Physics Informed and Recursive Neural Networks for Ordinary Differential Equations

By

Ronald Alexander Balint

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of Masters of Science in General Mathematics

Middle Tennessee State University

Fall 2023

Thesis Committee:

Dr. Abdul Khaliq, Chair

Dr. Wandi Ding

Dr. Rachael Leander

Dr. James Hart

Contents

List of Figures

Abstract

The human brain is an amazing organ. This 3-pound mass of fats and proteins is able to take signals from all over the body, process those signals, and determine the best course of action, all instantly. The brain has the ability to learn, remember, and forget information throughout its lifetime. Brains have certain regions that handle certain tasks. This organ is being constantly studied and researchers are constantly finding out new information about this organ. Modern computer scientists have been trying to re-create this organ in a digital form. Computers are being programmed to take signals, in the form of data, process that data, transform the processed data into a new signal, and generate meaningful output. This is done with data processing, training, validating, and testing known sets of signals and desired outputs. Recently, the use of customizable loss functions gave rise to the subfield of Physics Informed Neural Networks (PINN) and using these Informed Neural Networks to estimate parameters of Ordinary Differential Equations (ODE). This paper will investigate and compare different network structures for this task.

1 Introduction – What is a neural network

Machine Learning is a field of computer science that uses statistical techniques to give computer systems the ability to 'learn' with data, without being explicitly programmed [14]. One facet of machine learning is Representation Learning. Representation Learning is a set of methods that allows a computer to take data and automatically discover the representations needed for classification or detection. Deep Learning takes Representation Learning to another scale by repeating the process by composing simple, non-linear modules that transform the representation at one level into a representation at a higher, more abstract level (LeCun, 2015). The simple non-linear module is the activation function.

1.1 Overview of Neural Networks

A neural network takes an input vector of length p and builds a nonlinear function to predict a response. The input vector is referred to as the input layer and the output vector is referred to as the output layer. The first step in the calculation is to take each value in the input vector X_i, multiply each value by a weight w_i, sum the product $\sum_{i=1}^{N} X_i w_i$, and add a bias amount b_i. The calculation for each combination is called a *node* or *neuron* and would then be

$$_l = \sum_{i=1}^{N} X_i w_i + b_i.$$

The weight and bias are trainable parameters. This sum is transformed using an activation function $y_l = f(\sigma_l)$. The activation function must be nonlinear, otherwise, the structure will be a linear model. [12]. The resulting values through the activation function can then be used as new input values, or as the final, output, vector. The collection of intermediate combinations of previous outputs to form new inputs is

referred to as a hidden layer. The process of taking the intermediate calculations and using them as an input for a new layer is referred to as propagation [14] This construction yields a collection of composite functions

$$y_i^M = F(y_i^{\{M-1\}})$$

.

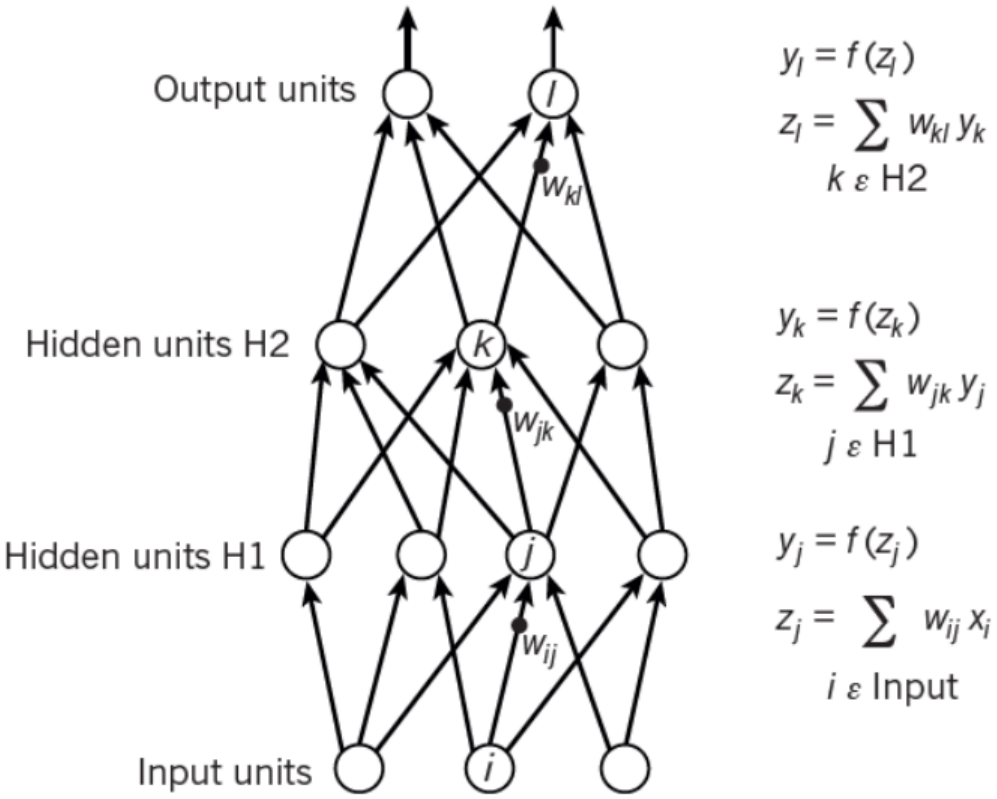

Fig. 1: A sample dense neural network with 2 hidden layers.

1.2 Data

An NN is able to work with several types of data, the only stipulation is that the data needs some numeric value attached to it so the activation functions are able to perform their calculations. The data is stored is through a matrix, which is called

a *tensor*. A tensor is a generic way to refer to data structures. A scalar is a *rank-0 tensor*. A vector is a *rank-1 tensor*. A matrix is a *rank-2 tensor*. To visualize a *rank-3 tensor*, visualize a stack of matrices. Each higher rank tensor is going to be a 'stack' of lower rank tensors [3]. This definition makes sense since a vector is a stack of scalars, and a matrix is a stack of vectors. These tensors are able to translate different types of data into numeric values. For example, an image is made of pixels. Each pixel has a color and a location. The locations can be a sequence starting from the top left of the image and reading each pixel to the lower right. Each color for each pixel can be represented in a 1-color code, the number being a grayscale, a 3-color code, each number referencing the colors Red, Green, or Blue, or a 4-color code, with each number representing Cyan, Magenta, Yellow, and Black. A video file would add another dimension: time. Also, audio files can be broken into tensors.

1.3 Types of Neural Networks

There are numerous ways that the hidden layers and the network as a whole can be structured. Each type of network has different kinds of problems that are suited for that network.

1.3.1 Convolutional Neural Network (CNN)

A CNN is a type of network that is based on how humans process images. Starting with an image, the CNN tries to identify features based on the whole image, looking at each pixel, or a small region of pixels. The network then uses a *Convolution* layer, which is made up of a number of learnable filters that isolate various features common to a class. This layer is based on a *convolution*, treating each group of pixels as a matrix, then using matrix multiplication and addition in each element to generate a new matrix. After the convolution has been processed through an activation function,

sets of pixels are pooled together in a *pooling layer*. The pooling layer takes a large square matrix and through various methods, it generates a smaller square matrix. These pooled layers are then reprocessed through a new convolution layer, and the process repeats until there is a vector with scalar values, which are then used to classify the image into a category. [12] In the image below, a $48x48$ image is broken in to different layers, each possibly representing a color or pattern. The image is broken in to 6 overlapping sub-images of $44x44$. The *Max-pooling* activation takes the largest value of the matrix to pass on to the next layer. Each set of overlapping images is increased in number, and decreased in size to determine more features. The final layers then classify the image based on learned pieces of the image and defined output sets, such as trees or cats.

