8-2022

Neural Networks and Stochastic Differential Equations

Stephanie L. Flores
The University of Texas Rio Grande Valley

DEEP LEARNING AND STOCHASTIC MODELING

A Thesis

by

STEPHANIE L. FLORES

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Applied Statistics and Data Science

The University of Texas Rio Grande Valley

August 2022

DEEP LEARNING AND STOCHASTIC MODELING

A Thesis
by
STEPHANIE L. FLORES

COMMITTEE MEMBERS

Dr. Hansapani Rodrigo
Chair of Committee

Dr. Tamer Oraby
Committee Member

Dr. Erwin Suazo
Committee Member

Dr. Santanu Chakraborty
Committee Member

August 2022

ABSTRACT

Flores, Stephanie L., Deep Learning and Stochastic Modeling. Master of Science (MS), August, 2022, 53 pp., 11 figures, references, 25 titles.

Influenced by the seminal work, "Physics Informed Neural Networks" by Raissi et al., 2017, there has been a growing interest in solving and parameter estimation of Nonlinear Partial Differential Equations (PDE) with Deep Neural networks in recent years. In fact, this has broadened the pathways and shed light on using deep learning to solve stochastic PDE's (SPDE).In this work, we intend to investigate the current approaches of solving and parameter estimation of the stochastic heat equation with convolution neural networks and generative adversarial neural networks. The combination of methods can improve speeds, accuracy, and lessen data-related difficulties in solving Stochastic PDEs. Such improvements can assist a wide array of the sciences in computational research and data sciences.

DEDICATION

To my parents, who always supported me in my pursuit of education.

And to my favorite person, Jorge. Thank you for showing me the importance of a good laugh and a good cup of coffee.

ACKNOWLEDGMENTS

I want to thank Dr. Oraby and Dr. Suazo for their continued help in my studies with stochastic differential equations. The research conducted with you both and Dr. Yoon during the summer REU helped me become a better student.

I want to thank Dr. Chakraborty for being a part of the committee and helping with aspects of the research.

Finally, I want to thank Dr. Rodrigo for her help and encouragement during the difficult and rewarding journey of writing a thesis. As a student who started research with a minimal knowledge of deep learning and programming, I believe that Dr. Rodrigo encouraged me to take the initiative to learn and grow as the student I am today.

Working on this thesis is an experience I won't forget, and I'll won't forget how everyone has helped me on this journey. I could not have done this alone, thank you.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Deep Learning

Machine learning is an process that allows computers to solve problems, using algorithms such as logistic regression and the naive Bayes to solve problems such as medical recommendations and e-mail spam detection (Goodfellow, Bengio, and Courville 2016). Many modern luxuries are possible by the works of machine learning. From technology that can conduct efficient webs search, automated driving in vehicles, computer vision and optical character recognition (Liu et al. 2017). Even with these successes, the performance of machine learning processes began to falter once more human processing mechanisms (such as speech and vision) were involved (Liu et al. 2017). One of the problems with these artificial intelligence task was a requirement of **representations** of data; there has to be a human labeling component, which can become difficult to perform with a large amount of data (Goodfellow, Bengio, and Courville 2016).

The deep learning model is a solution to this problem. Deep learning is a computational model comprised of multiple processing layers that learn representations of data with multiple levels of abstraction (LeCun, Benglo, and Hinton 2015). There are several factors that make deep learning an attractive option when problem-solving: a universal learning approach, which allows models to be used in all types of domains, robustness, which does not require specification for features when the model is learning, generalization, in which in which different types of data and applications are able to use the same deep learning model, and scalability (Alzubaidi et al. 2021). Thus, the deep learning model has led to much breakthrough in processing images, video, speech, and audio (LeCun, Benglo, and Hinton 2015). Due to ability of a deep learning model to process

and interpret large amounts of *unlabeled* data, the, deep learning has become a staple in the big data analysis scene, used in cyber security, medical informatics, and social media (Liu et al. 2017). It is because of this ability to interpret abstract data that deep learning practices would be useful for solving problems traditionally done numerically — differential equations.

