Solving and Discovering Partial Differential Equations via Deep Learning Models

October 10, 2021

Abstract

We review two deep learning models that concern the dynamics underlying sparse and/or noisy spatiotemporal data appearing in [9, 12, 13]. The first model aims to infer the solution to a known partial differential equation of the form $u_t = \mathcal{N}(u; \lambda)$ from space and time data. The second model aims to identify the PDE itself. We apply these models to Burger's Equation to recreate published results and to the 1-dimensional Swift-Hohenberg equation, which has not appeared in associated literature.

1 Introduction

In recent years, there has been a growing interest in using neural networks to solve nonlinear partial differential equations from noisy or sparse observations taken from a spatiotemporal domain. Even more novel is the discovery of the PDE itself from these measurements. In a series of papers beginning in 2017, Maziar Raissi et. al. provide a way to identify a PDE from gathered data and then solve this PDE to forecast future states of the system. Broadly speaking, if the PDE is of the form $u_t = \mathcal{N}(u; \lambda)$ with solution $u(t, x)$, these neural networks try to produce a model $\hat{u}$ such that

$$\hat{u}(t, x) = u(t, x) \quad \text{and} \quad \hat{u}_t = \mathcal{N}(\hat{u}; \lambda).$$

This paper is outlined as follows. In Section 2 we review basic information concerning deep neural networks. In Section 3 we discuss the two learning models of interest in detail and in Section 4 we recreate some published results and apply the learning models to a new PDE. Lastly, Appendix A contains figures and tables from the explorations.

2 Deep Feed-forward Neural Networks

We first review the structure of a single layer feed-forward neural network [5]. If we have K responses there will be K output units, denoted Y_k with $k = 1, \ldots, K$. Derived features Z_m are created via linear combinations of M inputs and then the target Y_k is modeled as a function of linear combinations of the Z_m:

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \qquad m = 1, \ldots, M \\ T_k &= \beta_{0k} + \beta_k^T Z, \qquad k = 1, \ldots, K \\ f_k(X) &= g_k(T), \qquad k = 1, \ldots, K. \end{aligned}$$

The activation function σ can vary based on the network design, but is often the sigmoid function given via $\sigma(x) = \frac{1}{1+e^{-x}}$, the rectified linear unit (ReLU) given by $\sigma(x) = \max\{0, x\}$, a sinusoid or a hyperbolic tangent. The variables Z_m are referred to as hidden units and belong to the single hidden layer. A deep feed-forward neural network has the same components as a single layer perceptron, only with more hidden layers [4].

2.1 Network Architecture

One of the main considerations when constructing a neural network are the number of hidden layers, referred to as the depth, and how many hidden units to include in each layer, which is called the width.

The other considerations are the type of hidden units, how to connect each layer, the choice of activation function σ and the output function g_k which allows for a final transformation of the hidden layer outputs.

2.1.1 Universal Approximators and Network Depth

While neural networks can have multiple hidden layers, it is possible to represent any measurable function via a single layer perceptron with sufficient width. This is called the Universal Approximation Theorem and for this reason, neural networks are sometimes referred to as universal approximators.

Theorem 2.1. [6] *Given any function $f : [-1,1]^d \to \mathbb{R}$ and any $\epsilon \geq 0$ and any non-polynomial activation function σ, there exists a sufficiently large h, such that there is a single layer neural network $\beta\sigma(\alpha\mathbf{Z} + \alpha_\mathbf{0})$, that has activation function σ and h hidden units in the hidden layer such that for all $x \in [-1,1]^d$,*

$$\|f(x) - \beta\sigma(\alpha\mathbf{Z} + \alpha_\mathbf{0})\| \leq \epsilon.$$

In other words, no matter what function we seek to learn, a sufficiently large single layer perceptron is able to represent it. However, this does not mean that the training algorithm will be able to learn this function. Furthermore, the width of the hidden layer may need to be impractically large. For example, the number of parameters can grow exponentially with the number of inputs.

Example 2.2. [4] Suppose we want learn a particular binary function on the set of n dimensional vectors v with entries in $\{0,1\}$. There are 2^n vectors in this set.

Since a binary function f assigns either 0 or 1 to each input vector v, the possible number of binary functions defined on this domain of vectors is 2^{2^n}. Specifying one such function requires 2^n parameters. If n is large, the number of parameters necessary to implement this model would become unfeasibly large.

In many circumstances, using a deeper model can reduce the number of units needed to represent the function of interest and the number of layers is typically chosen via experimentation [4]. In general, it is better to have too many hidden units than too few. This is because if there are too few hidden units, the model might not have enough flexibility to capture nonlinearities in the data but with too many hidden units, the extra weights can be minimized with appropriate regularization. From a heuristic point of view, having multiple layers reflects the belief that the function our model should learn can be represented as the composition of several simpler functions. Alternatively, we can think of having multiple layers as the belief that the function we seek is a procedure consisting of sequential steps where each step using the previous step's output [5].

2.1.2 Other Considerations

The design of the hidden units is an active area of research and there are few definitive guiding theoretical principles [4]. In general, each hidden layer has the same structure as the single hidden layer for the single layer perceptron discussed earlier. The activation function can be the same for each layer or it can vary.

Furthermore, most networks connect these layers in a chain structure, with each layer being a function of its preceding layer. For example, the first layer is often given by

$$\mathbf{T}^{(1)} = \sigma^{(1)}\left(\beta_0^{(1)} + \beta^{T,(1)}\mathbf{X}\right)$$

and the second layer is given by

$$\mathbf{T}^{(2)} = \sigma^{(2)}\left(\beta_0^{(2)} + \beta^{T,(2)}T^{(1)}\right).$$

Every neuron in a given layer may be connected to every neuron in the previous layer or there may be partial connections. Additionally, some neurons may skip a layer and feed directly into the next one. These connectivity choices are often informed by prior beliefs about the model or experimental results [4].

The neural networks of interest in this paper will connect hidden layers by a linear transformation via a weight matrix $\mathbf{W}$, which connects every input to every output. Furthermore, the activation function for each hidden layer will be the same.

2.2 Training the Neural Network

2.2.1 Automatic Differentiation

Recall that when a feed forward neural network takes an input x and produces an output $\hat{f}(x) = \hat{y}$, this is done by flowing information forward through the neural network. For this reason, this is called forward propagation. During training, the value of the loss function is computed after forward propagation, and then the back-propagation algorithm allows this information to flow backward through the network to compute the gradient.

In order to compute the gradient, we must compute derivatives. There are several different ways of doing differentiation within a machine. The first is symbolically, like what is done with Matlab or Mathematica and then the value of interest is plugged in. The second is numerically, which uses finite difference approximations, and the third is a combination of the two, in which the derivative is computed by hand, hard-coded into the program, and then evaluated at points of interest. This is however not a practical strategy for large neural networks for which there may be many terms. Additionally, symbolic and numerical differentiation are computationally expensive.

