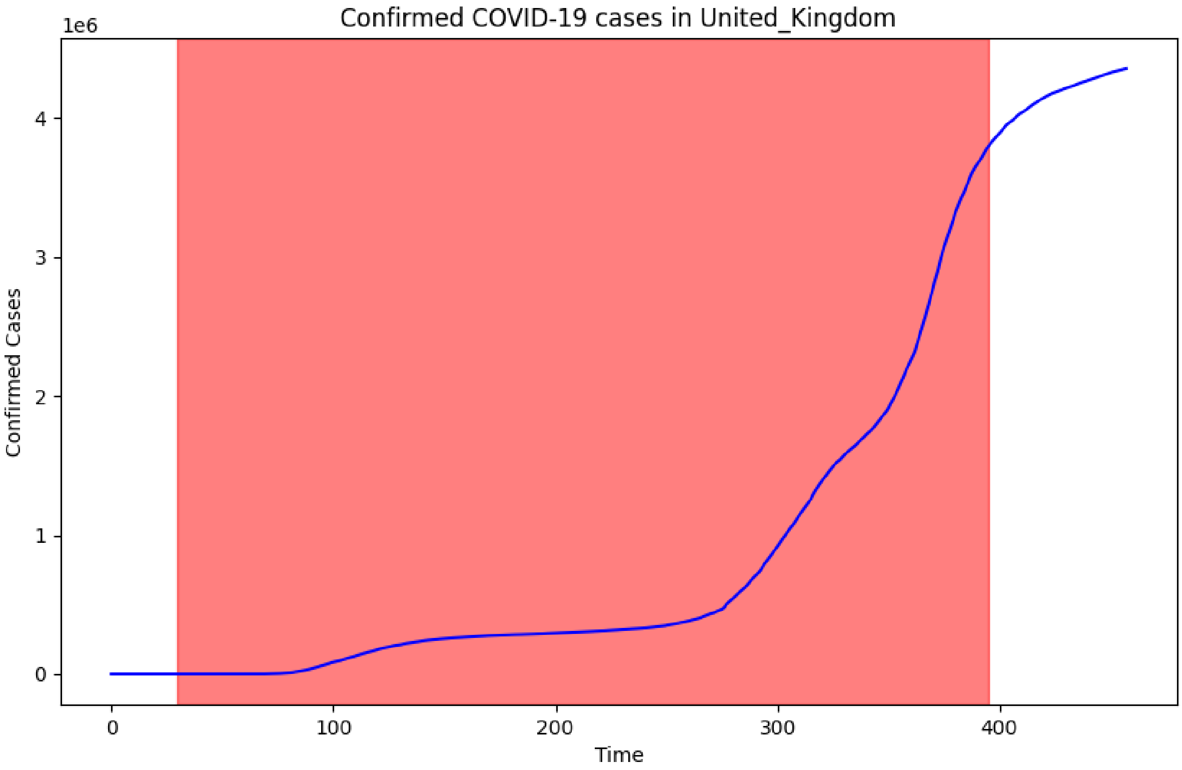

Figure 9: Plot displaying the number of cumulative confirmed COVID-19 cases in the UK. Day zero is 1st January 2020. The red region begins from the first confirmed case, which was on 31st January 2020, until a year after the first confirmed case.

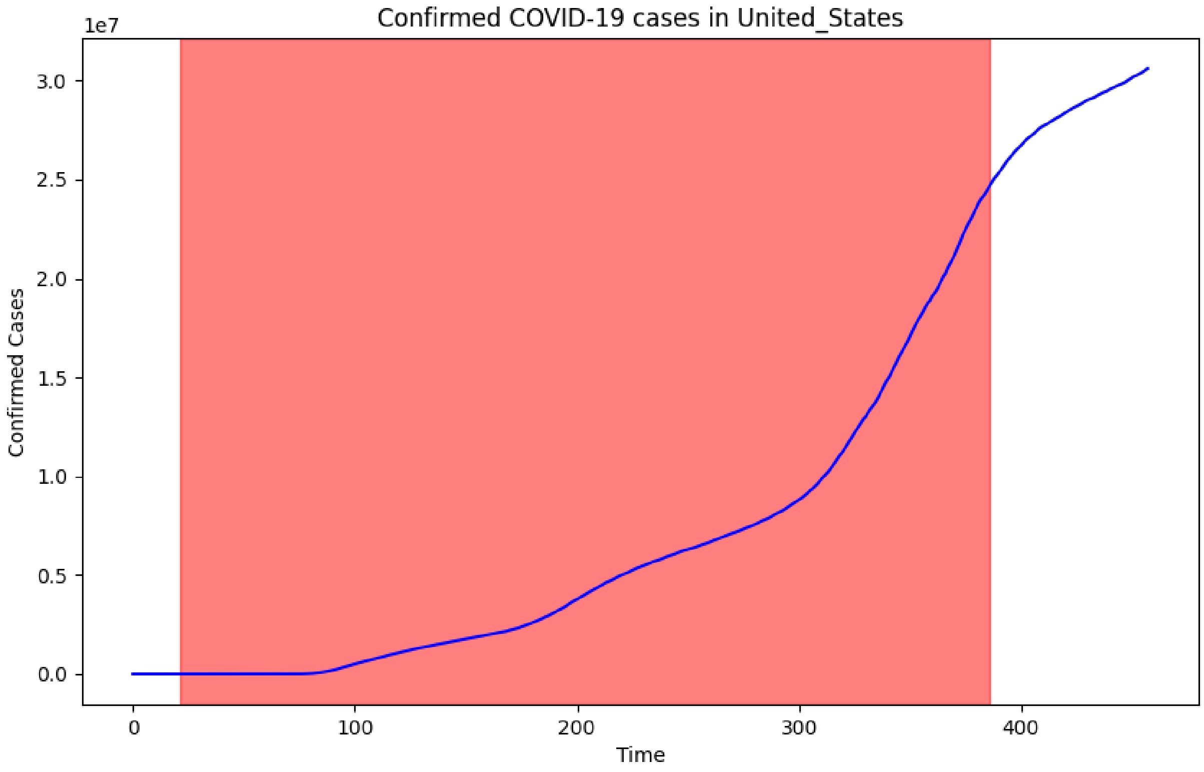

Figure 10: Plot displaying the number of cumulative confirmed COVID-19 cases in USA. Day zero is 1st January 2020. The red region begins from the first confirmed case, which was on 22nd January 2020, until a year after the first confirmed case.