1.2 Stochastic Partial Differential Equations

In order to describe natural phenomena, differential equations are usually utilized in areas such as biology, mathematics, physics, and more. Differential equations are used to solve problems such as image smoothing medical magnetic resonance images, estimating population growth and decay, fluid motion and more (Lysaker, Lundervold, and Tai 2003) (Bahuguna and Dabas 2008) (Siemrod 1976). *Differential equations* are described to be mathematical models relating to an unknown function and one or more of its derivatives, while a *partial differential equation* is a differential equation that involves partial derivatives of a function of more than one variable (Trench 2013). One type of differential equation that is especially interesting is the *stochastic differential equation*, a type of differential equation with randomness in the coefficients of the equation (Øksendal 1998). There are many ways to solve these types of differential equations, depending on the problem at hand. Some traditional methods of solving these types of problems are filtering and Ito integration (Handel n.d.).

What exactly is a stochastic differential equation? We stated that it is a PDE with a stochastic term. To further elaborate, The stochastic term represents a stochastic process. A common archetype of stochastic processes is Brownian motion, which describes many natural phenomena, such as the movement of pollen particles in fluid, gaseous momentum, motions of fluids, and more (Paul and Bashchnagel 2000). Brownian motion describes the phenomenon of the outcome of many unpredictable and at times unobservable events, such as the change of the stock market or collisions of particles, or investment decisions, which all lead to a observed effect (Paul and Bashchnagel 2000).

A common modern problem of the information age is the abundance, or lack, of data. This conundrum of data is addressed in the paper (Yang, Zhang, and Karniadakis 2018), in which the

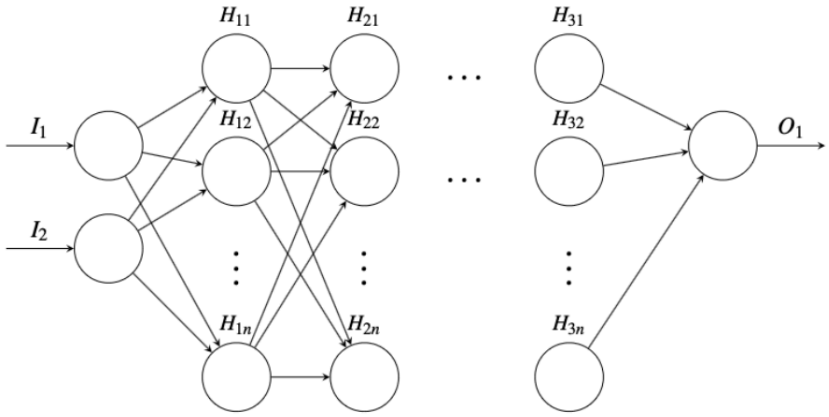

Figure 1.1: Simple schematic of a neural network

authors discuss the three cases of data in physical fields:

- the lack of data, but understanding of underlying physics
- the access of some data and some understanding of physics
- the abundance of data, but a lack of understanding of physics

How does one solve differential equations with these data conundrums? We want a solution that is adaptable and can handle a variety of cases, as described above. This paper and the research behind it is heavily inspired by the writings of Raissi and their paper (Raissi, Perdikaris, and Karniadakis 2017) (insert complete one, better). The works of Raisai et. all speak of using physics informed neural networks to solve partial differential equations.

1.3 Further information on Stochastic Differential Equations

In previous works, we considered the following two different stochastic Burgers equations with a space-uniform white noise of the form

$$du = (A(t)\partial_{zz}u + B(t)u\partial_z u + C(t)\partial_z u + D(t)u)dt + E(t)\partial_z u\, dW_t \tag{1.1}$$

and

$$du = (A(t)\partial_{zz}u + B(t)u\partial_z u + C(t)\partial_z u + D(t)u)dt + E(t)dW_t \tag{1.2}$$

for $t \in [t_0, T]$ and $z \in \mathbb{R}$ with $u(0,z) = \phi(z)$ for $z \in \mathbb{R}$ (Flores et al. 2020). The diffusion heat equation that we want to work on takes a similar form to equation 1.1, but with $B(t) = 0$, $C(t)$=0, $D(t) = 0$, and $E(t) = \sigma$.