The last option is automatic differentiation, which tends to be the most efficient. The term automatic differentiation refers to a collection of techniques that compute derivatives by accumulating values during code execution that are then used to obtain numerical derivative evaluations. This allows accurate evaluation of derivatives at machine precision with a small amount of overhead and computational cost [2]. Because automatic differentiation was a new technique learned during the implementation of the deep learning models, we review it here.

Broadly speaking, automatic differentiation has two modes; a forward accumulation mode (tangent linear) and a reverse mode. This discussion focuses on the reverse mode, since it is most useful in the case $f : \mathbb{R}^m \to \mathbb{R}^n$ where $m >> n$ (as is the case for a loss function; $f : \mathbb{R}^n \to \mathbb{R}$).

The first step is to construct a computational graph with the following variables:

$$\begin{aligned}
&\text{Input variables: } v_{i-n} = x_i, \qquad i = 1, \dots, n \\
&\text{Intermediate variables: } v_i, \qquad i = 1, \dots, \ell \\
&\text{Output variables: } v_{\ell-i} = y_{m-i}, \qquad i = m-1, \dots, 0.
\end{aligned}$$

In additional to constructing these variables and nodes, we also construct an adjoint associated to each intermediate variable, given by

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

After running a forward pass where we initialize the nodes, variables and record the dependencies of the nodes. Then, we run a reverse pass to propagate the adjoints in reverse. This backward pass yields all of the partial derivatives.

Example 2.3. Suppose that we want to model $f(x_1, x_2) = x_1 x_2 + \cos(x_1) - e^{x_1}$ as a computational graph. Then,

we would set

$$\begin{aligned} v_{-1} = x_1 \quad &\rightarrow \quad v_1 = \cos(v_{-1}) \\ &\quad\;\; v_2 = v_0 \cdot v_{-1} \\ &\quad\;\; v_3 = v_3 = e^{v_{-1}} \quad \rightarrow \quad y = v_5. \\ &\quad\;\; v_4 = v_1 + v_2 \\ v_0 = x_2 \quad &\rightarrow \quad v_5 = v_4 + v_3 \end{aligned}$$

As a computational graph, this function takes the following form:

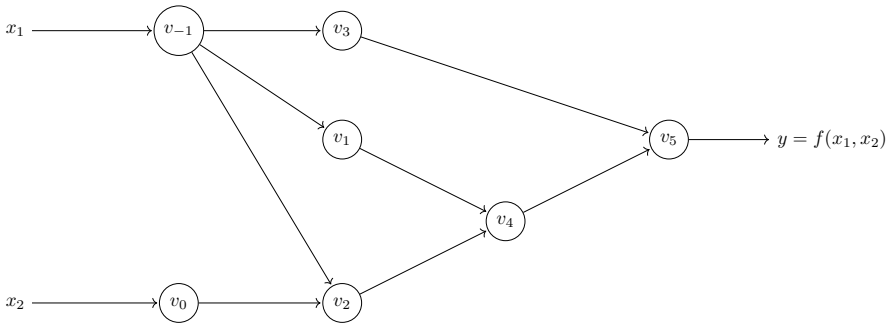


Figure 1: Computational graph of $f(x_1, x_2) = x_1 x_2 + \cos(x_1) - e^{x_1}$

On the backward pass, we can compute the partial derivatives of f with respect to both x_1 and x_2. This is represented by the dashed errors in the below diagram.

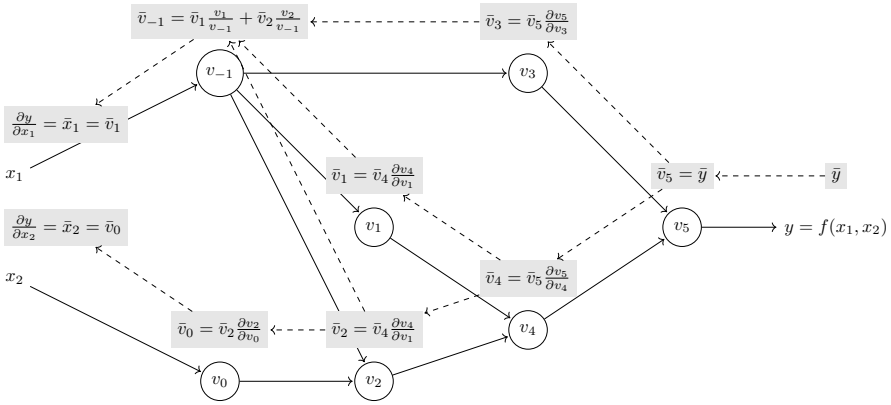


Figure 2: Depiction of computing the partial derivatives with a single backward pass.

Back within the context of neural networks, training a neural network is an optimization problem with respect to its weights. So the back-propagation algorithm is simply a special case of automatic differentiation is which we are evaluating a loss function as in terms of its weights. When constructing the computational graph, we add weights w_i to the edges. Using automatic differentiation, we can easily compute the partial derivatives that are needed to perform the weight updates. The weight updates are typically done using gradient-descent or a higher order method (see Section 2.2.2).

We will use TensorFlow to construct our models because TensorFlow implements automatic differentiation via a computational graph that is defined statically. In other words, the computational graph and partial derivatives are computed before optimization starts, reducing the computational cost. Additionally, the computational graph in TensorFlow is versatile since it works with objects called tensors, which are arrays of arbitrary dimension [1].

2.2.2 Optimization Methods

Finally, we review the optimization methods that are used to update the weights of our neural network during training. In particular, two optimization methods with adaptive learning rates are used; one for unperturbed training data and another for noisy training data. Both methods perform updates by computing derivatives and so rely on the back-propagation algorithm discussed above.

L-BFGS

The Limited-Memory Broyden-Fletch-Goldfarb-Shanno algorithm incorporates the advantages Newton's method, which is a second order method, without the computational cost [4]. If L denotes the loss function, the $k+1$st update to the parameters β is given via

$$\beta^{(k+1)} = \beta^{(k)} - \mathbf{H}^{-1}\Big|_{\beta=\beta^{(k)}} \nabla_\beta L\left(\beta^{(k)}\right),$$

where $\mathbf{H}$ is the Hessian of the loss function L with respect to β. The largest computational cost in Newton's method is computing the inverse Hessian matrix since it could be very large.