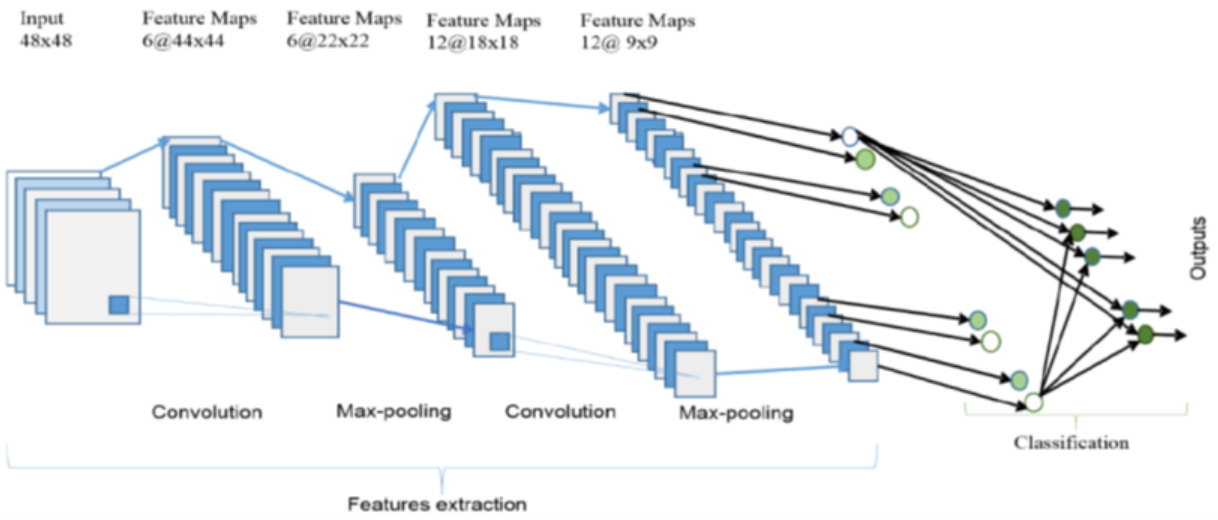

Fig. 2: A sample CNN.

1.3.2 Recursive Neural Network (RNN)

Unlike the CNN, the RNN allows operation over a sequence of vectors in an order. In an RNN, a sequence of tensors X, used for input, is passed through a sequence of activation functions: one weight vector, one activation, and one bias vector used

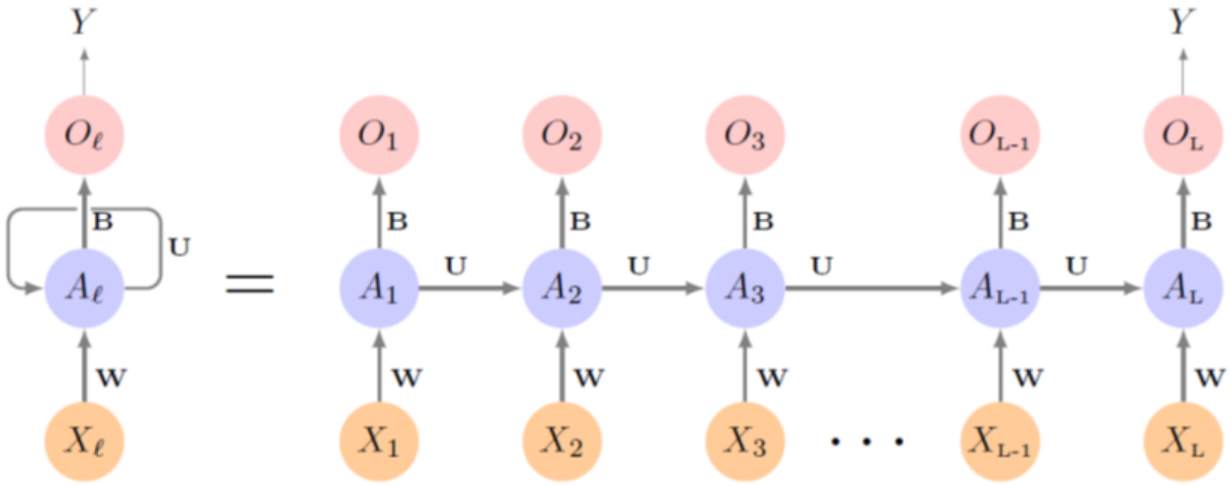

Fig. 3: A sample RNN.

for each element in the X sequence. Since each input is discrete, but the output is a result, there needs to be a shared calculation. This shared calculation is represented by the U matrix. This matrix is a square matrix with dimensions equal to the number of hidden layers K. The purpose of this matrix is to pass on the shared weights from one hidden layer to the next. A sample calculation is below.

$$A_{lk} = A(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{lj} + \sum_{s=1}^{K} u_{ks} A_{l-1,s})$$

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk}$$

[12]

1.3.3 Long Short-Term Memory (LSTM)

The Long Short-Term Memory is a modification to the weight matrix. This is a structure added to the weight matrix that will either keep or update weights as the sequence progresses. The LSTM uses an *input gate*, which protects the weights

against information from less useful inputs, and an *output gate*, which protects against information that is not useful but stored in the memory cell. [10], In the algorithm below, each of the notations for unique to the algorithm and not shared by other pieces of a larger network.

Algorithm 1 LSTM

$f_t = \sigma(W_f\,[h_{t-1}, x_t] + b_f)$
$i_t = \sigma(W_i\,[h_{t-1}, x_t] + b_i)$
$\tilde{C}_t = tanh(W_C\,[h_{C-1}, x_t] + b_C)$
$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
$O_t = \sigma(W_O\,[h_{t-1}, x_t] + b_O)$
$h_t = O_t * tanh(C_t)$

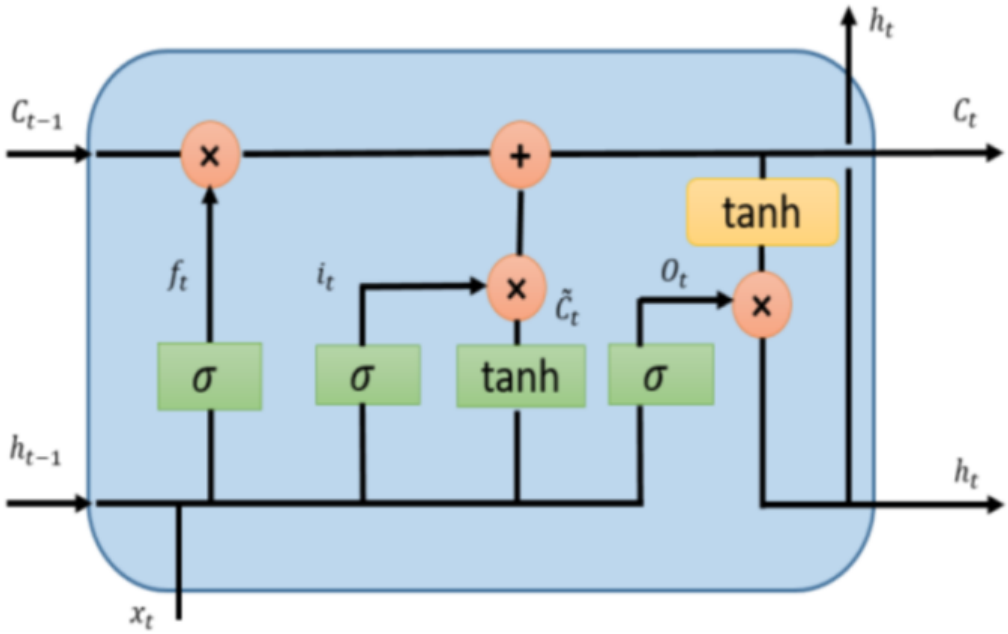

Fig. 4: A Diagram of a LSTM Memory Cell [1].