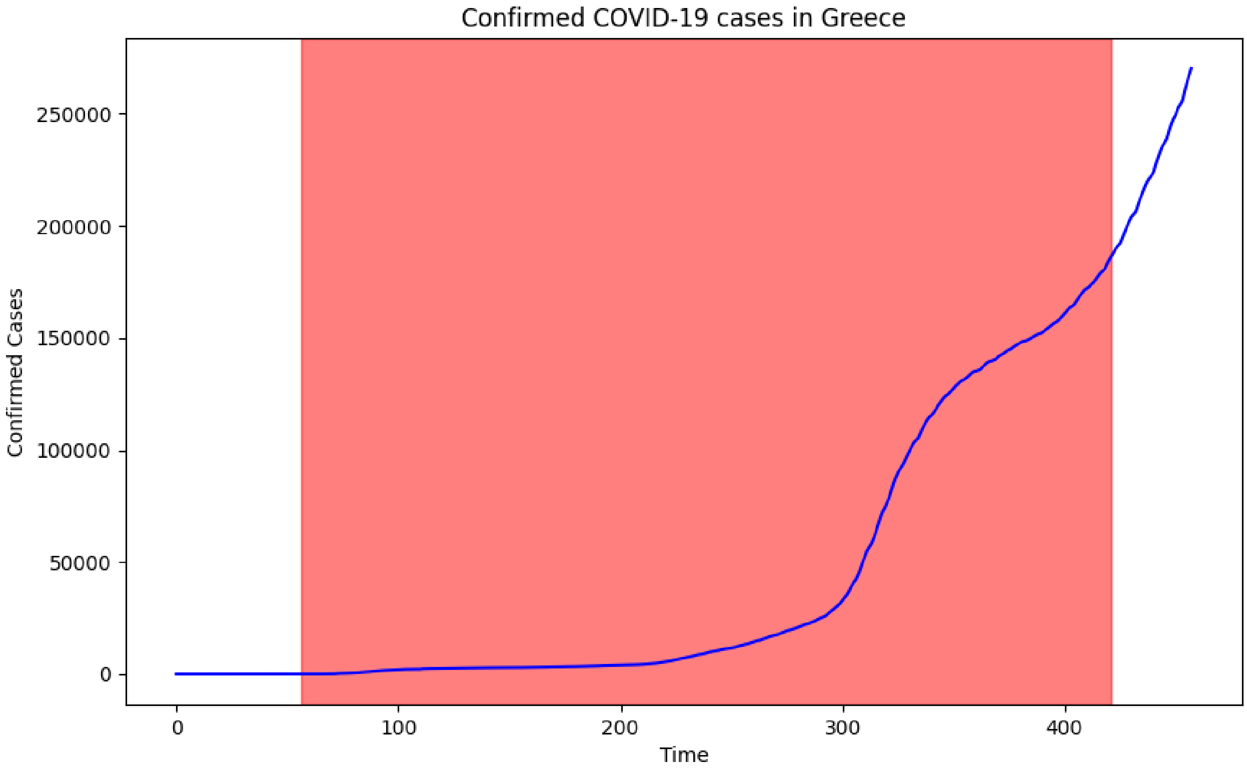

Figure 11: Plot displaying the number of cumulative confirmed COVID-19 cases in Greece. Day zero is 1st January 2020. The red region begins from the first confirmed case, which was on 26th February 2020, until a year after the first confirmed case.

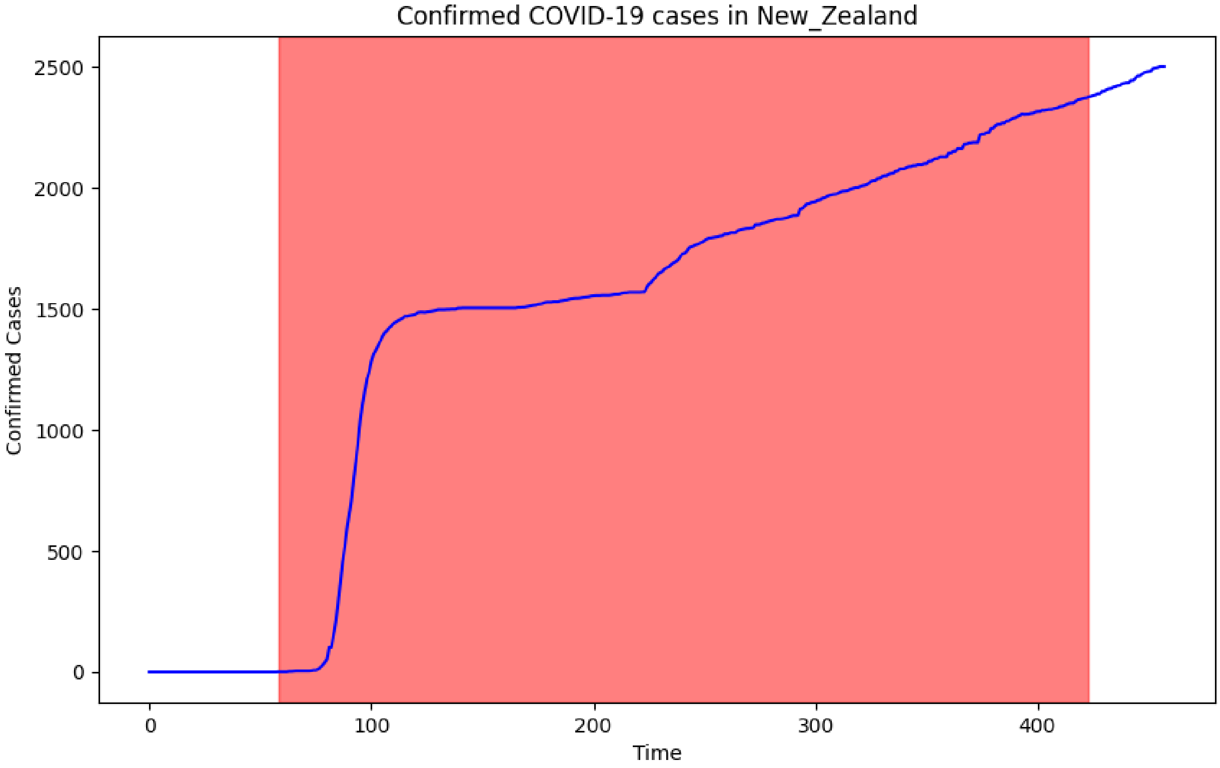

Figure 12: Plot displaying the number of cumulative confirmed COVID-19 cases in New Zealand. Day zero is 1st January 2020. The red region begins from the first confirmed case, which was on 28th February 2020, until a year after the first confirmed case.

4 Explanation of model

Now that we understand what data we are dealing with, we can begin to inspect the precise model that was used. All code can be seen on the GitHub repository [26]. There are a multitude of hyperparameters and variables that need to be set. These include:

- Neural Network architecture
- Activation functions
- Metric
- Loss function
- Optimizer (including learning rate and momentum)
- Epochs
- Training and validating dataset size
- Window size
- Batch size
- Shuffle buffer size

For this investigation, a bidirectional LSTM model was chosen as the *neural network* of choice due to the aforementioned benefits of long term dependencies. The network has a single layer with 64 cells. Each model has in total $33,921$ parameters that have to be optimized. The *activation function* was a hyperbolic tangent, while the recurrent activation function was a sigmoid function. The Mean Absolute Error (MAE) was used as the *metric* as it doesn't penalize large errors as much as other metrics such as the Mean Square Error. Not penalizing harshly large errors is important as it is possible for values to vary due to sampling errors related to COVID-19 testing, especially at the beginning of the pandemic [27].

In addition, the Huber *loss function* [10], defined in Equation 25, was used, as it is quadratic for small values of x, and linear for large values; hence, not penalizing as harshly large errors or anomalies compared to other loss functions.

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \le \delta, \\ \delta\big(|x| - \frac{1}{2}\delta\big) & \text{otherwise.} \end{cases} \tag{25}$$

Stochastic Gradient Descent was the chosen *optimizer* with *momentum* set to zero. The *learning rate* for SGD is a hyperparameter used to control how quickly we change the model after updating the weights. It is an extremely important hyperparameter as having too large of a learning rate means that we update the weights too quickly, while having a very small learning rate can cause the model to get stuck in a local minimum. The learning rate was determined by finding the learning rate that minimized the error of the model. The number of epochs for running the model to find the optimal learning rate was was set to 150 while for training the actual model (using the optimal learning rate), the number of *epochs* was set to 250. The original *dataset was split* into 70% training dataset and 30% validating dataset, an extremely common choice for such a split, as seen in [28] and others.