The BFGS algorithm approximates the inverse Hessian; making it a quasi-Newton method. More precisely, $\mathbf{H}^{-1}$ is approximated via a matrix $\mathbf{M}_k$ that is iteratively updated to become a better approximation of $\mathbf{H}^{-1}$. Once this matrix $\mathbf{M}_k$ is constructed, the update to β is given by

$$\beta^{(k+1)} = \beta^{(k)} - \eta_k \mathbf{M}_k \nabla_\beta L\left(\beta^{(k)}\right),$$

where the step size η_k is determined by line search in the direction of descent. More precisely, $\mathbf{M}_0$ is some sparse, symmetric and positive definite matrix that approximates the inverse Hessian of L. Then, $\mathbf{M}_{k+1}$ is given by

$$\begin{aligned}
\mathbf{M}_{k+1} &= \mathbf{V}_k^T \mathbf{M}_k \mathbf{V}_k + \rho_k s_k s_k^T \\
\text{where } \rho_k &= \left[\left[\nabla_\beta L\left(\beta^{(k+1)}\right) - \nabla_\beta L\left(\beta^{(k)}\right)\right]^T \left(\beta^{(k+1)} - \beta^{(k)}\right)\right]^{-1} \\
\mathbf{V}_k &= I_N - \rho_k \nabla_\beta \left(L(\beta^{(k+1)}) - L(\beta^{(k)})\right)\left(\beta^{(k+1)} - \beta^{(k)}\right)^T.
\end{aligned}$$

The BFGS algorithm requires a fair amount of memory, since the Hessian matrix for n parameters has n^2 entries, which is impractical for learning models with many parameters.

The L-BFGS circumvents this issue by not storing the complete inverse Hessian approximation $\mathbf{M}_k$. In particular, L-BFGS performs BFGS m times, where m is specified by the user and then updates $\mathbf{M}_0$ using the pairs $\{y_j, s_j\}_{j=1}^m$ where

$$\begin{aligned}
y_j &= \nabla_\beta L\left(\beta^{(j+1)}\right) - \nabla_\beta L\left(\beta^{(j)}\right) \\
s_j &= \beta^{(j+1)} - \beta^{(j)}
\end{aligned}$$

This reduces the amount of memory required to use this algorithm, making it more useful for large learning models [8].

Lastly, we use the variation L-BFGS-B method where the 'B' stands for bounds. This uses the same methodology as above, only the algorithm terminates after a maximum number of iterations or if the difference in loss functions after successive iterations goes below a certain value.

Adam

The Adam algorithm (Adaptive Moment Estimation) is an adaptive learning algorithm that keeps an exponentially decaying average of past squared gradients and an exponentially decay average of past gradients [7]. This

method is well suited for large learning models and handles problems with noisy data well.

More precisely, recall that the kth moment of a random variable is given by $E[x^k]$ and denote the estimate of the first and second moment of the gradients as m_t and v_t respectively. These values will be initialized as vectors of zeros. Because of this choice, v_t and m_t will be biased towards zero at the initial time steps.

In order to counteract these biases, we fix $\lambda_1, \lambda_2 \in \mathbb{R}$ as decay rates and at the kth iteration, compute the bias corrected terms:

$$\hat{m}_t = \frac{m_t}{1-\lambda_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\lambda_2^t}.$$

To see why these should be the unbiased estimates, denote the gradient of the loss function as g and let $g_1, \ldots, g_r$ be the gradients at time steps $g_1, \ldots, g_r$. These will typically be observations taken from an underlying gradient distribution.

Then, the update of the exponential moving average at time step k can be written as

$$\begin{aligned} v_k &= \lambda_2 \cdot v_{k-1} + (1-\lambda_2) \cdot g_k^2 \\ &= (1-\lambda_2) \sum_{i=1}^{k} \lambda_2^{k-i} \cdot g_i^2; \end{aligned}$$

where g_i^2 denotes element-wise square. Then, if we take the expectation,

$$\begin{aligned} E[v_k] &= E\left[(1-\lambda_2) \sum_{i=1}^{k} \lambda_2^{k-i} g_i^2\right] \\ &= E\left[g_k^2\right] \cdot (1-\lambda_2) \sum_{i=1}^{k} \lambda_2^{k-i} + \zeta; \end{aligned}$$

where ζ is a discrepancy term for the second moments in the past

$$= E\left[g_k^2\right] \cdot (1-\lambda_2^k) + \zeta.$$

We want to choose the exponential decay rate λ_1 so that the exponential moving average assigns small weights to gradients too far in the past; effectively keeping ζ small. Additionally since the term $(1-\lambda_2^k)$ is a result of initializing the running average as a vector of 0s, we divide by this term to correct the initialization bias.

The parameters are then updated as

$$\beta_{k+1} = \beta_k - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t,$$

where $\epsilon > 0$ is fixed and small. A recommendation for the fixed values in [7] are

$$\lambda_1 = .9, \quad \lambda_2 = .999, \quad \epsilon = 10^{-8}, \quad \eta = .001.$$

Note that since the effective step size taken in parameter space is $\Delta_k = \eta \hat{m}_k / \sqrt{\hat{v}_k}$, which changes with each iteration, this is an adaptive learning algorithm.

3 Physics-Informed Neural Networks

3.1 Basic Algorithm

Broadly speaking, physics-informed neural networks (PINNs) are a class of deep feed-forward neural networks that can address nonlinear problems without making restrictive prior assumptions. These neural networks will be constrained to respect the principles that determine the observed data; i.e. as governed by a time dependent non-linear partial differential equation.

These neural networks can address two areas of interest; the first is finding a data driven solution to a PDE of the form

$$u_t = \mathcal{N}(u; \lambda), \tag{1}$$

with λ a parameter and the second is data-driven discovery of the PDE itself. In the first scenario, $u(t, x)$ is the function that we want to learn and $N(\cdot; \lambda)$ is a nonlinear operator parameterized by λ with known value. In the second scenario, we still want to learn $u(t, x)$, but now λ may not be known, or the entire function $\mathcal{N}(u; \lambda)$ may not be known either.

While continuous and discrete time models are addressed in [9, 12, 13], we will focus on the continuous case. In all of the following discussing, the data $\{(t, x, u)\}$ is assumed to be the true solution to the PDE, usually with specified boundary conditions. Unless otherwise stated, the training data will be a small randomly sampled subset of $\{(t, x, u)\}$ and the testing data will be the entire set of data.

3.2 Data Driven Solutions

First suppose that the PDE of the form Equation 1 is known. We are first interested in the case that the model parameters λ are known and fixed and we would like to determine what can be said about the solution $u(t, x)$.