Most physical and biological systems are not homogeneous, in part due to fluctuations in environmental conditions and the presence of nonuniform media. Therefore, most of the nonlinear equations with real applications possess coefficients varying spatially and/or temporally and even stochastic terms. Reaction –diffusion equations play a fundamental role in a large number of models of heat diffusion and reaction processes in nonlinear acoustics, biology, chemistry, genetics and many other areas of research (Flores et al. 2020).

Consider the probability space $(\Omega, \mathscr{F}, \mathbf{P})$ for which the Brownian motion $\{W_t, t \geq 0\}$ is defined and $E(W_s W_t) = \min(s,t)$ for all $s,t \geq 0$. Also consider the filtration $\mathscr{F}_t := \sigma(W_s : s \leq t)$ being the smallest $\sigma-$algebra to which W_s is measurable for $s \leq t$.

Then consider the stochastic differential equation (SDE) with variable coefficients (Flores et al. 2020)

$$dX_t = \alpha(t, X_t)dt + \beta(t, X_t)dW_t, \tag{1.3}$$

with initial state X_{t_0} and for $t \in [t_0, T]$. The SDE in (1.3) has a general solution given by

$$X_t = X_{t_0} + \int_{t_0}^{t} \alpha(s, X_s)ds + \int_{t_0}^{t} \beta(s, X_s)dW_s$$

for $t \leq T$. If $\alpha(t) := \alpha(t, X_t)$ and $\beta(t) := \beta(t, X_t)$, then equation (1.3) has a general solution given by

$$X_t = X_{t_0} + \int_{t_0}^{t} \alpha(s)ds + \int_{t_0}^{t} \beta(s)dW_s$$

for $t \leq T$. The process $\{W_t; t \geq 0\}$ is a Wiener process with respect to a filtration $\{\mathscr{F}_t; t \geq 0\}$. The initial state X_{t_0} is $\mathscr{F}_{t_0}$ and the functions $\alpha(t)$ and $\beta(t)$ are Lebesgue measurable and bounded

on $[t_0, T]$. The latter implies both the global Lipschitz and linearity growth conditions required to ensure the existence and (path-wise) uniqueness of a strong solution to (1.3), (Flores et al. 2020).

Let X_t and Y_t be any two diffusion processes like those defined by the solution of equation (1.3). If $F(x,y)$ is a differentiable function that works as a transformation for two processes X_t and Y_t, then the general bi-variate Itô formula (Flores et al. 2020) gives

$$\begin{aligned} dF(X_t,Y_t) &= \partial_x F(X_t,Y_t)dX_t + \partial_y F(X_t,Y_t)dY_t + \frac{1}{2}\partial_{xx}F(X_t,Y_t)(dX_t)^2 \\ &\quad + \frac{1}{2}\partial_{yy}F(X_t,Y_t)(dY_t)^2 + \partial_{xy}F(X_t,Y_t)dX_t dY_t. \end{aligned} \tag{1.4}$$

$F(t,y)$ is a differentiable function. If Y_t is a diffusion process that solves (1.3), then the Itô formula becomes (**Kloeden1992**)

When $X_t = t$ the general Itô formula of $F(t,y)$ is a differentiable function. If Y_t is a diffusion process that solves (1.3), then the Itô formula becomes (**Kloeden1992**)

$$dF(t,Y_t) = f(t,Y_t)dt + g(t,Y_t)dW_t, \tag{1.5}$$

where

$$f(t,x) = \partial_t F(t,x) + \alpha(t,x)\partial_x F(t,x) + \frac{1}{2}\beta^2(t,x)\partial_{xx}F(t,x)$$

and

$$g(t,x) = \beta(t,x)\partial_x F(t,x).$$

Before introducing the numerical algorithm for solving (1.1) and (1.2), we must first introduce the following central proposition. The proposition also assists in finding exact solutions for equations (1.1) and (1.2), in particular when exact solutions of the deterministic differential equations (1.6).

The following is the stated propositions in (Flores et al. 2020): Let $A,B,C,D,E \in \mathscr{C}^b([t_0,T])$ be bounded continuous functions on $[t_0,T]$. Assume that $B(t) > 0$ for all $t \in [t_0,T]$. Then, we have:

1. The stochastic Burgers equation with the initial value problem (1.1) has a solution $u(t,z) =$

$U(t,X_t)$, where $U(t,x)$ is the solution of

$$\partial_t U = (A(t) - \frac{1}{2}E^2(t))\partial_{xx}U + B(t)U\partial_x U + D(t)U, \quad U(0,x) = \phi(x) \tag{1.6}$$

and X_t is the solution of

$$dX_t = C(t)dt + E(t)dW_t \tag{1.7}$$

with initial state $X_{t_0} = z$ and for $t \in [t_0, T]$.