The *window size* is the number of COVID-19 cases used to predict the following day(s) confirmed cases. In our case, the window size was set to 5 as it usually takes five days for a person who has contracted COVID-19 to be tested positive. We then decided to use these 5 days to predict the 6th day of confirmed cases for the window.

The *batch size* is the number of samples that are propagated through the network at a single moment. For this project, as we are considering the first 365 days worth of confirmed cases; it was decided to set the batch size to 365 so that we provide all of the data in one go. The main reason one uses a batch size smaller than the size of the dataset is because it requires less memory and the model trains faster with minibatches. However, doing so reduces the accuracy of the estimate, so in order to not reduce the accuracy of our models, the batch size was set to the size of the entire dataset.

The *shuffle buffer size* is used to shuffle the dataset. This variable is used so that you don't have to shuffle all the elements at once which can be a costly computation. However, due to the size of the dataset, it was decided to set the variable equal to the size of the dataset as it is not computationally expensive to shuffle 365 observations.

The recurrent dropout is the fraction of the units that are dropped for the linear transformation of the recurrent state. In the case of this investigation, no units were dropped.

In addition, LSTMs are very sensitive to data, so in order to reduce the sensitivity we used the Minimum Maximum normalization technique. Equation 26 is used to scale the feature, while Equation 27 is used to unscale the feature [29].

$$x_{\text{scaled}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{26}$$

$$x_{\text{unscale}} = x_{\text{std}} * (\max - \min) + \min \tag{27}$$

5 Results and Analysis

For each country, a bidirectional LSTM was used to model their COVID-19 confirmed cases. We then inspected the individual models to determine whether or not the model fit the data well.

Looking at the data of Greece and the United Kingdom, we can see from Figure 13 and 14 for Greece and the United Kingdom respectively that they fit the data extremely well. For Greece, the prediction curve is persistently close to the validation curve except for time greater than 340 where the deviation begins to widen. While for the UK, the predictive curve is practically identical to the validation curve until time greater than 320 where the deviation begins to increase. This is reinforced by the mean absolute error, where for Greece it was 0.02152913, while for the UK it was 0.01650905.

We can also inspect how the MAE and loss of the model change as the number of epochs increase over time, as seen in Figure 15 and Figure 16.

We see that for both Greece and the UK, the loss and MAE converge quickly after 100 epochs, so it was decided to reduce the epochs to 250 for all models.

After training a model for each country, their weights were split up based on their layer type and neuron type for further analysis. The allocation can be seen below alongside their dimension size:

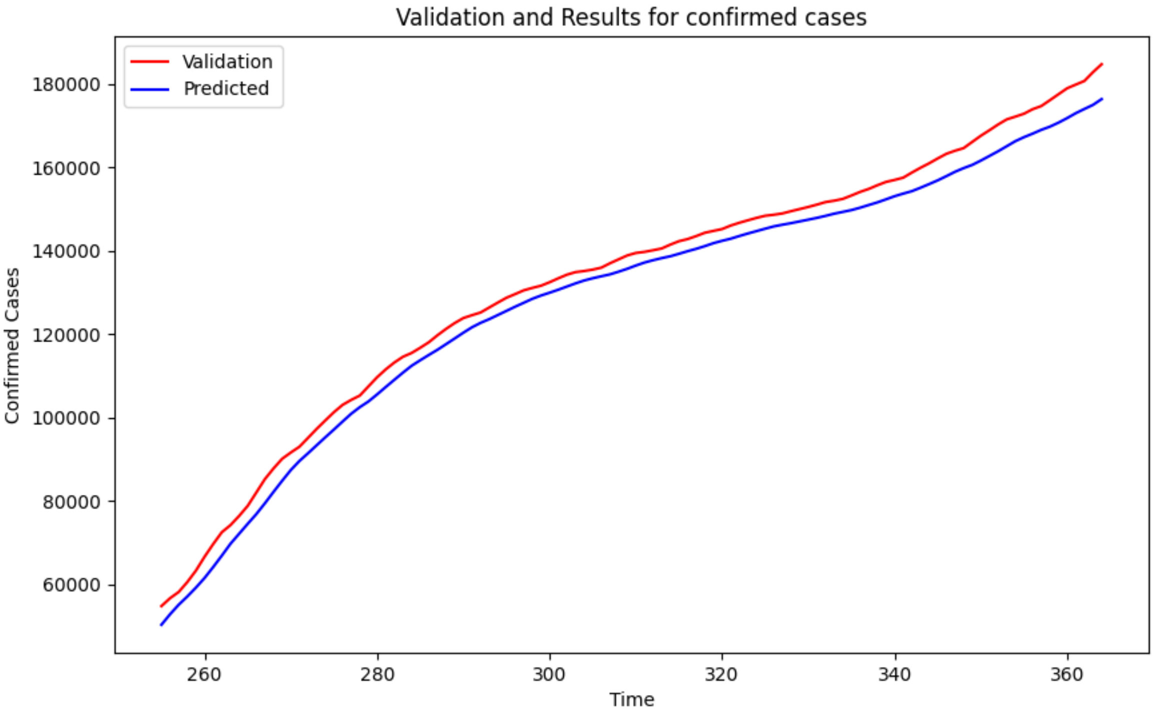

Figure 13: Plot of Greece's true cases and predicted cases for the validation set based on the Bidirectional LSTM model, where day 0 is 26th February 2020.

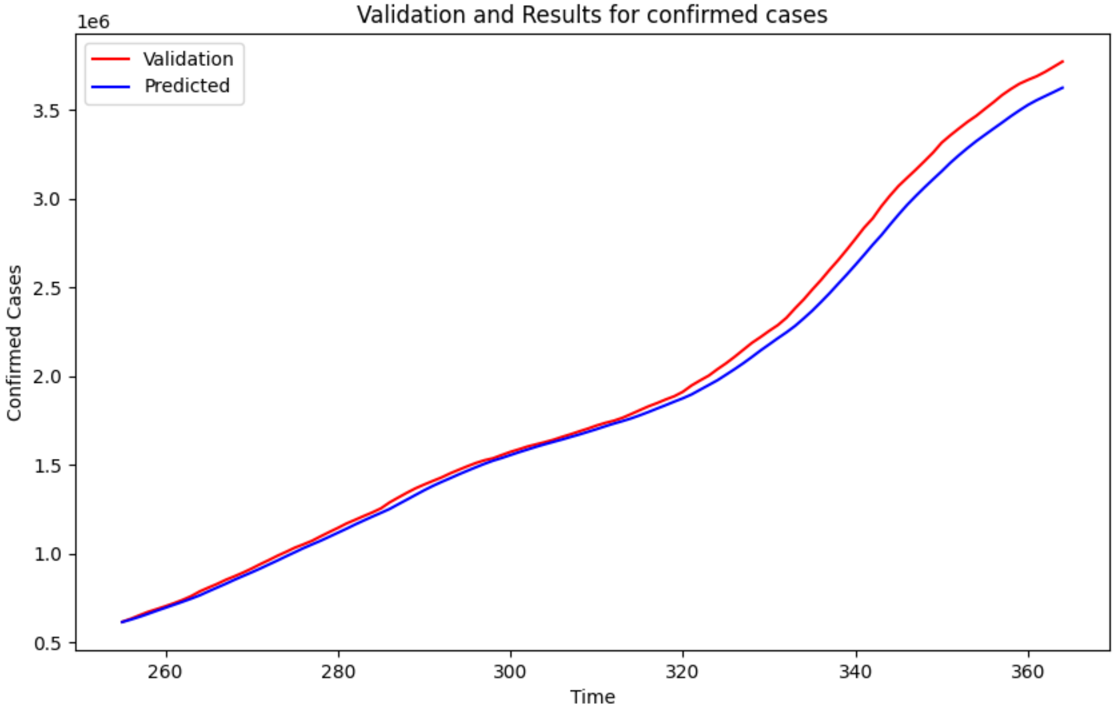

Figure 14: Plot of UK's true cases and predicted cases for the validation set based on the Bidirectional LSTM model, where day 0 is 31st January 2020