We first model u as a neural network with undetermined parameters and then express

$$f(u) = u_t - \mathcal{N}(u; \lambda) \tag{2}$$

in terms of the derivatives of u, which are computed via automatic differentiation (see Section 2.2.1). Suppose our training data consists of a sample of initial and boundary data for the solution u, $\{(t_{u,i}, x_{u,i}, u_i)\}_{i=1}^{N_u}$ and a set of collocation points, $\{(t_{f,i}, x_{f,i})\}_{i=1}^{N_f}$ for which $f = u_t - \mathcal{N}(u)$. If $\hat{u}$ denotes the estimated model, the shared parameters of the two neural networks u and f can then be determined by minimizing the mean squared error:

$$\begin{aligned} MSE &= MSE_u + MSE_f \\ &= \frac{1}{N_u} \sum_{i=1}^{N_u} \|\hat{u}(t_{u,i}, x_{u,i}) - u_i\|^2 + \frac{1}{N_f} \sum_{i=1}^{N_f} \|f(t_{f,i}, x_{f,i})\|^2 \\ &= \frac{1}{N_u} \sum_{i=1}^{N_u} \|\hat{u}(t_{u,i}, x_{u,i}) - u_i\|^2 + \frac{1}{N_f} \sum_{i=1}^{N_f} \|\hat{u}_t(t_{f,i}, x_{f,i}) - \mathcal{N}(\hat{u}(t_{f,i}, x_{f,i}))\|^2. \end{aligned} \tag{3}$$

The MSE is then minimized using the L-BFGS-B optimization algorithm.

The first loss term in Equation 3 corresponds to the initial and boundary data, motivating the neural net to produce a model $\hat{u}$ such that $\hat{u} \approx u$ on these points. The second loss term enforces the structure of the underlying PDE at a finite set of points on the domain. Furthermore, over-fitting is less of a concern because the MSE_f term in the loss function acts as a regularization mechanism that penalizes solutions that do not satisfy the underlying PDE.

If X denotes the set of all N data points, one of the ways we will evaluate the performance of the algorithm is

through the relative error and the absolute error:

$$\text{Abs. Error} = \sum_{i=1}^{N} (u(x_i, t_i) - u_i)^2$$

$$\text{Rel. Error} = \sum_{i=1}^{N} (u(x_i, t_i) - u_i)^2 \left(\sum_{i=1}^{N} u_i^2 \right)^{-1}.$$

Note that since the training data $\mathcal{T}$ is assumed to be a small subset of X, the model is being tested on many previously unseen points.

3.3 Data Driven Discovery

3.3.1 Parameter Discovery

Now suppose that we do not know the parameter λ in Equation 1. Then, we can proceed as before in Section 3.2 by initializing the unknown solution $u(t, x)$ as a neural network and defining the function $f(u)$ as before as well,

$$f(u) = u_t - \mathcal{N}(u; \lambda).$$

Previously, the neural networks u and f had the same unknown parameters. Now, the unknown parameter λ from the PDE is an unknown parameter of the neural network f as well. Now, the shared parameters of the neural networks $u(t, x)$ and $f(u)$ along with the unknown parameters λ of the differential operator are minimized using the mean squared error loss:

$$\begin{aligned} MSE &= MSE_u + MSE_f \\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\|\hat{u}(t_i, x_i) - u(t_i, x_i)\|^2 + \|f(t_i, x_i)\|^2 \right). \end{aligned} \tag{4}$$

In this situation, our training data is of the form $\{(t_i, x_i, u_i)\}_{i=1}^{N}$ is obtained by solving the PDE using some other solver and is sampled randomly from the domain. Again, the loss function MSE_u corresponds to the training data while MSE_f enforces the structure of the PDE by penalizing solutions that do not respect this structure.

In this case, the neural network is trained on both noisy and unperturbed data. When we add noise, we do so by specifying a noise factor n. More precisely, the measurements u in the dataset are now of the form

$$u_i = u(x_i, t_i) + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$ and $u(x, t)$ is the true model. The variance σ^2 is given by

$$\sigma^2 = \left(n\sigma_u \right)^2$$

where n is the noise factor and σ_u is the sample standard deviation of the unperturbed measurements u. When training on the unperturbed data, the MSE is minimized with the limited memory BFGS method and when the data is noisy, the MSE is minimized with the Adam optimizer (see Section 2.2.2).

3.3.2 Discovery of $\mathcal{N}$

Now suppose that we have a nonlinear PDE of the general form

$$u_t = \mathcal{N}(t, x, u, u_x, u_{xx}, \dots), \tag{5}$$

where $\mathcal{N}$ is a nonlinear function of time t, space $x = (x_1, \dots, x_m)$, the solution $u = (u_1, \dots, u_n)$, and its derivatives.

Furthermore, assume that we do not know the particular structure of $\mathcal{N}$. By first constructing the solution u as a feed-forward neural network and the function $\mathcal{N}$ as another neural network, we can define a deep hidden physics model via

$$f = u_t + \mathcal{N}(t, x, u, u_x, \dots),$$

where the activation function is sinusoidal.

Parameters of the neural networks u and $\mathcal{N}$ can be learned by minimizing the sum squared error:

$$SSE = \sum_{i=1}^{N} \left(\|u(t_i, x_i) - u_i\|^2 + \|f(t_i, x_i)\|^2 \right); \tag{6}$$

where $\{(t_i, x_i, u_i)\}$ are the training data on u. The first term attempts to fit the training data by adjusting the parameters of the neural network u while the second term learns the parameters of $\mathcal{N}$ by training to satisfy $u_t = \mathcal{N}$ at the points $\{(t_i, x_i)\}$. Training these parameters can be done simultaneously by minimizing Equation 6 or in sequential fashion by training u first and $\mathcal{N}$ second. Finally, in order to ensure that $\mathcal{N}$ is acceptable, we solve the PDE given by $u_t = \mathcal{N}$ using PINNs as in the previous section. Note that this algorithm is similar to that given in the previous sections, but now we seek to estimate the entire function $\mathcal{N}$.

The challenge with this approach is that $\mathcal{N}$ is a black-box function, so its analytic form is not known. Additionally, when initializing the neural network to approximate $\mathcal{N}$, we must specify the maximum order of derivatives that the neural net must compute. Thus, there is a trade-off between estimating the maximum order of derivatives needed to capture the dynamics and the complexity, i.e. training time, of the network. Note that in our applications we know the PDE from which the data was generated, so there is no guesswork in how many derivatives to include in the initialization of the model, but this will not generically be the case.

4 Applications

In this section we apply these neural networks to Burger's equation and the Swift-Hohenberg equation.

Burger's equation was explored with the intention of recreating some of the results in [9, 12, 13]. The Swift-Hohenberg equation has not appeared in the literature concerning physics informed neural networks and is explored in more detail. The data for Burger's equation and the class files the neural networks were obtained from the author's github but implemented independently [10, 11].