1.3.4 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit is similar to the LSTM structure. In the LSTM structure, there is a limit to the amount of the memory cell is available, which is controlled by the output gate. In the GRU structure, the entire memory cell is available for

updates and usages. The activation h_t^j of the GRU at a time is a linear interpolation between the previous activation of the unit h_{t-1}^j and the candidate activation $\widetilde{h}_t^j$. An update gate z_t^j decides how much the unit updates its content. Between the two

Algorithm 2 GRU

$h_t^j = \left(1 - z_t^j\right) h_{t-1}^j + z_t^j \widetilde{h}_t^j$
$z_t^j = \sigma \left(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1}\right)^j$

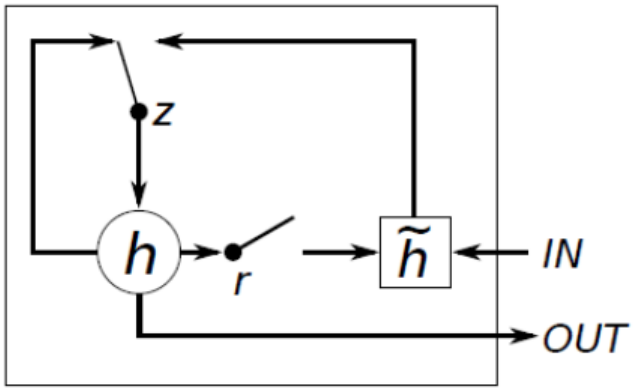

Fig. 5: A Diagram of a GRU Memory Cell [4].

types of memory units, there is no clear advantage from using one or the other. By construction of the two types of memory units, the GRU uses fewer computations and would be able to process faster, but the LSTM would be more stable since there are more calculations to safeguard the weight matrix. Both types of memory units show increased performance when dealing with time-series data where the usefulness of initial data decreases over time, such as forecasting foreign exchange prices or language modeling.

1.3.5 Bi-Directional Recursive Neural Networks (Bi-RNN)

Research to improve the structure of the RNN stretches back to the late 1980s. Research with regard to speech recognition, specifically having computers recognize spoken letters ending with an 'ee' sound, like the letters *b*, *g*, and *d*. One type of

improvement was a Time Delay Neural Network (TDNN). In a TDNN, there is a sequence of inputs connected to a single weight. Using this type of structure allows for the relation and comparison of current inputs to previous inputs [20]. However, the amount of delay is task dependent, and the optimal amount of delay is often found by trial and error. This trial and error uses more computing power, since the same task needs to be completed for each possible number of delays. One way of overcoming this limitation is considering all available input information in both the future and the past of a current time point. Each direction would have its own shared weights and biases. This structure is a Bi-Directional Recurrent Neural Network (Bi-RNN). The structure of a Bi-RNN is similar to a RNN, but the neuron, a single hidden part of the calculation, is split into a part for a positive time-direction and a part for a negative time-direction. The outputs of each forward and backward neuron are not connected [19].

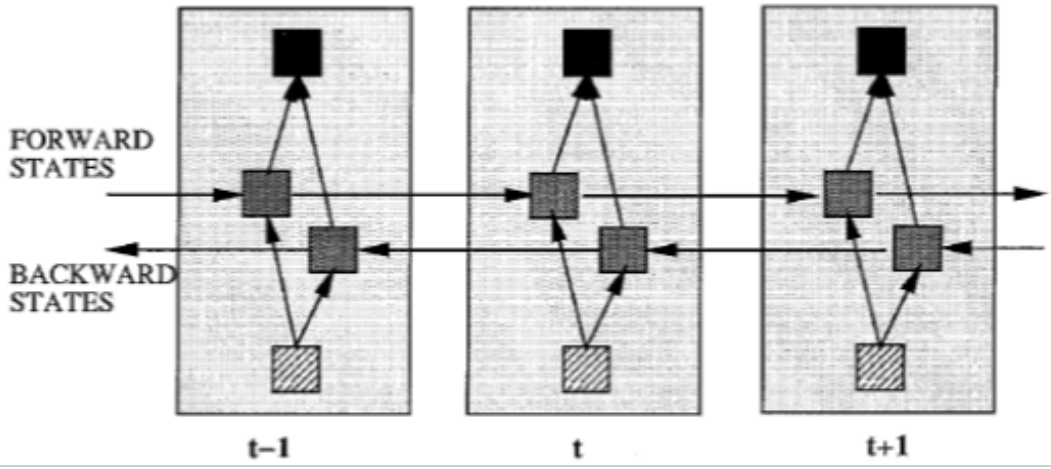

Fig. 6: Structure of the Bidirectional Neural Network (BiRNN) shown for 3 time steps

1.4 Activation Functions

The activation function is based on a similar process in the brain. In the brain, electrical signals are passed from neuron to neuron. If a signal is not intense enough, the

next neuron will not fire, or pass on the signal, and the signal dissipates. The activation function takes each linear combination as constructed above and determines a new value to pass on to the next part of the calculation [14]. Activation functions have been used and different structures have been researched since the advent of the field. There are some key features that need to be considered for an activation function. The function should add a non-linear curvature in the optimization landscape to improve training convergence. The function should not increase the computational complexity of the model. The function should not interfere with gradient flow during training. Lastly, the function should not interfere with the distribution of the data to improve the training of the network [7]. The most common of the activation functions are the Sigmoid, Hyperbolic Tangent, and the Rectified Linear Unit (ReLU). After the research of Nair and Hinton with regard to ReLU, there are a number of variants of this function with respect to negative real numbers. Some of the variants are the Leaky Linear Unit, the Exponential Linear Unit, and the Gaussian Error Linear Unit.

1.4.1 Sigmoid

The sigmoid activation is defined as

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

The sigmoid has a domain of all real numbers and a range of $[0, 1]$. The sigmoid function is one of the most used activation functions. By operating on the sigmoid function, there are multiple variants. The activation function known as swish is defined by

$$A(x) = x \bullet sigmoid(x) = \frac{x}{1 + e^{-x}}.$$

This activation function is smooth and is continuously differentiable. Another variant is known as logish, which is defined by

$$A(x) = x \bullet \ln(1 + sigmoid(x)) = x \bullet ln\left(1 + \frac{x}{1 + e^{-x}}\right)$$

[21].

1.4.2 Hyperbolic Tangent

The Hyperbolic Tangent Function is similar to the Sigmoid Function. The Tanh function has a domain of all real numbers, but the range is $[-1, 1]$. A variant of the Tanh is known as Mish. Mish is defined as

$$A(x) = x \bullet tanh(ln(1 + e^{x}))$$

[21].

1.4.3 Rectified Linear Unit

The Rectified Linear Unit is a Piecewise Function.

$$A(x) = \begin{cases} 0; x \leq 0 \\ x; x > 0 \end{cases}$$

The domain would be all real numbers, and the range would be $[0,\ \infty]$ [16]. The Leaky Linear Unit (LReLU) is based on ReLU and allows for a small, non-zero gradient. This allowance alleviates potential problems caused by the hard 0 of the ReLU. LReLU is defined by

$$A(x) = \begin{cases} x; x > 0 \\ \frac{x}{\alpha}; x \leq 0 \end{cases}$$

where α is a chosen, constant parameter. [15]. If α is allowed to be a learned parameter vector instead of a fixed value, the LReLU becomes a Parametric Rectified Linear Unit (PrReLU) [8]. The Exponential Linear Unit (ELU) is based on ReLU, but the function allows for negative output values, which allows for the mean of the activations to be closer to zero. This then allows for faster learning on the training data. The ELU is defined as

$$A\left(x\right) = \begin{cases} x;\ x > 0 \\ \alpha(\exp\left(x\right) - 1); x \leq 0 \end{cases}$$

where α is a hyperparameter that controls the value to which the ELU saturates for negative inputs [5]. A further modification to the function value for negative inputs yields a different activation function known as DLU. The DLU function is defined as

$$A\left(x\right) = \begin{cases} x;\ x > 0 \\ \frac{x}{1-x}; x \leq 0 \end{cases}$$

which is monotonic [13]. The Gaussian Error Linear Unit (GELU) is based on the cumulative distribution function of the standard normal distribution. The GELU is defined as

$$A\left(x\right) = xP\left(X \leq x\ \right) = x\Phi\left(x\right) = x \cdot \frac{1}{2}(1 + erf\left(\frac{x}{\sqrt{2}}\right))$$

but, this can be approximated by

$$A(x) \approx 0.5x\left(1 + tanh\left(\sqrt{\frac{2}{\pi}}\right)\left(x + 0.044715x^3\right)\right))$$

[9].