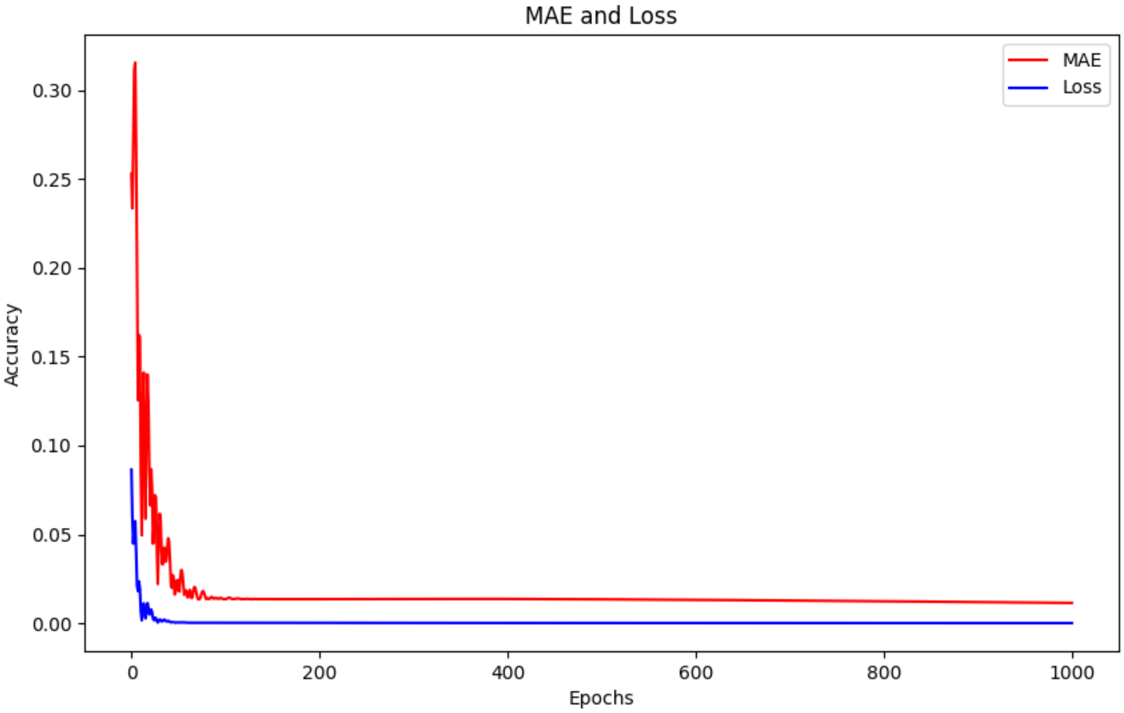

Figure 15: Plot of Greece's loss and MAE with respect to epochs.

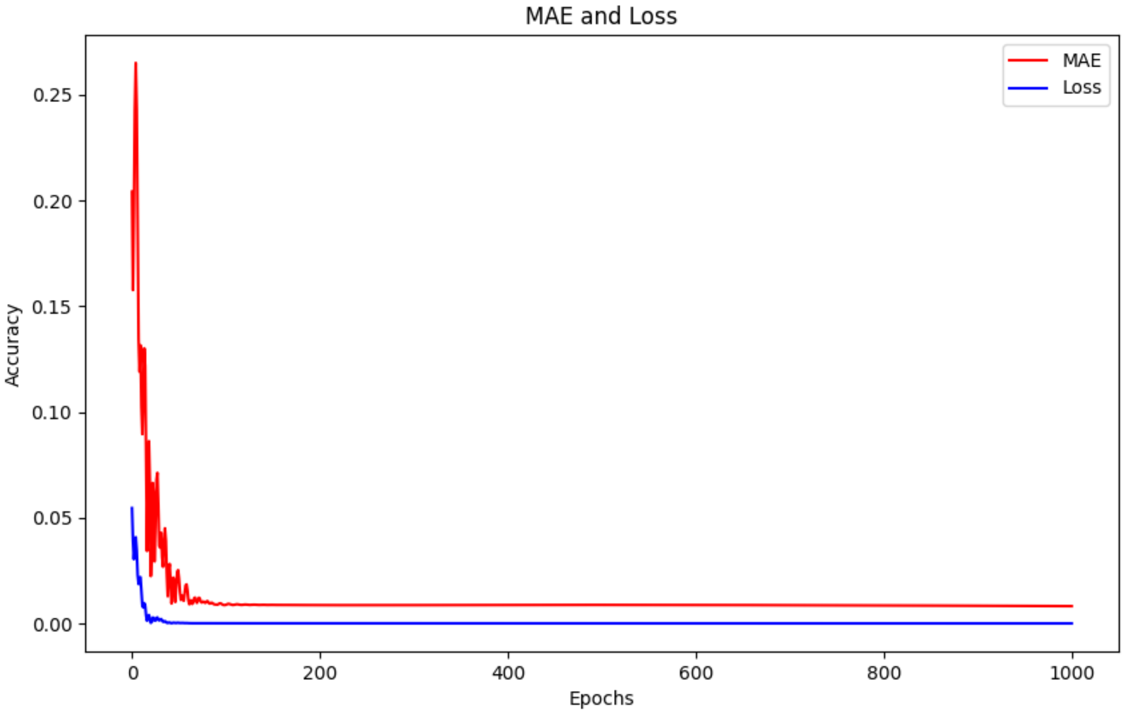

Figure 16: Plot of UK's loss and MAE with respect to epochs.

- Dense bias: (1 × 1)
- Dense kernel: (128 × 1)
- forward bias: (256 × 1)
- backward bias: (256 × 1)
- forward kernel: (256 × 1)
- backward kernel: (256 × 1)
- forward recurrent kernel: (64 × 256)
- backward recurrent kernel: (64 × 256)

Based on these dimensions, it can be seen that the majority of the information is held within the forward and backward recurrent kernel, which is why they will be used for further analysis. It was also seen from the visualizations afterwards that both the forward and backward recurrent kernels performed similarly, so the figures below will all be from the forward recurrent kernel.

After this stage, it was time to begin creating visualizations to interpret the results. Uniform Manifold Approximation and Projection (UMAP) were chosen as the main visualizations technique [30]. UMAP is a dimensionality reduction technique that is used for visualizations, similar to t-SNE, but can also be used for general non linear reduction. UMAP is based on three main assumptions on the data, namely:

1. The data follow a uniform distribution on a Riemannian manifold
2. The Riemannian metric is locally constant or approximately constant
3. The manifold is locally connected

Two UMAP plots were made: one displaying how different continents performed and one highlighting a subset of countries. Inspecting Figure 17, we see that based on continents, Oceania overall seems to be located in the same area, while South America seems to be focused on one half of the map.