4.1 Burger's Equation

Consider Burger's equation in one spatial dimension with Dirichlet boundary conditions:

$$\begin{aligned} u_t &= -uu_x + \frac{0.01}{\pi} u_{xx}, \qquad \text{with } x \in [-1, 1], t \in [0, 1] \\ u(0, x) &= -\sin(\pi x) \\ u(t, -1) &= u(t, 1) = 0. \end{aligned} \tag{7}$$

Thus, the function $f(u; t, x)$ is given by

$$f(u; t, x) = u_t + uu_x - \frac{0.01}{\pi} u_{xx} \tag{8}$$

This equation leads to a shock formation and was solved using the Chebfun package in Matlab via a Fourier discretization with 256 modes and a fourth order Runge-Kutta temporal integrator with time step size 10^{-4} [3, 12]. The data for this solution can be found on Github [11].

The above neural networks for both discovering the solution to this PDE and discovering the structure of the PDE itself were implemented. The results are consistent with those published [9, 12, 13].

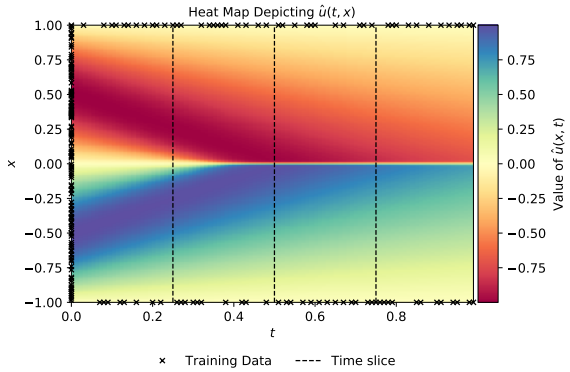
4.1.1 Data Driven Solutions

We begin by assuming that coefficients on the spatial derivatives are known. We first choose $N_u = 200$ training points from the boundary of the domain and $N_f = 10000$ collocation points. When we train the model, we obtain the following errors on the entire data set:

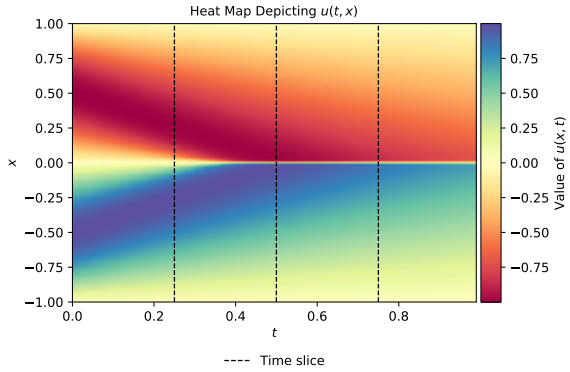
Rel. Error	Abs. Error	Max. Pointwise Error
$1.96 \cdot 10^{-3}$	$1.91 \cdot 10^{-1}$	$1.88 \cdot 10^{-2}$

Table 1: Relative and Absolute Error of $\hat{u}$ for Burger's equation.

The dynamics of the true and predicted solution are as follow. Additionally, we can plot the true and predicted

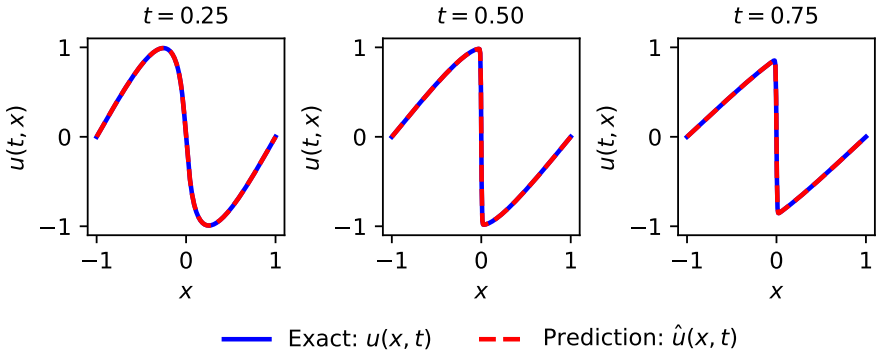


(a) Predicted Dynamics



(b) Exact Dynamics

solutions for fixed times over the spatial domain.



(c) Predicted Dynamics

As depicted, the model does a good job of extrapolating the dynamics on the entire spatiotemporal domain from the training data. Furthermore, these results are consistent with those in [12].

4.1.2 Data Driven Identification

First we suppose that we now longer know the PDE parameters so we write

$$u_t = -\lambda_1 u u_x + \lambda_2 u_{xx}.$$

Recall that $\lambda_1 = 1$ and $\lambda_2 = \frac{0.01}{\pi} \approx 3.18 \cdot 10^{-2}$.When we predict the parameters with and without noise, we obtain the following:

Noise Level	Est. λ_1	λ_1 Error	Est. λ_2	λ_2 error	Rel. Error u	Abs. Error u
0.0	.956	4.36%	$4.21 \cdot 10^{-3}$	.102%	$4.98 \cdot 10^{-2}$	4.90
.01	.982	1.73%	$5.48 \cdot 10^{-3}$	.230%	$4.77 \cdot 10^{-2}$	4.85

Table 2: Errors of parameter discovery with 200 training points.

Note that the performance with the noisy data was better with respect to some of the above metrics. This could be because the model trained on noisy data was optimized using a different optimization algorithm.

4.2 Swift-Hohenberg Equation

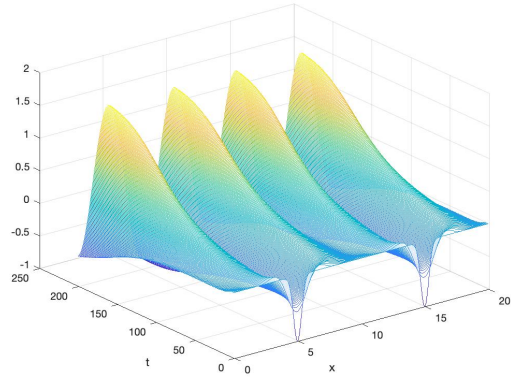
We will be interested in the following boundary value problems:

$$\begin{aligned} u_t &= -u_{xxxx} - 2u_{xx} + (-1+\mu)u + \nu u^2 - u^3 \\ u_1(0,x) &= \exp\left\{-(x-10)^2\right\} \\ u_2(0,x) &= -\exp\left\{-(x-5)^2\right\} - \exp\left\{-(x-15)^2\right\} \\ x \in [0,20], &\quad t \in [0,2], \quad u \in \mathbb{R}. \end{aligned} \tag{9}$$

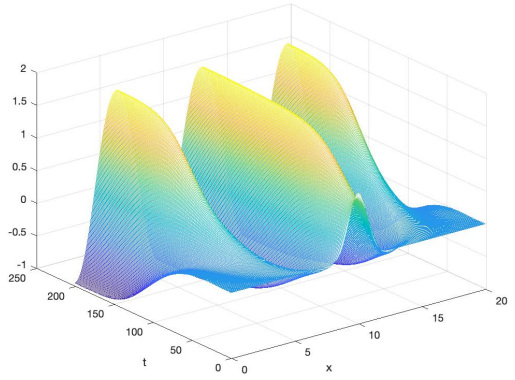
We will fix $\mu = 1$ and $\nu = 1.2$.