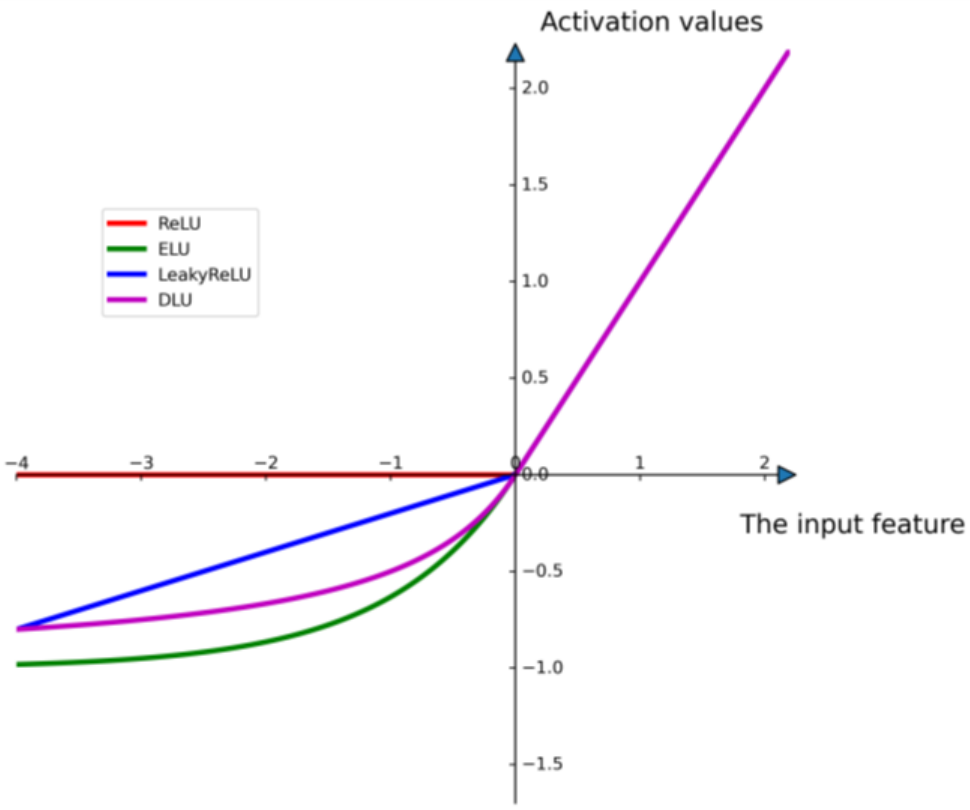

Fig. 7: ReLU, LeReLU(slope = 0.2), ELU(alpha=1) and DLU

The above information is used for one run, or epoch, of the testing data. Before today's advanced computing power, each weight and bias would have to be tuned by hand and another epoch would have to be run. In a neural network, the various adjustments takes time and requires each pathway to be analyzed. Starting in the 1990s and continuing today, means of automating the process of optimizing the neural network is being performed and further analyzed. With the automation of the optimization process allowed the networks to become larger and more connected and complex. The process of optimizing a neural network has 2 key parts, backpropagation and gradient descent.

2.1 Gradient Decent

A key requirement of a neural network is that the structure learns various weights and biases that are applied to the fixed input vector. The way that the structure learns is through gradient descent. Gradient descent is an algorithm that is used for finding local minima of a function. The idea of gradients is from calculus and is well known. A requirement for gradients is a differentiable function almost everywhere. The use of a gradient is to take a step in the direction of the negative of the gradient of the function at the given point. The updated parameter vector would be a linear combination of the old parameter, minus a vector that is the direction of the gradient. The reason for the negative of the gradient is that the gradient points in the direction of the steepest ascent. In the algorithm, θ refers to a vector of weights and biases.

2.2 Stochastic Gradient Descent

In a large neural network, calculating the gradient of each pathway each of the calculations take from input to output takes time and resources. Research was performed,

Algorithm 3 Gradient Descent [1]

Inputs: Loss function ϵ, learning rate η, dataset inputs and outputs X, y, and model F
Outputs: Optimum value that minimizes the loss function
Repeat until convergence:
$\widetilde{y} = F(\theta,\ x)$
$\theta = \theta - \ \eta \bullet \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \epsilon(y, \widetilde{y})}{\partial \theta}$

and each gradient does not need to be calculated for the network to be optimized to the test data. Finding the gradient of fewer, random pathways provides a suitable optimization. This process of finding the gradient descents of the random pathways is referred to as stochastic gradient descent.

Algorithm 4 AStochastic Gradient Descent (SGD)

Inputs: Loss function ϵ, learning rate η, dataset X, y, and model F
Outputs: Optimum value θ that minimizes the loss function ϵ
Repeat until convergence:
Shuffle X, y
For each batch of (X_i, y_i) in X, y do
$\widetilde{y_i} = \ \mathcal{F}(\theta, x_i)$;
$\theta = \theta - \eta \ \bullet \frac{1}{N} \ \sum_{i=1}^{N} \frac{\partial \varepsilon(y_i, \widetilde{y_i})}{\partial \theta}$

2.3 Backpropigation

Backpropagation is the process of using the error between the calculated output and the expected, given, output and assigning that error through the hidden layers. Since each layer was constructed as a functions of a linear combinations, derivatives would then require repeated application of the chain rule. There are two key parts of the backpropagation calculation, calculating the change in the weight amount and the change in the bias amount. Calculating the change in the weight amount starts with

the hidden layer directly before the output layer $l-1$.

$$\delta_j^{l-1} = \frac{\partial \epsilon}{\partial \sigma_j^{l-1}} = \Sigma_{j'} \frac{\partial \epsilon}{\partial \sigma_{i'}^{l}} \frac{\partial \sigma_{j'}^{l}}{\partial \sigma_j^{(l-1)}}.$$

Since each of the σ's are linear transformations of data,

$$\sigma_{j'}^{(l)} = \sum_{i=1}^{N} w_i X_i^{(l-1)} + b_{j'} = \sum_{i=1}^{N} w_i A^{(l-1)} \left(\sigma_{j'}^{(l-1)} \right) + b_{j'}$$

Therefore,

$$\begin{aligned} \delta_j^{l-1} &= \left. \frac{dA^{(l-1)}}{d\sigma} \right|_{\sigma_j^{(l-1)}} \sum_{j'} \frac{\partial J}{\partial \sigma_{j'}^{(l)}} w_j^{(l)} \\ &= \left. \frac{dA^{(l-1)}}{d\sigma} \right|_{\sigma_j^{(l-1)}} \sum_{j'} \delta_j^{l} w_j^{(l)}. \end{aligned}$$

Isolating the effect to each weight,

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}} = \frac{\partial J}{\partial \sigma_{j'}^{(l)}} \frac{\partial \sigma_{j'}^{(l)}}{\partial w_{j,j'}^{(l)}} = \delta_{j'}^{(l)} A_j^{(l-1)}.$$

This last equation allows the calculation of various δ in a layer provided the δ is known to the right. As each δ is calculated, it allows for a new δ to be calculated closer to the input layer. To calculate the affect of the loss function with respect to the bias term,

$$\frac{\partial J}{\partial b_{j'}^{(l)}} = \delta_{j'}^{(l)}$$

[14]

2.4 Learning Rate

From Algorithms 1 and 2, the learning rate defines how large of a step to take when computing the next choices of parameters based on the gradient descent. If the