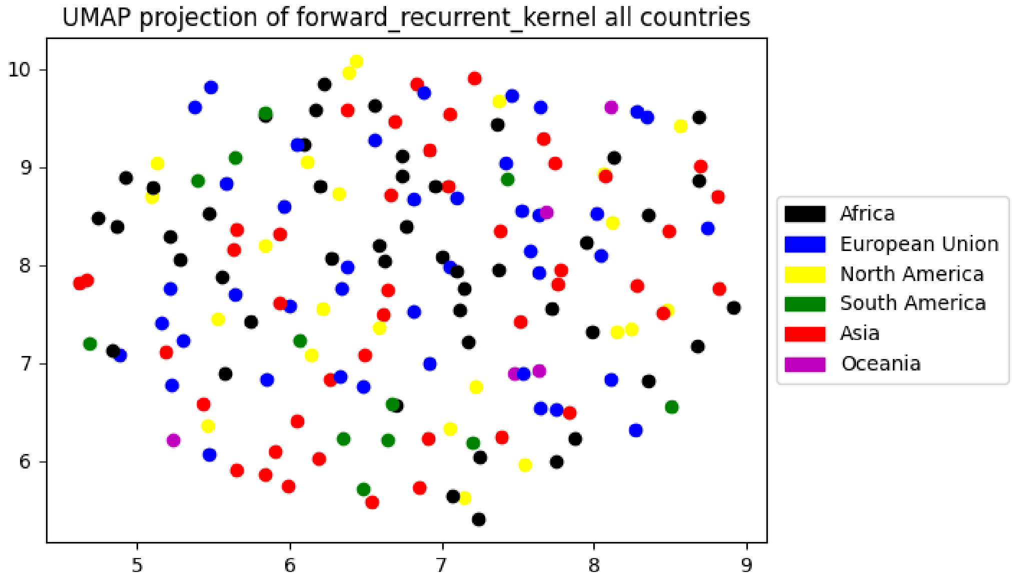

Figure 17: UMAP plot of the forward recurrent kernel layer based on continents. The axes should be seen as arbitrary.

We see in Figure 18 that some countries are located close to one another. Below are some of the clusters identified:

- Greece, South Korea, New Zealand
- USA, South Africa, Brazil

This intuitively makes sense, as Australia, New Zealand etc. performed very similarly, having very few cases for their population sizes, while countries such as USA and Brazil had many cases.

Moreover, we see that overall, the points in the plot are relatively spread out. This could be due to the large amount of noise in the system, as each country implemented different measures to tackle COVID-19.

Taking a closer look at Figure 18, it seems that for large y-axis values, the larger the x-axis value, the more a country's pandemic curve was dominated by its early cases (e.g. Italy and New Zealand who were at their worst during the beginning of the pandemic). While, countries with smaller x-

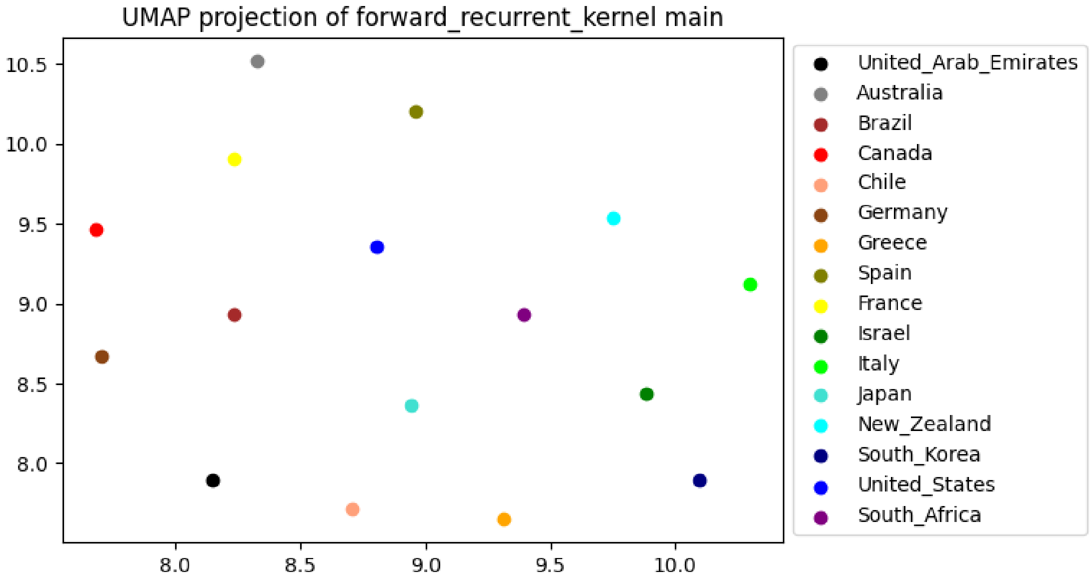

Figure 18: UMAP plot of the forward recurrent kernel layer for a subset of specific countries. The axes should be seen as arbitrary.

axis values (e.g. Canada, France, Spain) had a slow progression at the beginning of the pandemic, but then had significant increases in their second and third waves. This is reinforced by the fact that some countries such as the United States and Spain, which had relatively bad first wave but significant surges of cases to form a second wave, lie within the middle of the x-axis. The only country this does not hold for is Australia, which had an extremely low number of cases and thus are considered anomalous.

It is important to note that the aforementioned observation point holds only for large y-axis values, as for countries with small y-axis values, regardless of their x-axis value, had prolonged periods of low cases at the beginning of the pandemic, such as Greece, South Korea, Japan, and Germany. Countries that lie in the middle of the y-axis tend to have had some form of prolonged periods of low cases but also having sharp increases within their first waves, such as South Africa.

After creating the aforementioned UMAP plots, the question of how much information for each layer is actually useful arose. In order to address this, Johnson-Lindenstrauss (JL) projections were used [31]. The main idea of JL projections is that we can preserve the essential features of a dataset when creating an embedding.

Two methods exist: JL Gaussian transformation and JL Sparse transformation [31]. The Gaussian Transformation reduces the dimensionality by taking the original dataset and projecting it onto a space using a randomly generated matrix, whose components are from a $\mathcal{N}(0, \frac{1}{n})$, where n is the number of components. The Sparse transformation reduces the dimensionality by taking the original dataset and projecting it onto a space using a sparse random matrix. The main difference between the two is that the sparse transformation is much more memory efficient while being computationally quicker, all while having similar quality.

More rigorously, we create a mapping $F : \mathbb{R}^d \to \mathbb{R}^m$ where m is much smaller than d. Given some tolerance δ, F is a mapping that satisfies Equation 28.

$$1 - \delta \leq \frac{\|F(u^i) - F(u^j)\|_2^2}{\|u^i - u^j\|_2^2} \leq 1 + \delta \tag{28}$$

for all pairs $u^i \neq u^j$. We are creating a projected dataset where all pairwise squared distances are preserved up to a multiplicative factor of δ. Creating such a mapping with high probability in reality isn't hard to create, provided that the projected dimension has as a lower bound of $\frac{1}{\delta^2}\log(N)$. It is crucial to notice that the projected dimension does not depend on the original dimension of the dataset: it is dependent only on the number of observations.

Each vector was originally of dimension 16384. As there are 173 countries being used, we were dealing with a (173×16384) matrix. By setting δ to 0.1, after the Johnson-Lindenstrauss transformation, it became a (173×4426) matrix. The transformation reduces the data by a factor of $\frac{16384}{4426} \approx 3.7$.