We also take into consideration the following Lemma, taken from (Flores et al. 2020):

1. The stochastic process X_t solving

$$dX_t = C(t)dt + E(t)dW_t$$

with $X_{t_0} \sim N(x_{t_0}, \sigma_0^2)$ independent of W_t, is a non-stationary Gaussian process with mean $x_{t_0} + \int_{t_0}^t C(s)ds$ and variance $\sigma^2(X_t) = \sigma_0^2 + \int_{t_0}^t E^2(s)ds$.

2. The covariance of the two processes X_t and W_t is

$$\sigma(X_t, W_t) = \int_{t_0}^t E(s)ds.$$

3. Moreover,

$$[X_t|W_t = w] \sim N\left(x_{t_0} + \int_{t_0}^t C(s)ds + \frac{\int_{t_0}^t E(s)ds}{t}w, \sigma_0^2 + \int_{t_0}^t E^2(s)ds - \frac{(\int_{t_0}^t E(s)ds)^2}{t}\right).$$

The proof of this lemma can be found in previous works (Flores et al. 2020).

1.4 Neural Networks

Neural Networks have been a hot topic as of late. A neural network defines a wide class of flexible nonlinear regression and discriminant models, data reduction models, and nonlinear dynamical systems, defined in (Sarle 1994). These neural networks are named so because of their

resemblance of a neuron located in the human brain, and their behavior is similar (source). (add a picture, mention layers, hidden, outputs use the one in report) Common terminology of neural networks is defined in (Sarle 1994) as follows: variables are *features*, independent variables are called *inputs*, predicted values are called *outputs*, dependent variables are called either *targets* or *training values*, and residuals (the measure of difference between outputs and targets) are called *errors*.The estimation by neural networks is commonly called *training*, obserations are called *patterns*, parameter estimates are called *weights*, interactions are called *higher-order neurons* (Sarle 1994). Transformations in the model are called *funcitonal links*, regression and discriminant analysis is called *supervised learning*, and interpolation and extrapolation are called *generalization* (Sarle 1994). (Mention ANN and DNN) With neural networks, data is often divided into a *training set* and a *test set* for cross-validation (Sarle 1994). As advances in deep neural networking continue to grow, and with the inspiration from the aforementioned research work, we aim to further explore and possibly extend the SDEs with deep neural networks (Raissi, Perdikaris, and Karniadakis 2017).

Along with techniques inspired by Raissai et. all, we would like to utilize another deep learning model: generative adversarial networks (GANs). GANs are a model architecture for training a generative model, and are comprised of a **generator** – a model that is used to generate new plausible examples from the problem domain – and a **discriminator** - a model that is used to classify examples as real (from the domain) or fake (generated) (Brownlee 2019).

CHAPTER II

METHODOLOGY

2.1 Setup

2.1.1 The Diffusion Equation Solution

The equation that we decide to focus on in this project is the diffusion equation. The diffusion equation is a PDE which describes density fluctuations with materials undergoing diffusion ("The Diffusion Equation" 2018). The standard diffusion equation is as follows:

$$\frac{\partial u}{\partial t} = \nabla \cdot (D(u,x)\nabla u), \tag{2.1}$$

in which $u(x,t)$ represents the diffusing material at position x and time t. The diffusion coefficient for density u at x is denoted by $(D(u(x,t),x)$ ("The Diffusion Equation" 2018). In our research, we focused on the case in which D is constant, which reduces equation 2.1 to

$$\frac{\partial u}{\partial t} = D \cdot \frac{\partial^2 u}{\partial x^2}, \tag{2.2}$$

which call also be described as the heat equation. We decide to focus in on the one-dimensional heat equation, as well as choosing Cauchy conditions: $|x| < \infty$, $t > 0$, with initial conditions $u(x,0) = f(x)$ (Kumar n.d.). Implementing a change of variables and considering the heat equation's properties of conservation of energy, it can be found that the fundamental solution of the heat equation is as follows (Kumar n.d.):