Algorithm 5 Backpropigation

Input: A network with i layers, the activation function A_l , the outputs of hidden layer $h_l = A_l\left(W_l^T h_{l-1} + b_l\right)$ and the network output $\widetilde{y} = \ h_l$

Compute the gradient: $\delta \leftarrow \frac{\partial \varepsilon(yi, \widetilde{yi})}{\partial y}$

For $i \leftarrow l$ to 0 do: Calculate gradient for present layer:

$$\frac{\partial \varepsilon\left(y, \widetilde{y}\right)}{\partial W_l} = \frac{\partial \varepsilon\left(y, \widetilde{y}\right)}{\partial h_l} \ \frac{\partial h_l}{\partial W_l} = \ \delta \frac{\partial h_l}{\partial W_l}$$

$$\frac{\partial \varepsilon\left(y, \widetilde{y}\right)}{\partial b_l} = \frac{\partial \varepsilon\left(y, \widetilde{y}\right)}{\partial h_l} \ \frac{\partial h_l}{\partial b_l} = \ \delta \frac{\partial h_l}{\partial b_l}$$

Apply gradient descent using $\frac{\partial \varepsilon(y, \widetilde{y})}{\partial W_l}$ and $\frac{\partial \varepsilon(y, \widetilde{y})}{\partial b_l}$

Back-propagate gradient to the lower layer

$$\delta \ \leftarrow \ \frac{\partial \varepsilon\left(y, \widetilde{y}\right)}{\partial h_l} \ \frac{\partial h_l}{\partial h_{\{l-1\}}} = \ \delta \ \frac{\partial h_l}{\partial h_{\{l-1\}}}$$

End

learning rate (η) is too large, the network may start to diverge by jumping over various minima. If η is too small, more calculations may need to be performed, wasting time and computing resources. It is possible for a small η to stop in a poor local minima. A common practice is to start with a large η and decrease it over time. Either the learning rate can be adjusted by a constant defined at the start, or use exponential decay to decrease the learning rate incrementally over each epoch $\eta_t = \eta_0 \beta^t$. Where β is the decay rate, usually $\beta = 0.1$, η_t is the learning rate at a step t, and η_0 is the initial learning rate. [1]

3 Interpreting Results

After each epoch of the training set in the Data-Driven Simulation, the network needs to know how well it performed. This calculation is broken into two parts, the loss and the error. The difference between the calculated result through the network and the expected result is known as the loss, and the sum of the losses is known as the error.

There are different loss and error calculations that can be performed based on what the network is trying to accomplish. As stated earlier, the goal of the backpropagation and gradient descent algorithms is to minimize the loss or error.

3.1 Error Calculations

The major calculations for numeric data are Mean Squared Error (MSE)

$$MSE = \frac{1}{n}\sum (Net_{out} - expected)^2$$

Where n is the number of data points and Mean Absolute Error (MAE)

$$MAE = \frac{1}{n}\sum |Net_{out} - expected|.$$

The MSE calculation can be modified by taking the square root, making the error calculation a linear calculation called Root Mean Square Error (RMSE), which can be interpreted easier.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum (Net_{out} - expected)^2}.$$

Neural networks can be used to calculate probabilities used in classifications instead of regressions. The common calculations performed are error rate

$$Error\ Rate = \frac{1}{n}\sum I(Net_{out} \neq expected)$$

In this calculation, I is an indicator variable with values of 1, if $Net_{out} \neq expected$, or 0 if $Net_{out} = expected$. [12]. If the classification problem has 2 results, then a binary

cross-entropy loss (BCEL) function can be used.

$$BCEL = \sum_{i=1}^{k} -expected \log\left(Net_{Out}\right) - (1 - expected) \log\left(1 - Net_{out}\right).$$

If the classification problem has k classes, a general cross-entropy loss (CEL) function can be used.

$$CEL = -\sum_{i=1}^{k} \left(expected_i \log\left(net_{out_i}\right)\right)$$

3.2 Underfit or Overfit

An important criterion of using any neural network, or any statistical method of model fitting, is the overall fit of the data. Given any set of points, it is possible to find a model that perfectly fits all of the points. This can be completed using an interpolating polynomial. If there are 100 points in the data set, it is possible to create a 100-degree polynomial that will perfectly represent the points. However, any inference drawn would be wildly incorrect. If the data was divided in to 2 sets, a training and a testing set, the errors in the test set would be large, because the curve was designed to fit the test points exactly. This is an example of overfitting. Relating this to neural networks, the network could be set to perfectly represent the testing data, either through a large number of epochs or setting an extremely small error tolerance. Just like with the interpolating polynomial example, the test set error makes the network invalid. When training the network, this idea of overfitting needs to be included. The idea of the network is to generalize to unknown data, which is different from the testing data. It is also important to avoid underfitting the model. An underfit occurs when the model does not sufficiently generalize parameters from the testing data. This could occur from using a small number of epochs or setting a tolerance that is too large. The test set error would also be large for this reason. Using statistical reasoning, an error rate below 5% would be acceptable. This idea

keeps in mind the fact that there are outliers in every data set. It was considered a breakthrough and the start of a greater revival with AlexNet and the ImageNet challenge in which AlexNet exhibited an error rate of 16.4%. [1].

4 Informed Neural Networks (INN)

The above error calculations would be used for known results; examples include curve fitting, or image classification. NNs can be used for parameter estimation, in which the output of the NN would be unknown internal parameters. NNs can be used for further extrapolation of the data set as well. These are done by using loss functions that include the model in question. Variations include Physics Informed Neural Network (PINN), or Biologically Informed Neural Networks. Most of the research cites PINN, though the idea can be modified for any use or situation. The main idea of PINN is encoding a specific ODE or PDE [17], or laws of physics that act as constraints on the output of the original NN to decrease the range of admissible solutions [18]. The use of PINNs may have been limited to a Dense NN. This research uses PINNs, encoding a specific ODE in to the loss function of a NN, in conjunction with RNN structures, specifically GRU and LSTM cells.

5 Why Informed Neural Netowrks

The motivation for using an NN to find the parameters instead of using the NN to make a curve for the solution is the amount of data given. If the NN is not given a complete data set, for example the first or last observations omitted, then the NN would be used to generate a solution curve will be inaccurate.

In a Motivating Experiment, a Curve-fitting NN was given incomplete data. The first set is missing the last 5 observations and the second set is missing the first 5.

As seen in figures 8 and 9, both of the curves fit the given data very well. But,

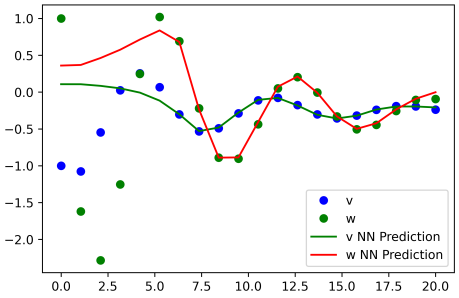

Fig. 8: FitzHugh-Nagumo - Missing first 5 data points

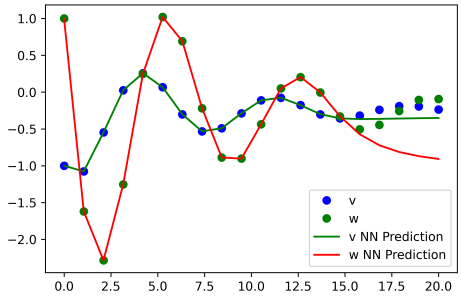

Fig. 9: FitzHugh-Nagumo - Missing last 5 data points

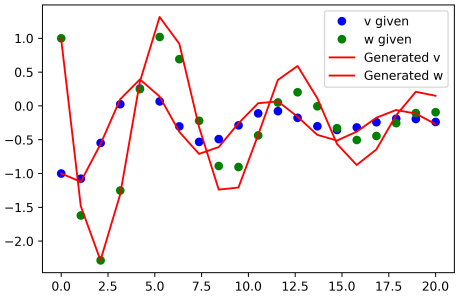

Fig. 10: FitzHugh-Nagumo - Found Parameters - Missing first 5 data points

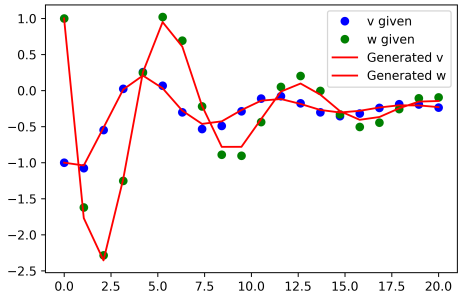

Fig. 11: FitzHugh-Nagumo - Found Parameters - Missing last 5 data points

where the data was missing, the approximation was inaccurate. In the NN with the beginning missing, the curve does not approximate the shape of the curve well. However, an Informed Neural Network found approximations for the missing parameters and generate an accurate graph.