Beginning with the Gaussian transformation, by looking at Figure 19 we see that the main clusters identified are:

- Australia, Greece
- South Korea, New Zealand
- Canada, Brazil, Italy
- USA, Chile

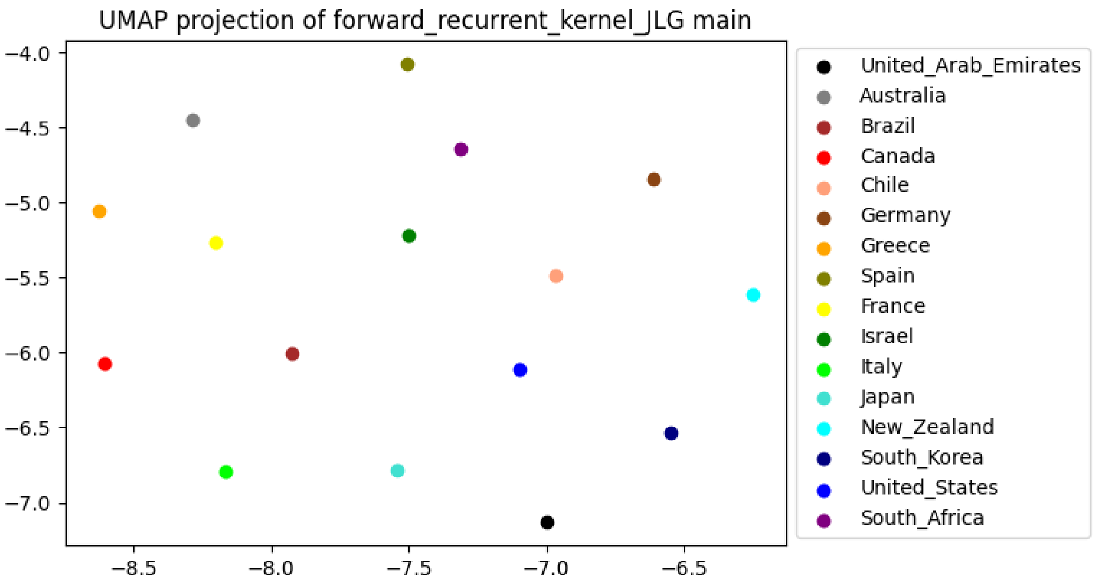

Figure 19: UMAP plot of the forward recurrent kernel layer for a subset of specific countries after performing Johnson-Lindenstrauss Gaussian transformation. The axes should be seen as arbitrary.

These clusters also make sense, however some of the original clusters identified from Figure 18 were split e.g. Greece was grouped with Australia instead of South Korea and New Zealand. Even more intriguing, the aforementioned cluster was split and is on opposite ends of the figure, with Greece in the top left corner of the plot, while South Korea and New Zealand are on the far right side of the plot.

One important observation is that the larger the x-axis value, the more likely a country had a long periods of slowly increasing cases (without specifying at what time period this occurred) such as South Korea, New Zealand, while smaller x-axis values indicate periods of extremely quick increases in the number of cases e.g. Australia, Greece and France. This contrasts with the original finding of the x-axis for Figure 18, but does resemble slightly the observation made. This abstraction in the observation could be because the transform only keeps the basic patterns, but fails to address the more detailed patterns.

By comparing Figure 19 with Figure 18, most countries seem to be around the same position in both plots or have moved slightly, with some exceptions such as Italy, UAE and Israel which are now on the opposite side of where they originally were.

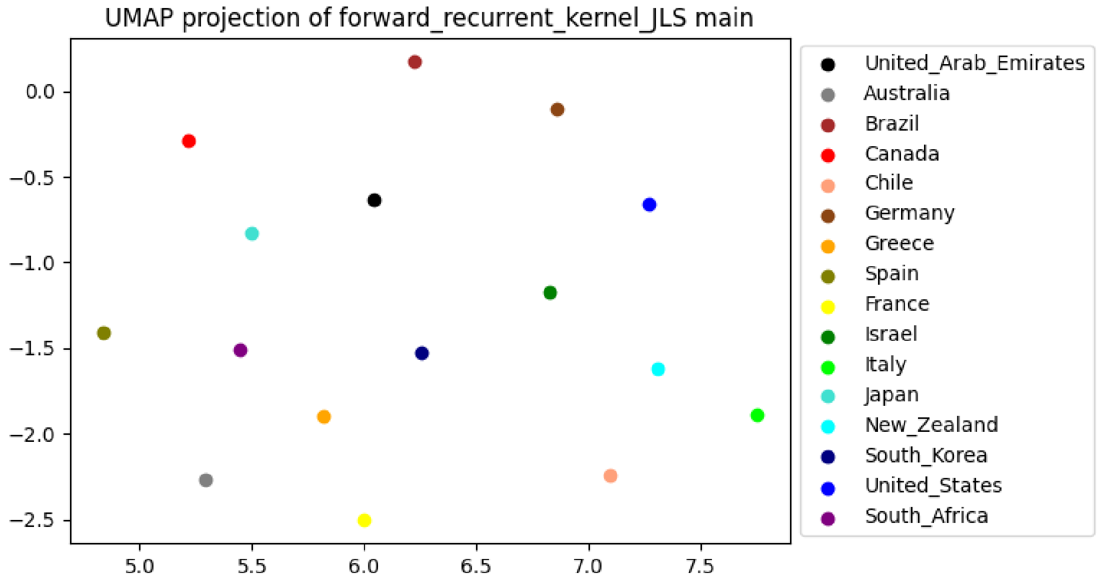

Figure 20: UMAP plot of the forward recurrent kernel layer for a subset of specific countries after performing Johnson-Lindenstrauss Sparse transformation. The axes should be seen as arbitrary.

Moving onto the sparse transformation, by looking at Figure 20 we see that we see that the main clusters identified are:

- New Zealand, Italy, Chile
- Greece, South Korea, Israel
- Japan, UAE, Canada

Again, overall these clusters are logical, however, similarly to the Gaussian transformation, some of the original clusters were split or do not exist e.g. New Zealand being in a separate cluster from Greece and South Korea.

An important observation is that the Sparse transformation has the same pattern as Figure 18, in that the larger the x-axis value, the more a country's pandemic curve was dominated by its early cases (e.g. Italy and New Zealand who were at their worst during the beginning of the pandemic).

While, countries with smaller x-axis values (e.g. Canada, Japan, Spain) had a slow progression at the beginning of the pandemic, but then had significant increases in their second and third waves.

Furthermore, by comparing Figure 20 with Figure 18, we see that around half of the countries are in the same location in both figures (Japan, Italy, South Africa) and the other half deviate significantly (Germany, Australia, France). However, it seems that the countries that have moved went closer to the clusters they were originally in, rather than mixing and creating new ones. For example, New Zealand moved even closer to Italy and Israel than it was originally located.

By comparing the Gaussian and Sparse transformations, it seems that the Sparse encapsulated more information within the transformation than the Gaussian transformation. The Sparse transformation is more similar to Figure 18 than the Gaussian transformation. This could be due to the Gaussian transformation creating dense Gaussian random projection matrices while the Sparse transformation creates sparse random matrices.

UMAPs aren't the only visualization that can be used to analyze the problem at hand. Another insightful option is Functional Principle Component Analysis (FPCA). FPCA, for this investigation, is a statistical method used to obtain a small set of curves known as eigencurves that can explain the majority of the original curves. By applying this to all the epidemic curves showing all the confirmed cumulative COVID-19 cases, as seen in Figure 21, we can obtain the 3 principle components, as seen in Figure 22 [32].