$$\Phi(x,t) = \frac{1}{\sqrt{4\pi D t}} e^{-\frac{x^2}{4Dt}}. \tag{2.3}$$

Using this information as well as the previous works done in (Flores et al. 2020), particularly referring to 1.1 and Proposition 1.1 in (Flores et al. 2020), it follows that the stochastic 1-D heat equation has the following solution:

$$X(t) = X(0) + N(0, \sigma^2 * t; 0, t) \tag{2.4}$$

,

Thus, we add a stochastic term to equation 2.5 to get

$$\Phi(x,t) = \frac{1}{\sqrt{4\pi D t}} e^{-\frac{(x+\sigma * t)^2}{4Dt}}. \tag{2.5}$$

Once we had our solution for the stochastic 1-D diffusion equation, we can generated a dataset for our deep learning models. We chose the following parameters for our diffusion equation: $u(x,0) = f(x)$, where $f(x) = x$. We generated 50 arrays for each pair of σ and D, to end up with a dataset of 5,000 arrays. With these arrays, we intend to use our CNN model for parameter detection, and our GAN for solution generation.

2.1.2 CNNs

Our first objective concerns parameter discovery using convolutional neural networks (CNNs). CNNs are a form of deep supervised learning, which is consistent with the GANs (Alzubaidi et al. 2021). CNNs are great because the can identify relevant features in data without any human interference; the model has been used in applications such as computer vision, speech processing, facial recognition, and more (Alzubaidi et al. 2021). Another benefit of this model is the fact that they are able to share weights and local connections, which leads to a faster training process than regular fully connected networks (Alzubaidi et al. 2021).

There are several components of a CNN that differentiate them from other deep learning models. First, there is the convolutional aspect of the model, in which the input data (an image with height, width, and 3 layers representing the RGB channel of the image) has regional connections to nodes of the next layer of the network, as well as the fact that the layers have fixed weights

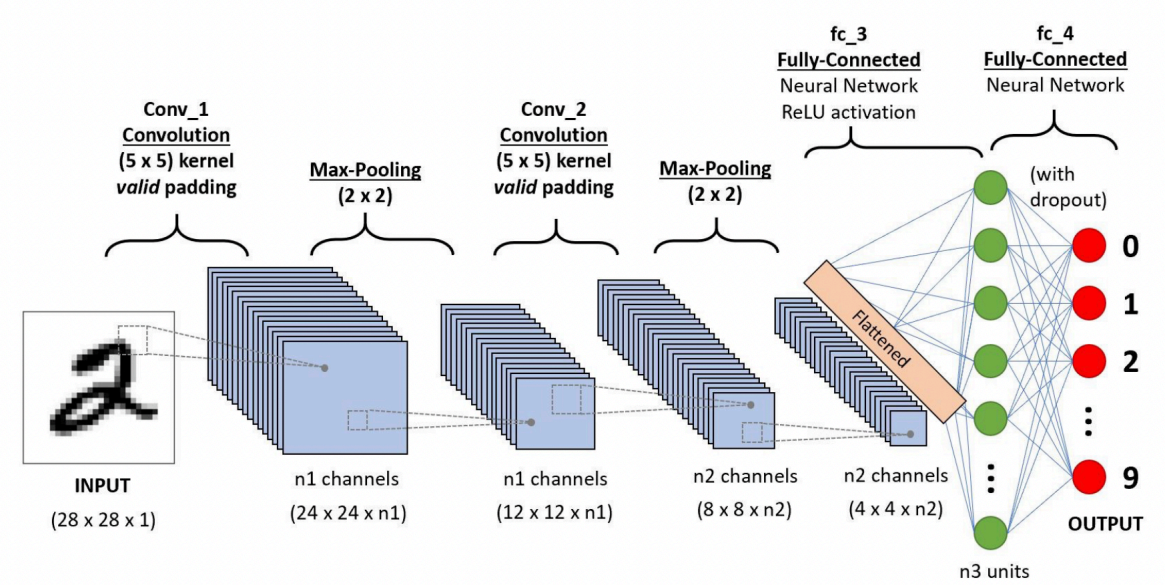

Figure 2.1: Architecture of a standard CNN. Demonstrates common practices such as padding and pooling of image. Image from ("What is the Convolutional Neural Network Architecture?" 2020)