6 FitzHugh-Nagumo Equations

The FitzHugh-Nagumo equations are used in computational neuroscience and used as a model for the potential in a nerve. The equations are commonly modelled as a system of ODEs.

$$\frac{du}{dt} = \frac{1}{c}(u - \frac{u^3}{3} - v)$$

$$\frac{dv}{dt} = c\,(u - av + b)$$

Where $a,\ b,\ c$ are parameters to be found where $0 < a < 1$, $b > 0$ and $c > 0$ and u, v are the solutions of the ODE [11]. In biological context, u represents a short, nonlinear elevation of membrane voltage, and v represents a recovery variable. In most works, one of the parameters is assumed known and the other two are to be solved for. In this research, none of the parameters are known.

6.1 Data

In this exploration, and in future explorations in this paper, the data points were generated using the given systems of ODEs with given parameters. An ODE solver package was called in Python to generate solutions over equally spaced time steps. The data points were generated over 21 equally spaced intervals from 0 to 20.

6.2 NN Structure

The general structure of the network is two-part. In the first part, a Dense NN, with 6 hidden layers and 80 nodes in each layer. is used to find a regression line to model the solution curve. This NN uses the time steps as inputs and 2 outputs, one corresponding to each variable. The error between the NN outputs and the points from the dataset is the MSE. The activation function for each of the hidden layers is the Tanh Function and the activation layer for the output layer is linear with no bias. Numerical differentiation is used to find the numerical derivatives centered at

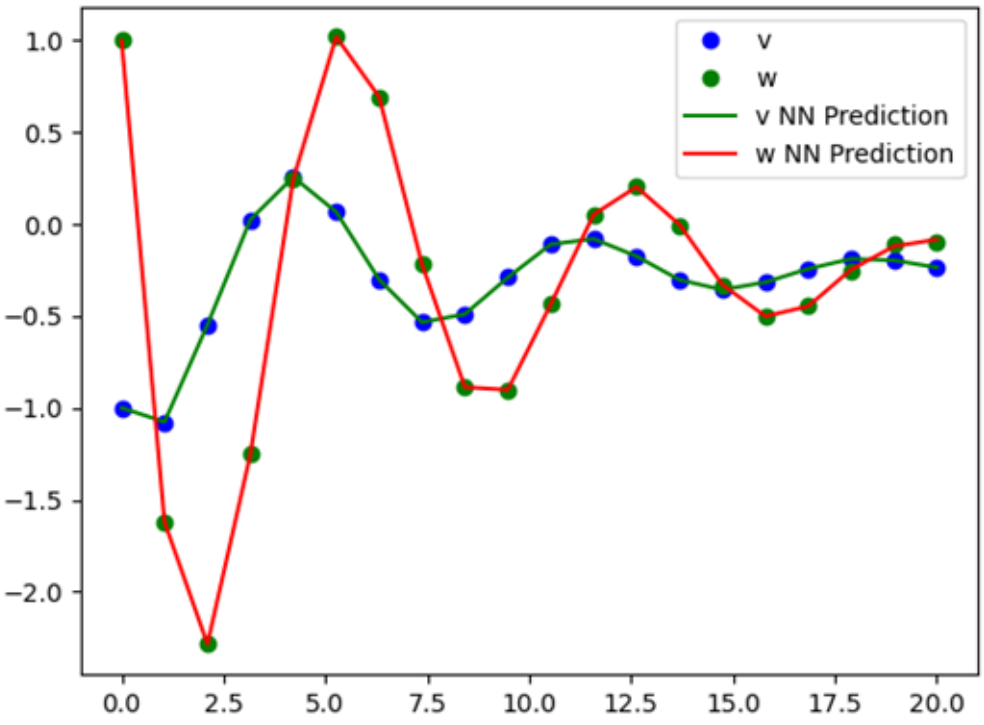

Fig. 12: NN Regression Line

each of the points. A second neural network implements the PINN procedure to find the parameters. The loss function uses a least-squares approach with the following structure.

$$SE = \left(\frac{du}{dt}_{NN} - \frac{1}{c}\left(u - \frac{{u_{NN}}^3}{3} - v_{NN}\right)\right)^2 + \left(\frac{dv}{dt}_{NN} - c\left(u_{NN} - av_{NN} + b\right)\right)^2$$

Where the derivatives are the numerical derivatives calculated earlier and $a, b,$ and c are the parameters, which are the 3 outputs from the NN. The outputs were calculated through a linear transformation with no bias.

6.3 Results

Presented in the table is a summary of the NN structures used, the estimated parameters, average computing time per epoch, and the MSE between the solutions found using the estimated parameters and the given parameters from the dataset. The *layers* column represents the number of hidden layers and the *nodes* column represents the number of nodes in each hidden layer. In the rows with *0*, were direct output with no hidden layers.

Type	Layers	Nodes	Epochs	A=0.2	B=0.2	C=3	Time/epoch	Error
Dense	1	1000	25000	0.202	0.827	-0.566	8ms	66180749125
Dense	40	1	25000	-0.125	0.471	-1.262	23ms	0.954679828
Dense	40	3	50000	0.196	0.189	3.073	22ms	0.000771507
Dense	0	0	25000	-0.131	0.478	0.110	4ms	3.659005714
Dense	8	4	25000	0.199	0.193	3.071	7ms	0.000568776
GRU	2	125	50000	0.198	0.191	3.070	83ms	0.00054922
GRU	40	1	50000	0.197	0.189	3.064	2s	0.00065181
GRU	0	0	25000	0.198	0.193	3.068	69ms	0.000426996
GRU	1	1000	50000	0.197	0.192	3.073	79ms	0.000528024
GRU	8	4	50000	0.196	0.190	3.070	435ms	0.000365781
LSTM	40	3	50000	0.198	0.192	3.066	1s	0.000482801
LSTM	0	0	50000	-0.124	0.469	-1.258	56ms	0.955921563
LSTM	1	1000	50000	0.198	0.191	3.071	87ms	0.00055551
LSTM	2	125	50000	0.199	0.191	3.081	152ms	0.000840055
LSTM	8	4	50000	0.200	0.190	3.073	92ms	0.000810989