The eigencurves suggest that most nations either follow curve 1, with a trajectory in which COVID-19 cases spiral out of control quickly, or curve 2 and 3, where there are some increases in cases but overall the curve is relatively flat. The latter implies that acting early is critical in reducing the number of cases within a country. Additionally, there doesn't seem to be a middle ground, considering the large difference between curve 1 with curve 2 and 3. This point is further reinforced by the aforementioned UMAP results we obtained.

After having analyzed the performance of countries holistically, we now move on to analyzing individual countries. This will be done by using a word cloud. A word cloud is simply a diagram that shows words on a plane, where the size of the word is determined by a metric. In this case, we use the total number of cases after a full year from the first day of a recorded case of COVID-19.

As seen in Figure 23, it is clear that the United States dominates the plot, having the most number

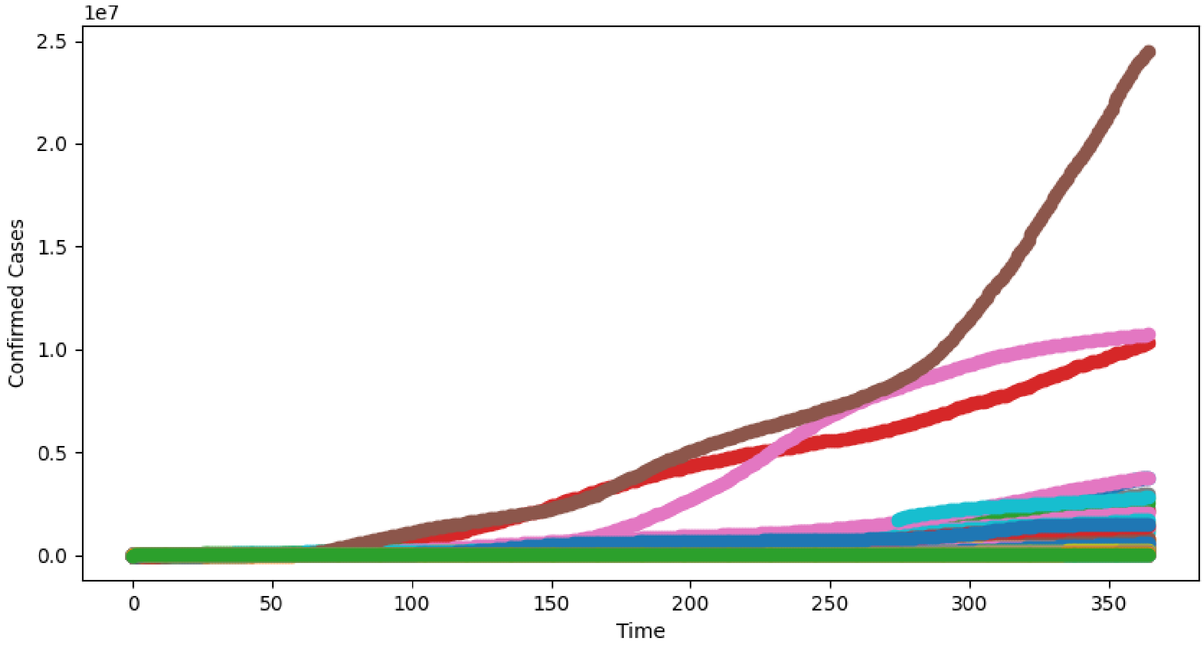

Figure 21: Plot showing all the confirmed cumulative COVID-19 cases over the year from their first confirmed case, for all 173 countries considered in the investigation. Day 0 is the first day of a confirmed case for each country.

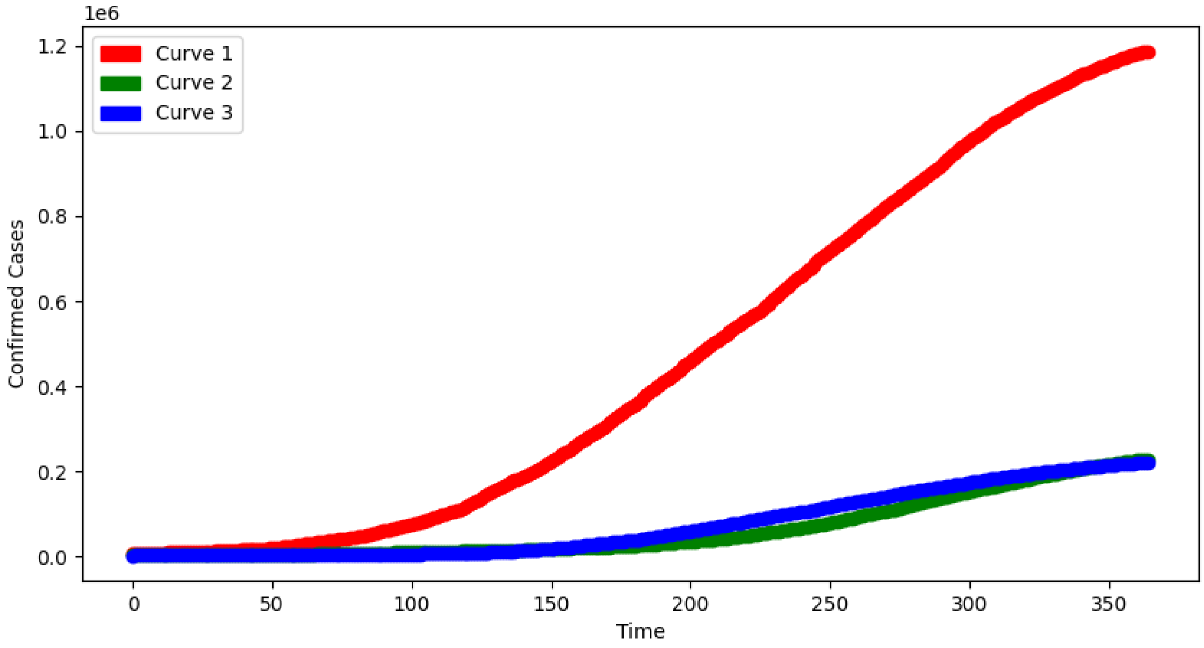

Figure 22: 3 principle eigencurves obtained after applying joint Functional Principle Component Analysis. Day 0 is the first day of a confirmed case.

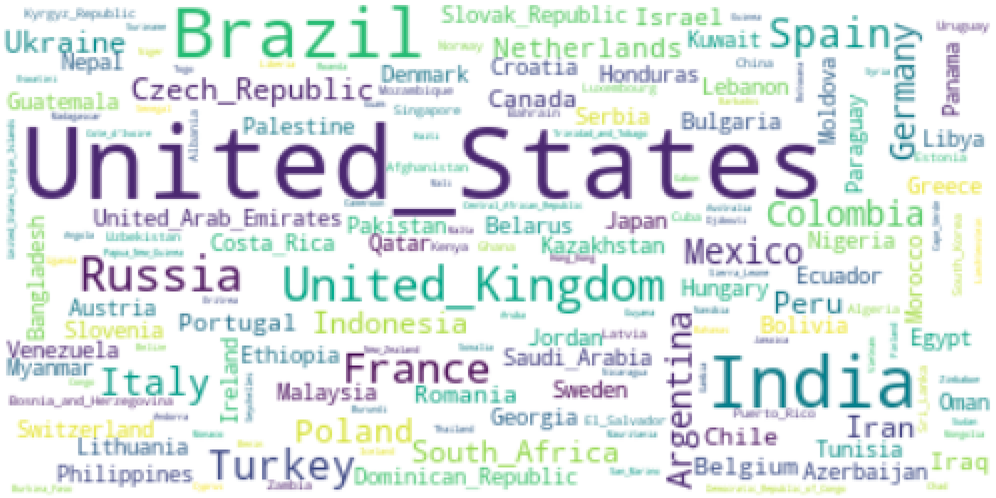

Figure 23: Word cloud displaying all the countries in the dataset. The size of a word is based on the country's total number of cases after 365 days from the first recorded case.

of cases within the first year, followed by Brazil, India and the United Kingdom. Greece is on the right hand corner of the plot, but is much harder to see, reinforcing that Greece had very few cases in the first year of recorded cases. New Zealand had so few cases that they almost cannot be seen.