The data for the solutions to the Swift Hohenberg equation were obtained using the Chebfun package in Matlab via a Fourier discretization with 512 model and an exponential time differencing fourth order Runge-Kutta (ETDRK) temporal integrator with time step 10^{-6}. The solution data was collected at temporal time steps of size $\Delta t = 0.025$ and spatial time step of size $\Delta x \approx .003$.

The two boundary problems lead to rolls.



(d) Initial Condition: $u(x,0) = -\exp\left(-5(x-5)^2\right) - \exp\left(-5(x-15)^2\right)$



(e) Initial Condition: $u(x,0) = \exp\left(-(x-10)^2\right)$

We will work with the dataset corresponding to Figure 5(b) unless otherwise noted.

4.2.1 Data Driven Solutions

In this section, μ and ν are known to the learning model. We choose the number of training points $N_u = 400$ and the number of collocation points to be $N_f = 10000$. Note that this is $\approx 2.73\%$ of the total number of data points.

The maximum number of iterations the neural network was allowed to perform was set to $N = 50000$ but in each case, the training terminated with less than 10000 iterations due to the relative loss function given by Equation 3.

In contrast to the success with Burger's equation, the neural network was not able to adequately predict the dynamics on the interior of the domain (x, t) when the training points were sampled solely from the boundary. This indicates that there is not enough variability in the boundary data to captures the dynamics that occur elsewhere. However, the addition of interior training points greatly improve the performance of the model. We train the model again, first with half of the training points restricted to the boundary and the other half restricted to the interior of the spatiotemporal domain. The model is then trained again on points that are taken only from the interior. The performance of the model trained on interior data points is marginally better than that of the model trained on a mixture of interior and boundary training points. The errors are depicted in the below table; see Appendix A.1 for figures depicting the learned and true dynamics.

Training Data Location	Rel. Error	Abs. Error	Max. Pointwise Error
Boundary	$9.13 \cdot 10^{-1}$	$2.34 \cdot 10^{-2}$	1.85
Interior	$1.43 \cdot 10^{-2}$	3.67	$1.99 \cdot 10^{-1}$
Boundary and interior	$2.34 \cdot 10^{-2}$	5.95	$2.16 \cdot 10^{-1}$

Table 3: Relative and Absolute Error of $\hat{u}$ with 400 training points and 10000 collocation points.

4.2.2 Data Driven Discovery of Parameters

The model was first trained to identify the parameters μ and ν in the absence of noise, then a noise factor was added and the process was repeated.

Recall that this means the measurements u are now of the form

$$u_i = u(x_i, t_i) + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$ and $u(x, t)$ is the true model. The variance σ^2 is given by

$$\sigma^2 = \big(n\sigma_u\big)^2$$

where n is the noise factor and σ_u is the sample standard deviation of the unperturbed measurements u.

The errors in parameter estimation in the absence of noise and the addition of noise with $n = 0.02$ are depicted in the below table. Recall the true parameter values are $\mu = 1$ and $\nu = 1.2$.

Noise Level	Est. μ	μ Error	Est. ν	ν error	Rel. Error u	Abs. Error u
0.0	.96	4.09%	1.23	2.84%	$2.34 \cdot 10^{-2}$	6.01
.02	1.003	.28%	1.19	.544%	$1.61 \cdot 10^{-2}$	4.13

Table 4: Errors of parameter discovery with 400 training points.

At first, it is surprising that the results were improved with noisy data. However, the Adam optimizer is used in addition to the L-BFGS-B optimization method when there is added noise. This implies that the Adam optimization method may be more well suited to parameter discovery than the L-BFGS-B optimization method.

The model was trained to discover the parameters for a variable number of noise levels, training data, layers and neurons. See Appendix A.2. In many of the combinations, the model is able to identify the parameters μ and ν with a high level of accuracy. This accuracy seems to break down consistently with a noise level of $n = .10$.

4.2.3 Data Driven Discovery of the PDE

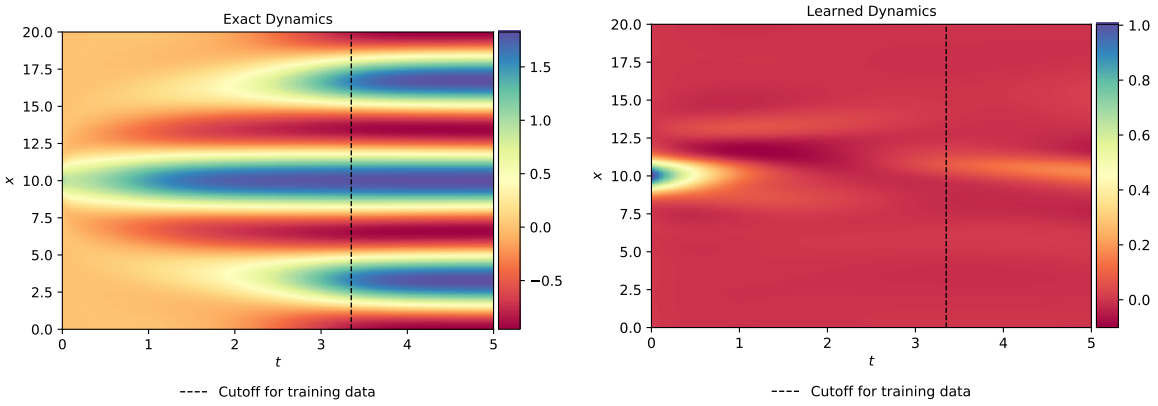
Lastly, we use the HPM class to discover the unknown PDE pf the form $u_t = \mathcal{N}(u)$ from the data and then we use the PINN class to solve the obtained PDE. The output is then compared to the known dynamics.

We choose 10000 training points and 200000 collocation points. First, we train the network that predicts $\mathcal{N}$ and the network that predicts u so that u satisfies $u_t = \mathcal{N}$ using the same data set. Following [9], we restrict the sampling of training data so that all training points come from the first 2/3 of the temporal domain. We then repeat this process but the training data that predicts u comes from a different dataset. This is testing the robustness of $\mathcal{N}$ since both datasets satisfy the same PDE and have different boundary conditions. Both results were unsatisfying. The errors are quantified below.

Data sets	Relative Error u	Abs. Error u
Same	$9.8 \cdot 10^{-1}$	$2.53 \cdot 10^{2}$
Different	2.54	$8.53 \cdot 10^{2}$

Table 5: Errors for PDE discovery and extrapolation.