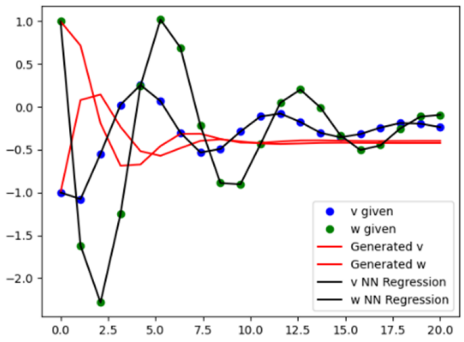

Fig. 13: Solutions using inaccurate parameters

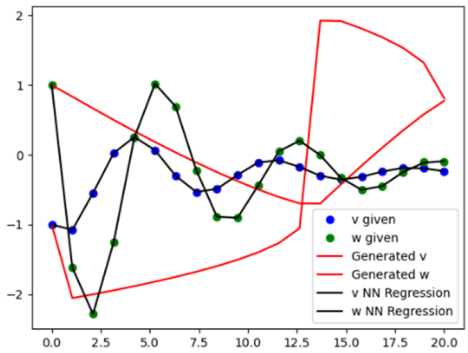

Fig. 14: Solutions using inaccurate parameters

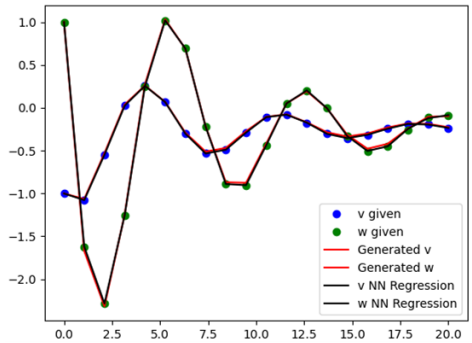

Fig. 15: Solution generated using LSTM and 1 hidden layer with 1000 nodes

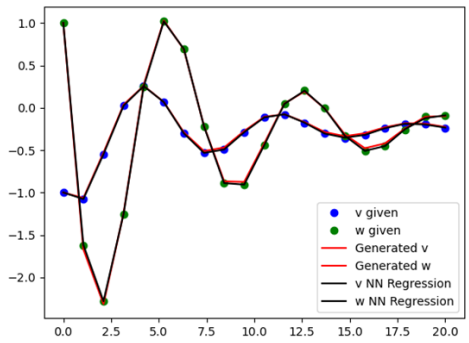

Fig. 16: Solution Generated using GRU and 2 hidden layers with 125 nodes

6.4 Conclusion

From the results, the GRU structure was the most stable using any structure. When calculating the output directly, the GRU structure was able to provide reasonable estimates where the LSTM and Dense structures were very inaccurate. Both the GRU and LSTM structures were stable when either more layers, or more nodes per layer were added.

An avenue for future research and experimentation would be to introduce noise into the dataset. The final accuracy of the calculations would depend on the accuracy of the regression line calculated through the first NN.

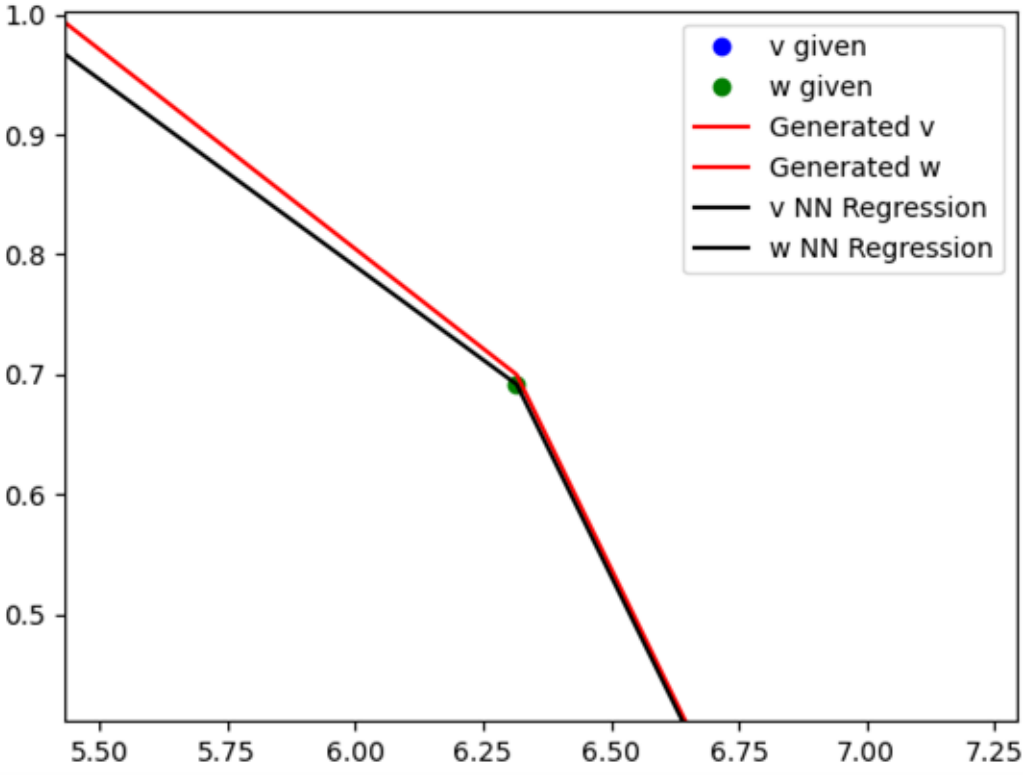

Fig. 17: Sample of difference between given and estimated parameters

7 Lotka-Volterra Model

The Lotka-Volterra Model is a two-population model that uses different dynamic features as equilibrium points as well as limit cycles. This model is named for Alfred James Lotka and Vito Volterra. The system of ODEs is:

$$\frac{dN}{Dt} = N\left(a - bP\right)$$

$$\frac{dP}{dt} = P\left(cN - d\right)$$

Where $a,\ b,\ c,\ d$ are non-negative parameters, where a is the growth rate of the prey, b is the death rate of the prey, c is the growth rate of predators per unit of prey, and d is the death rate of the predators. N represents the size of the prey population and P represents the predator population [2]. Simulated data was generated for 100 time steps in $t \in [0, 13], a = 1,\ b = 2,\ c = 1,\ d = 0.3$.

7.1 NN structure

The composition of the network used is similar to the structure for the FitzHugh-Nagumo equation network, except using 4 output nodes instead of 3.

7.2 Results

Presented in the table is a summary of the NN structures used, the estimated parameters, average computing time per epoch, and the MSE between the solutions found using the estimated parameters and the given parameters from the dataset.

Type	Layers	Nodes	Epochs	A=1	B=2	C=1	D=0.3	Time/epoch	Error
LSTM	0	4	10000	0.9933	1.979	0.9908	0.2993	30ms	2.7122×10^{-5}
LSTM	1	20	10000	1.0014	2.0154	1.0165	0.29997	42ms	5.0959×10^{-5}
LSTM	5	20	10000	0.979	1.9825	0.9843	0.3022	91ms	2.1464×10^{-5}
LSTM	1	200	10000	1.0038	1.9856	1.0072	0.2947	45ms	9.2868×10^{-5}
GRU	0	4	10000	0.9887	1.9672	1.0013	0.299	36ms	2.0832×10^{-5}
GRU	1	20	10000	0.9859	1.9887	1.0028	0.2952	42ms	8.8220×10^{-5}
GRU	5	20	10000	0.9905	1.9973	1.0047	0.2961	92ms	6.1271×10^{-5}
GRU	1	200	10000	0.9839	1.989	1.0046	0.2953	44ms	1.1295×10^{-5}
Dense	0	4	10000	0.9929	2.0001	0.9961	0.2976	23ms	3.5029×10^{-5}
Dense	1	20	10000	0.9659	1.9505	0.9957	0.2913	23ms	3.0416×10^{-4}
Dense	5	20	10000	0	0	0	0	35ms	0.0556
Dense	1	200	10000	0.9831	2.0038	0	0	23ms	42.0334