In hindsight, there are some improvements that could have been made to improve the results of the investigation. One improvement would have been to use a different optimizer. Instead of Stochastic Gradient Descent, the Adam optimizer could have been chosen, which handles very well sparse gradients on noisy problems [33], or the rectified Adam optimizer which can obtain a higher accuracy with fewer epochs compared to the Adam optimizer [34]. It could even be argued that Backpropagation Through Time would have been a better alternative, as it was specifically made for Recurrent Neural Networks [35]. In addition, as an extension, it would be interesting to model how Non-Pharmaceutical Intervention (NPI) methods effect the performance of a country with regards to COVID-19 cases, so that one could determine which NPIs result in the lowest number of COVID-19 cases and deaths.

6 Conclusion

In conclusion, the weights of a bidirectional LSTM encode information about the performance of a model with regards to COVID-19 cases. The weights of one country can be compared with the weights of other countries, to determine how one country performed compared to others with respect to how much COVID-19 spread within their country. Countries such as Greece, New Zealand and South Korea had very few cases within the first year of confirmed cases, while countries such as the United States, Brazil and the United Kingdom had many cases.

Lastly, the importance of acting early cannot be overstated. Implementing measures to reduce the spread of COVID-19 early on does indeed result in significantly less cases, as seen from the analysis of the UMAP results the Functional Principle Component Analysis figures. The analysis reflects the reality of countries such as Greece and New Zealand that implemented measures early on and had minimal first waves of cases. Failing to act early can result in a huge number of COVID-19 cases, as seen by countries such as the United Kingdom and Brazil.

References

[1] Yi-Cheng Chen et al. “A Time-Dependent SIR Model for COVID-19 With Undetectable Infected Persons”. In: *IEEE Transactions on Network Science and Engineering* 7.4 (2020), pp. 3279–3294. DOI: `10.1109/tnse.2020.3024723`.

[2] Ahmad Sedaghat and Amir Mosavi. “Predicting COVID-19 (Coronavirus Disease) Outbreak Dynamics Using SIR-based Models: Comparative Analysis of SIRD and Weibull-SIRD”. In: (2020). DOI: `10.1101/2020.11.29.20240564`.

[3] Ahmad Sedaghat and Amir Mosavi. “COVID-19 (Coronavirus Disease) Outbreak Prediction Using a Susceptible-Exposed-Symptomatic Infected-Recovered-Super Spreaders-Asymptomatic Infected-Deceased-Critical (SEIR-PADC) Dynamic Model”. In: (2020). DOI: `10.31219/osf.io/xevck`.

[4] Shiva Moein et al. “Inefficiency of SIR models in forecasting COVID-19 epidemic: a case study of Isfahan”. In: *Scientific Reports* 11.1 (2021). DOI: `10.1038/s41598-021-84055-6`.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. ISBN: 978-0387-31073-2.

[6] Bernard Marr. *Big Data: 20 Mind-Boggling Facts Everyone Must Read.* Nov. 2015. URL: `https://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read/?sh=4d8bbda817b1`.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[8] Kevin P. Murphy. *Machine learning: a probabilistic perspective.* The MIT Press, 2012.

[9] Augustin-Louis Cauchy. “Méthode générale pour la résolution des systèmes d’équations simultanées”. In: *C. R. Acad. Sci. Paris* (1847), 25:536–538.

[10] Peter J. Huber. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101. DOI: `10.1214/aoms/1177703732`.

[11] Yann Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[12] Nicolas Chapados and Yoshua Bengio. "Cost functions and model combination for VaR-based asset allocation using neural networks". In: *IEEE Transactions on Neural Networks* 12.4 (2001), pp. 890–906. DOI: 10.1109/72.935098.

[13] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012). DOI: 10.1109/cvpr.2012.6248110.

[14] Christopher Olah. *Understanding LSTM Networks.* Aug. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[15] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[16] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06.02 (1998), pp. 107–116. DOI: 10.1142/s0218488598000094.

[17] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[18] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[19] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (2000), pp. 2451–2471. DOI: 10.1162/089976600300015015.

[20] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013). DOI: 10.1109/icassp.2013.6638947.

[21] Alex Graves et al. "A Novel Connectionist System for Unconstrained Handwriting Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5 (2009), pp. 855–868. DOI: 10.1109/tpami.2008.137.

[22] Alex Graves. "Offline Arabic Handwriting Recognition with Multidimensional Recurrent Neural Networks". In: *Guide to OCR for Arabic Scripts* (2012), pp. 297–313. DOI: 10.1007/978-1-4471-4072-6_12.

[23] Mike Schuster and Kuldip Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681. DOI: 10.1109/78.650093.

[24] Alex Graves. "Supervised Sequence Labelling". In: *Studies in Computational Intelligence* (2012), pp. 5–13. DOI: 10.1007/978-3-642-24797-2_2.

[25] E Dong, H Du, and L Gardner. *JHU CSSE COVID-19 Data*. URL: https://github.com/CSSEGISandData/COVID-19.

[26] Dimitrios Pilitsis. *Covid-19 RNN Math Project*. May 2021. URL: https://github.com/Dimitrios-Pilitsis/covid_rnn_math_project.

[27] Simon N. Wood. "Inferring UK COVID-19 fatal infection trajectories from daily mortality data: Were infections already in decline before the UK lockdowns?" In: *Biometrics* (2021). DOI: 10.1111/biom.13462.

[28] M. Stone. "Cross-Validatory Choice and Assessment of Statistical Predictions". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 36.2 (1974), pp. 111–147. DOI: 10.1111/j.2517-6161.1974.tb00994.x.

[29] Scikit Learn. *Minimum Maximum Scaler*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[30] Leland McInnes et al. "UMAP: Uniform Manifold Approximation and Projection". In: *Journal of Open Source Software* 3.29 (2018), p. 861. DOI: 10.21105/joss.00861.

[31] William B. Johnson and Joram Lindenstrauss. "Extensions of Lipschitz mappings into a Hilbert space". In: *Conference on Modern Analysis and Probability* (1984), pp. 189–206. DOI: 10.1090/conm/026/737400.

[32] Derek Tucker. *Functional Data Analysis using the Square Root Slope Framework (fdasrsf)*. July 2013. URL: https://github.com/jdtuck/fdasrsf_python.

[33] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017). arXiv: 1412.6980 [cs.LG].

[34] Liu Liyuan et al. "On the Variance of the Adaptive Learning Rate and Beyond". In: (2020). URL: https://openreview.net/forum?id=rkgz2aEKDr.

[35] P.J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: `10.1109/5.58337`.