(Albawi, Mohammed, and Al-Zawi 2017). This 'window' and fixed weight implementation creates a convolutional matrix (filters), which can create operations such as edge detection, sharpening, and blurring of images (Albawi, Mohammed, and Al-Zawi 2017). Controlling how the filter moves through the image is called stride: controlling the amount of stride controls the size of the output. To prevent loss from the border of an image, we can implement padding in a CNN, which prevents data loss and manages output size of a convolutional layer (Albawi, Mohammed, and Al-Zawi 2017). The data then passes through a nonlinear activation layer (in our case, we used a ReLU activation function), and then a pooling layer in which the image is 'downsized', only the larges values of a window are kept as the image is downsized (Albawi, Mohammed, and Al-Zawi 2017). Finally, the last layers of the CNN include a flattening layer which then connects to a fully connected network, which ends with an output layer which is usually a classification or regression layer.

Since we want to use the CNN model for parameter estimation of σ and D, we decided to go for a multi output CNN, inspired by (Bressan 2020). Similar to the model of (Bressan 2020), we create a model with branches with similar layers, but with a regression output to approximate the parameters, as shown in image 2.2.

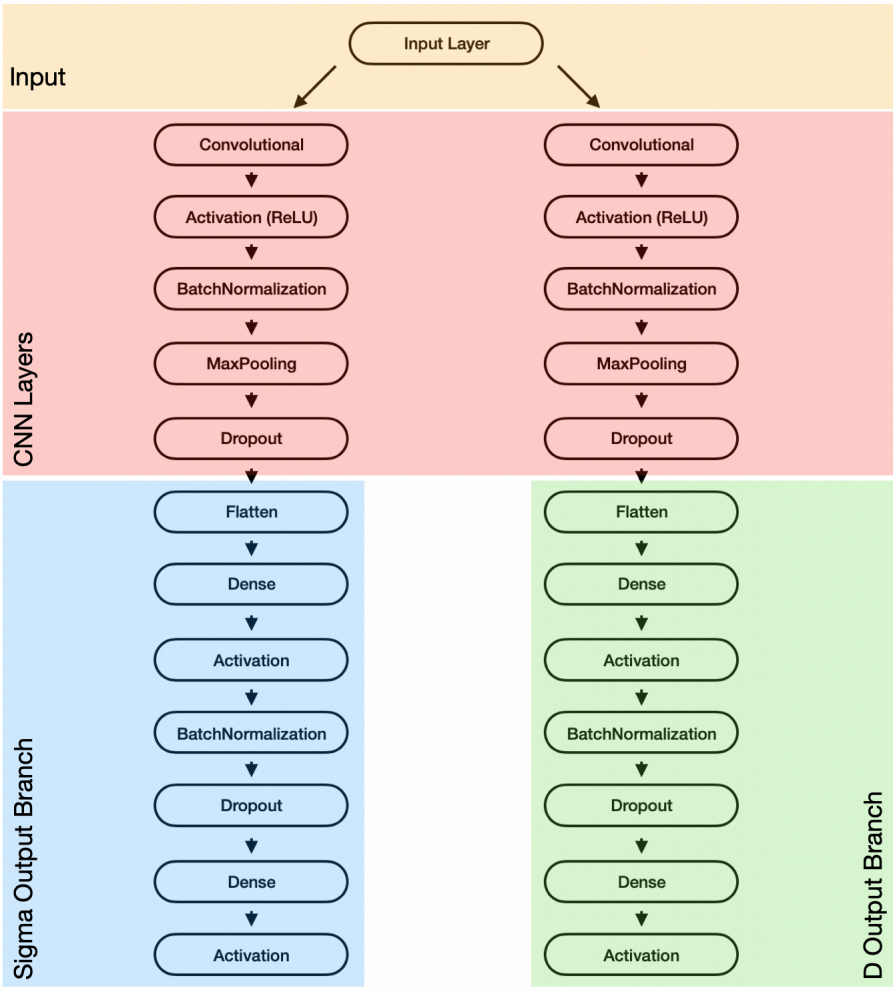

Figure 2.2: The architecture of the mult-output CNN, split into two branches. The red indicates the CNN layers, which take in the image input. Inspired by (Bressan 2020)