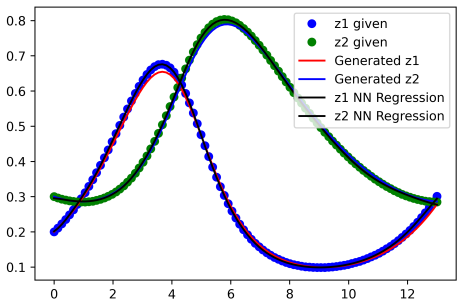

Fig. 18: Solution Curve with GRU nodes and 1 hidden layer

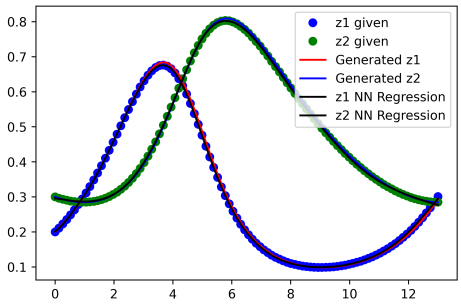

Fig. 19: Solution Curve with LSTM nodes

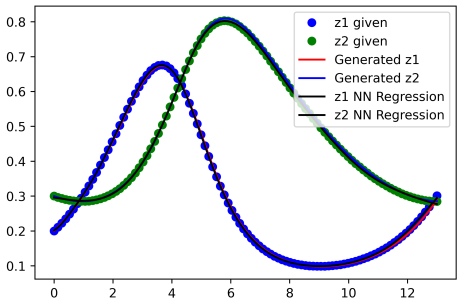

Fig. 20: Solution curve using GRU nodes

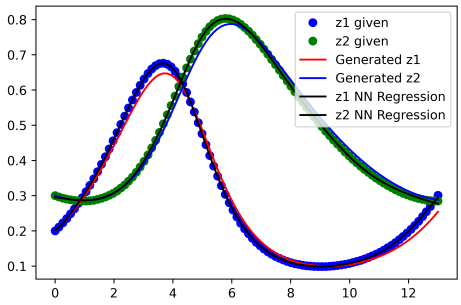

Fig. 21: Solution Curve with Dense NN and 1 hidden layer

7.3 Conclusion

Comparing results to Borzi: $a = 0.9163, b = 1.8150, c = 0.9847, d = 0.2936$ with a dense structure and 50000 epochs, the RNN structure provides for more accuracy and less processing time since there are fewer than 90000 epochs used.

8 Bellman's Problem

The following problem deals with reversible homogenous gas-phase reaction kinetics, which is also known as Bellman's problem. This problem models the following chemical reaction:

$$2NO + O_2 \underset{\leftarrow}{\overset{k_1}{\rightarrow}} 2NO_2.$$

The corresponding mathematical ODE is given by:

$$\frac{dz}{dt} = \theta_1 (126.2 - z)(91.9 - z)^2 - \theta_2 z^2$$

where $z_0 = 0$ and $t \in [0, 50]$. Simulated data was generated using $k_1 = \theta_1 = 4.57 \times 10^{-6}$ and $k_2 = \theta_2 = 2.7854 \times 10^{-4}$. [6]

8.1 Structure and Results

A similar structure from the preceding problems was utilized. In this problem, additional information was generated by introducing a stopping condition. The stopping condition was monitored by a lack of decrease in the loss function based on the ODE. These results are located at the bottom of the table.

Type	Hidden Layers	Nodes	epochs	time /epoch	k_1	k_2	MSE
Generated					4.57×10^{-6}	2.7845×10^{-4}	
GRU	0	4	10000	30ms	4.53E-06	0.000279527	0.016235703
GRU	1	20	10000	53ms	4.44E-06	0.000279518	0.181050759
GRU	5	3	10000	100ms	4.65E-06	0.00027948	0.059932683
LSTM	5	3	10000	115ms	4.49E-06	0.000254279	0.063716425
LSTM	1	20	10000	60ms	4.53E-06	0.000273859	0.006435396
LSTM	0	4	10000	38ms	4.50E-06	0.000246087	0.152757706
LSTM	1	20	12136	40ms	4.58E-06	0.000279474	0.001126257
GRU	0	4	4816	30ms	4.61E-06	0.000279431	0.012395682
LSTM	5	3	10987	90ms	4.57E-06	0.00027752	0.00018013
GRU	5	3	5125	82ms	4.50E-06	0.000279481	0.049137759
GRU	1	20	5494	40ms	4.68E-06	0.000279469	0.115510055
LSTM	0	4	12103	35ms	4.58E-06	0.000278438	0.000307594
Given in [6]					4.56E-06	2.77E-04	8.65E-04

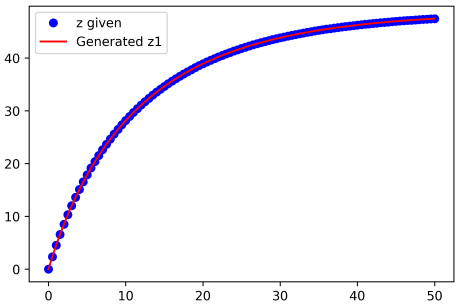

Fig. 22: Bellman's Problem with LSTM cells, 5 hidden layers, 3 cells in each layer, and stopping condition

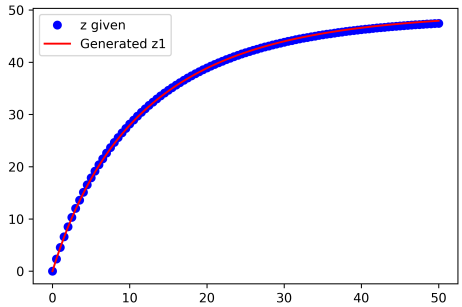

Fig. 23: Bellman's Problem with LSTM cells, 5 hidden layers, 3 cells in each layer, and no stopping condition

8.2 Conclusion

In this problem, the LSTM cells outperformed the GRU cells in accuracy when the stopping condition was added, although this added more processing time due to more epochs needed. When the network stopped at a common 10000 epochs, the GRU networks had better results.

9 Conclusion

Throughout the previous experiments using RNNs combined with PINN, the end goal was not a fitted curve, but finding the parameters. By finding the parameters, the network can then use the given ODE and be able to account for the missing data to produce an accurate solution curve. The parameters also represent some relation between variables, such as reaction rates or predator/prey death rates. By finding these parameters, more information about the systems of ODEs that generated the data set can be inferred.

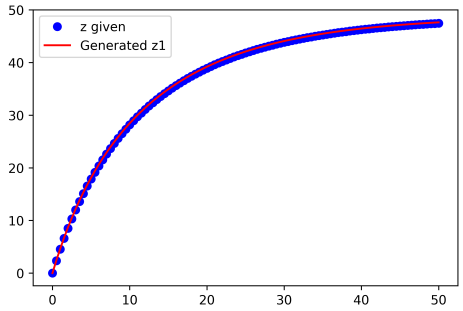

Fig. 24: Bellman's Problem with GRU cells, 5 hidden layers, 3 cells in each layer, and no stopping condition

References

[1] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.

[2] Alfio Borzì. *Modelling with ordinary differential equations: a comprehensive approach*. CRC Press, 2020.

[3] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[4] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[6] Vivek Dua. An artificial neural network approximation based decomposition approach for parameter estimation of system of ordinary differential equations. *Computers & chemical engineering*, 35(3):545–553, 2011.

[7] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 2022.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[9] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

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

[11] Mark H Holmes. *Introduction to scientific computing and data analysis*, volume 13. Springer Nature, 2023.

[12] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, et al. *An introduction to statistical learning*, volume 112. Springer, 2013.

[13] Jianfei Li, Han Feng, and Ding-Xuan Zhou. A new activation for neural networks and its approximation. *arXiv preprint arXiv:2210.10264*, 2022.

[14] Sébastien Lleo. Machine learning: An applied mathematics introduction: by paul wilmott, panda ohana publishing,(2019). paperback. isbn 978-1916081604., 2020.

[15] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.

[16] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[17] Guofei Pang, Lu Lu, and George Em Karniadakis. fpinns: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing*, 41(4):A2603–A2626, 2019.

[18] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[19] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[20] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[21] Hegui Zhu, Huimin Zeng, Jinhai Liu, and Xiangde Zhang. Logish: A new nonlinear nonmonotonic activation function for convolutional neural network. *Neurocomputing*, 458:490–499, 2021.

All programs used in this thesis are available at
https://github.com/Ronald-Balint/Thesis