The heat maps depicting the learned and true dynamics are given below. In the first set of figures, the neural network identifying $\mathcal{N}$ and the network predicting u were trained on the same dataset.

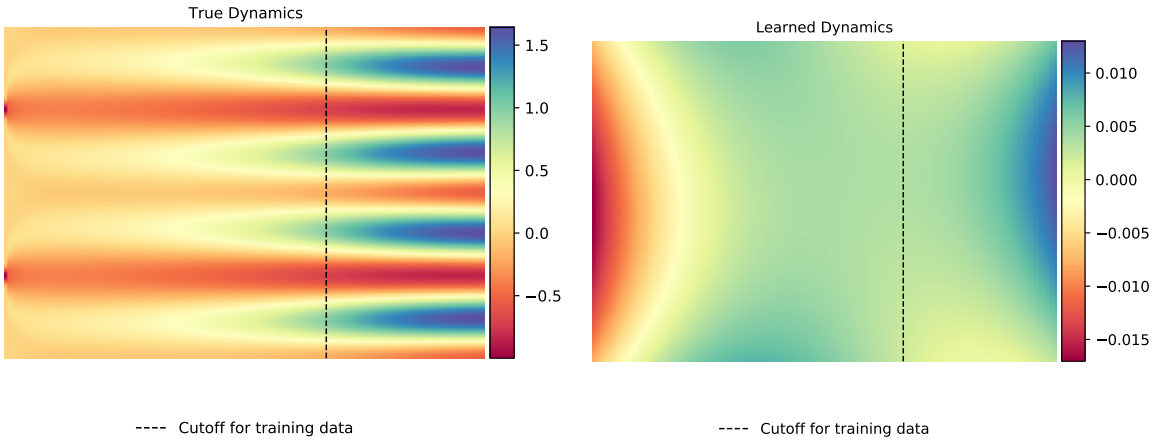


In the next set of figures, the neural network identifying $\mathcal{N}$ and the network predicting u were trained on the different datasets. This dataset was generated with initial condition

$$u(0, x) = -\exp\left\{-(x-5)^2\right\} - \exp\left\{-(x-15)^2\right\},$$

corresponding to Figure 5(a). The true and learned dynamics for u are below.

In both cases, the model either a): failed to identify the PDE $u_t = \mathcal{N}$ or b): failed to predict the solution u accurately from the identified PDE. There is not a way to distinguish between these cases with the tools that are currently available. Additionally, adding more layers may improve the performance of the models but the computer used does not have enough working memory to do so.



5 Conclusion

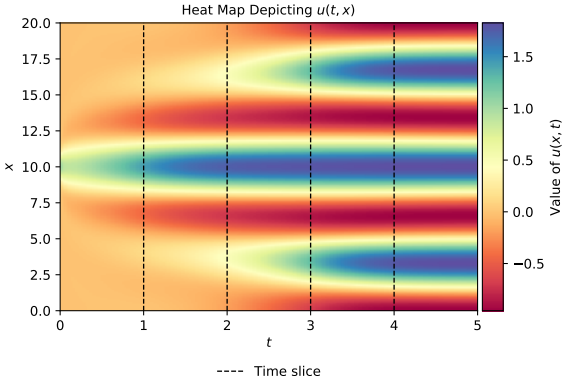
Most of the published results using physics informed neural networks were able to be reproduced with the Swift-Hohenberg equation. However, the results concerning data driven identification of the PDE were not reproducible. This could be due to the added difficulties that come along with a higher order PDE or it could be that the model was not adequately trained or deep enough.

It would be interesting to perform this analysis on the 2-D Swift-Hohenberg equation, which is known for its ability to produce solutions with spots, rings, and rolls. This was part of the original intent with this project but had to be reevaluated once computing limitations were made known.

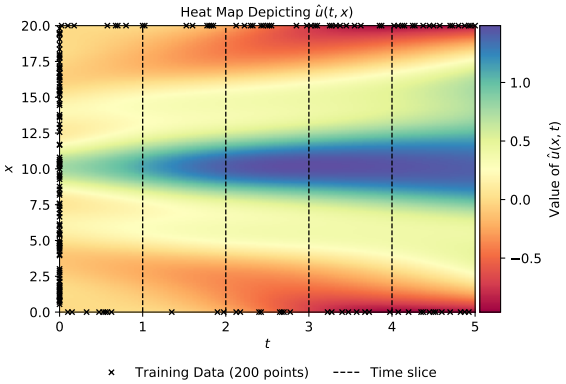
Appendix A Figures and Table

A.1 Data for Section 4.2.1

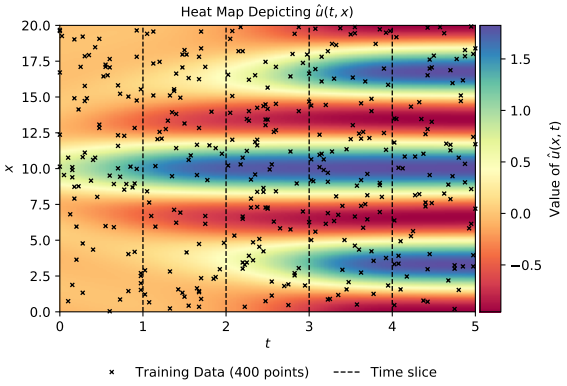
Below we have heat maps depicting the value of $u(t, x)$ over the domain (t, x). The dashed lines correspond to the locations of the snapshots depicted in Figure **??**.



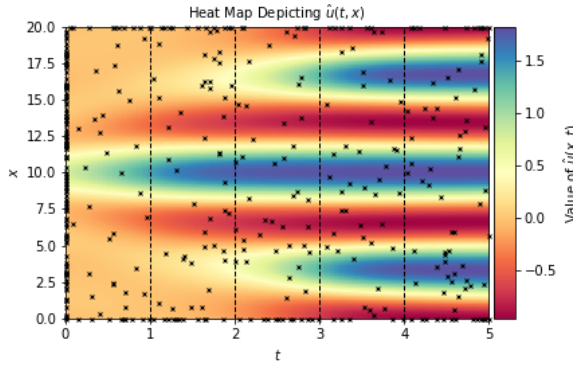
(j) True Dynamics



(k) Boundary Training Points



(l) Interior Training Points



(m) Mixed Location Training Points

Figure 3: True and learned dynamics with 400 training points and 10000 collocation points.

A.2 Data for Section 4.2.2

% Error for μ

Noise Level / # Training Points	0.0	0.05	0.10
250	0.164	0.470	10.051
750	0.087	1.133	0.216
1500	0.025	1.203	2.014

% Error for ν

Noise Level / # Training Points	0.0	0.05	0.10
250	0.016	2.720	13.906
750	0.079	0.519	0.330
1500	0.016	1.008	1.203

Table 6: Parameter estimation errors. The model had 10 hidden layers with 40 neurons per layer.