Since the output of the model is a regression, we decided to use the Mean Average Error (MAE) and the R2 score to examine the performance of the model. The MAE of a regression model measures the average magnitude of absolute differences between N predicted vectors $S = x_1, x_2, \ldots, x_N$ and $S^* = y_1, y_2, \ldots, y_N$ (Qi et al. 2020). The corresponding loss function is described as:

$$\mathscr{L}_{MAE}(S, S^*) = \frac{1}{N} \sum_{i=1}^{N} ||x_i - y_i||,$$

where $||\cdot||_1$ denotes L_1 norm (Qi et al. 2020).

The coefficient of determination, also known as the R^2, is a measure of the goodness of fit of a model, specifically measuring how well the regression line approximates the actual data ("Coefficient of Determination, R-squared" n.d.). A widely used formula for the coefficient of determination is as follows:

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y}_i)^2}, \tag{2.6}$$

where the numerator of the second term is commonly referred to as the sum squared regression (SSR) and the denominator is referred as the total sum of squares (SST) ("Coefficient of Determination, R-squared" n.d.). The SSR refers to the sum of residuals squared, and the SST is the sum of the distance the data is away from the mean all squared ("Coefficient of Determination, R-squared" n.d.). Usually, a score closer to one denotes a great fit, while a score close to zero or a negative number denotes a poor fit ("Coefficient of Determination, R-squared" n.d.).

2.1.3 GANs

Our research utilized a type of deep learning network called a generative adversarial network, also known as GANs. Initially, we explored the idea of exploring different types of GANs, such as vanilla GANs, Deep Convolutional Generative Adversarial Networks (DCGANs) or Wasserstein GANs (WGANs). Eventually, we settled on the utilization of the WGAN for several beneficial reasons: Although these models were primarily used in the context of computer vision (Mwiti n.d.),

novel research demonstrates that these models can be utilized with any array-based data. we believe that they can be successfully utilized in modeling PDEs and SDEs due to their probabilistic learning capability.

A plain vanilla GAN consists of two sub-models; a generator, which generates new scenarios/examples, and a discriminator, which classifies examples as real or fake (Brownlee 2019). Generators take as inputs a vector of noise, usual generated with a Gaussian distribution, which initially has no meaning (Brownlee 2019). After the model is trained, points in the multi-dimensional vector of space correspond to points in the problem domain; the generator model creates a latent (hidden) space which recognizes patterns in the domain that are not necessarily observable directly, and thus new points may be drawn from the latent space and can be used to generate new examples(Brownlee 2019). In vanilla GANs, the discriminator model takes as input an example from the problem domain as input (either real or generated) and conducts a binary classification as real or fake on the input (Brownlee 2019). Once the GAN has been trained, the discriminator is discarded, and only the generator is preserved (Brownlee 2019). Using this model, which corrects itself by rejecting bad examples and reinforcing its discrimination accuracy, creates an architecture that is able to learn and recognize patters such that little supervision is needed (Brownlee 2019). GANs contain architecture and layers used in CNNs within them, meaning that they can take input that represents an image and generate images as well (Brownlee 2019).

2.1.4 Wasserstein GANs

While GANs are powerful deep learning models, they are also difficult to train. Pitfalls such as mode collapse, which is the problem when multiple inputs to the generator map to the same output, or the case when the generator's output oscillates without converging (the former which is an issue in our own research), vanilla GANs are a difficult model to train (Brownlee 2019). Hoping to overcome these issues, we chose to work with Wasserstein GAN models Wasserstein GANs are different from plain vanilla GANs in that they seek an alternative way of training the generator model to better approximate the distribution of data observed in a given training data set, and rather than using a discriminator, the WGAN uses a *critic* to score the "realness" or "fakeness"

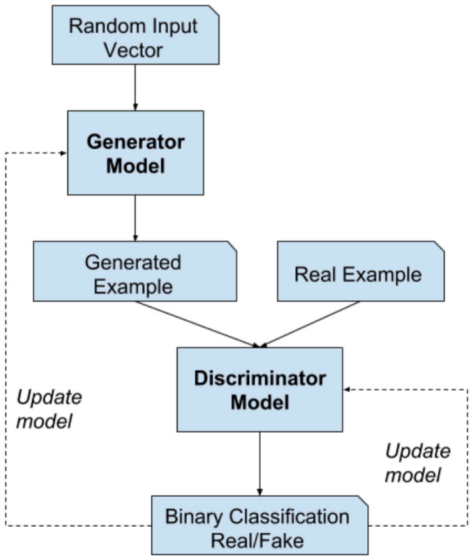

Figure 1.7: Example of the Generative Adversarial Network Model Architecture.