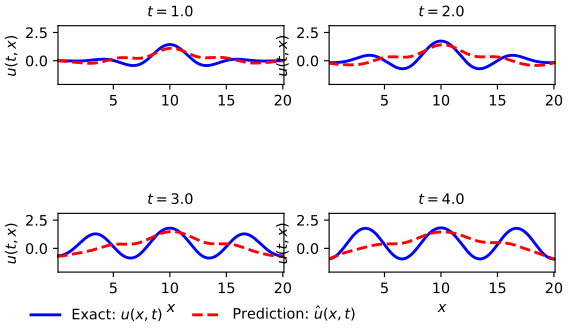
% Error for μ

# Layers / # Neurons	2	6	10
10	5.303	0.043	0.316
20	0.160	0.254	0.109
40	0.024	1.017	0.091

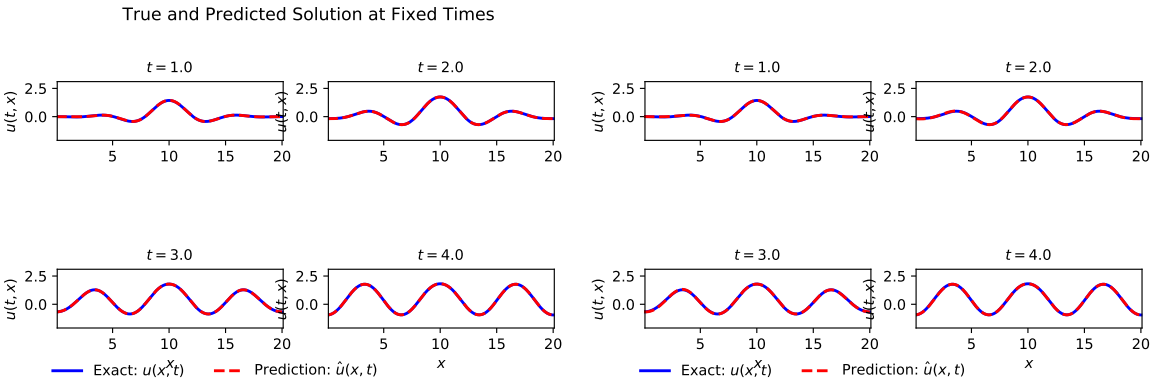
% Error for ν

# Layers / # Neurons	2	6	10
10	2.574	0.260	0.215
20	0.143	0.141	0.130
40	0.131	0.319	0.035

Table 7: Parameter estimation errors. The model was trained on 500 training points with no noise.



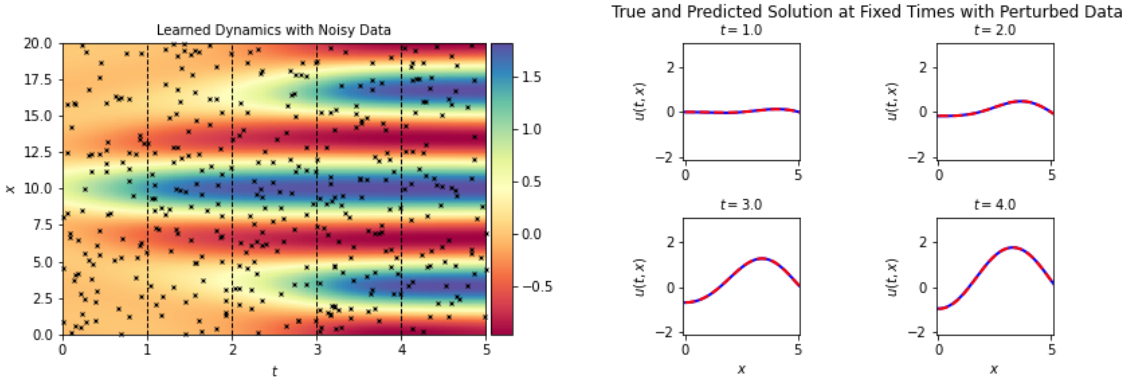
(a) Boundary Training Points



(b) Interior Training Points

(c) Mixed Location Training Points

Figure 4: Snapshots of solutions at fixed times with 400 training points and 10000 collocation points.



(a) Learned Dynamics

(b) Solution Snapshots

Figure 5: Dynamics with learned parameter values $\mu = 1.003$, $\nu = 1.19$ and noise level 0.02.

Appendix B Code

See the attached folder.. The Matlab file generated the data for the Swift-Hohenberg equation used in this project and the Python note book was used for the analysis.

% Error for μ

Noise Level # Layers	0.0	0.05	0.10
2	0.326	0.037	0.289
6	0.146	0.887	1.163
10	1.357	1.049	1.343

% Error for ν

Noise Level # Layers	0.0	0.05	0.10
2	0.207	0.045	0.182
6	0.253	0.654	1.073
10	0.232	0.276	0.330

Table 8: Parameter estimation errors. The model was trained on 500 training points with 40 neurons per layer.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, and et al. M. Devin, *Tensorflow: Large-scale maching learning on heterogeneous distributed systems*, arXiv preprint arXiv:1603.04467 (2016).

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: a survey*, Journal of Machine Learning Research **18** (2018), 1–43.

[3] T.A. Driscoll, N. Hale, and L.N. Trefethen, *Chebfun Guide*, Pafnuty Publications, Oxford, 2014. `https://www.chebfun.org/docs/guide/`, Accessed: 2021-04-01.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, Cambridge, MA, 2016. `http://www.deeplearningbook.org`.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning: Data mining, inference and prediction*, 2nd ed., Springer, New York, 2017.

[6] Kurt Hornik, Maxwell Stinchcombe, and Halbert White, *Multilayer feedforward networks are universal approximators*, Neural Networks **2** (1989), 359 –366.

[7] Diederik P. Kingma and Jimmy Ba, *Adam: a method for stochastic optimization*, Proceedings of the 3rd Conference on Learning Representations (2014).

[8] Dong C. Liu and Jorge Nocedal, *On the limited memory BFGS method for large scale optimization*, Mathematical Programming **45** (1989), 503–528.

[9] Maziar Raissi, *Deep hidden physics models: Deep learning of nonlinear partial differential equations*, Journal of Machine Learning Research **19** (2018), 1–24.

[10] ———, *Deephpms*, GitHub, 2018.

[11] ———, *Pinns*, GitHub, 2018.

[12] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis, *Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations*, arXiv preprint arXiv:1711.10561 (2017).

[13] ———, *Physics informed deep learning (part ii): Data-driven solutions of nonlinear partial differential equations*, arXiv preprint arXiv:1711.10561 (2017).