Figure 2.3: An illustration of the architecture of a vanilla GAN showing the relation of a generator and discriminator model from (Brownlee 2019)

.

of a given image (Brownlee 2019). The difference in the two GAN models can be seen in their respective objective functions: For the vanilla GAN, the game between the generator G and the discriminator D is the described in the minimax objective

$$\min_G \max_D \mathbb{E}_{x\sim\mathbb{P}_r}[log(D(x))] + \mathbb{E}_{\tilde{x}\sim\mathbb{P}_g}[log(D(\tilde{x}))], \tag{2.7}$$

where $\mathbb{P}_r$ is the data distribution, and $\mathbb{P}_g$ is the model distribution, where $\tilde{x} = G(z)$, where z is the input generated by a Gaussian distribution(Gulrajani et al. 2017). This value function, however, can lead to training issues such as vanishing gradients (Gulrajani et al. 2017).

Thus, WGANs propose using a Wasserstein-1 distance in minimization, as the measure is continuous everywhere and differentiable everywhere (Gulrajani et al. 2017). Thus, the WGAN value function becomes (with construction using Kantorovisch-Rubinstein duality)

$$\min_G \max_{D\in\mathscr{D}} \mathbb{E}_{x\sim\mathbb{P}_r}[D(x)] - \mathbb{E}_{\tilde{x}\sim\mathbb{P}_g}[D(\tilde{x})], \tag{2.8}$$

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

Figure 2.4: This is the WGAN gradient penalty as discussed in (Gulrajani et al. 2017). This algorithm was utilized in our models.

.

where $\mathscr{D}$ is a set of 1-Lipshitz functions, which was enforced with weight clipping of the critic to lie within a compact space $[-c, c]$ (Gulrajani et al. 2017).

Weight clipping can lead to some problems, such as capacity under use, which leads the critic toward simpler functions, and exploding/vanishing gradients, due to the interaction of the weight constraint and the cost functions (Gulrajani et al. 2017). That is why the final objective function we decide to use is the following, which is an alternative way to enforce the Lipschitz constraint:

$$\min_G \max_{D \in \mathscr{D}} \mathbb{E}_{x \sim \mathbb{P}_r}[D(x)] - \mathbb{E}_{\tilde{x} \sim \mathbb{P}_g}[D(\tilde{x})] + \lambda \mathbb{E}_{\tilde{x} \sim \mathbb{P}_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2], \tag{2.9}$$

where the last term of the equation is the gradient penalty, which incorporates a sampling distribution $\mathbb{P}_{\hat{x}}$ based on $\mathbb{P}_r$ and $\mathbb{P}_g$, a penalty coefficient λ, the discarding of batch normalization, and a two sided penalty which encourages the gradient to go towards 1. (Gulrajani et al. 2017)

The use of deep learning methods are extremely likely to improve and provide more stable and accurate results over traditional numerical methods. Additionally, as evidence by previous research (Yang, Zhang, and Karniadakis 2018) these techniques are ideal when the underlying

physics are known, yet partially obscured, and when there are several scattered measurements in addition to the conventional boundary and initial conditions.

2.2 Goals

The purposes of this paper are utilize two deep learning models in the following way:

1. Generating images of the 1-D heat equation to use as a dataset,
2. Using CNNs to estimate parameters of the heat equation, and
3. Using GANs to generate solutions with the calculated dataset.

In this paper we record the results of our project in the next chapter oraganized by the goals listed above, and end with a discussion of the results and possible future directions of the work.

CHAPTER III

RESULTS

3.1 SPDE Solution Generation

Our first results to report are on the generation of the stochastic differential equation we have chosen to model. As mentioned in the methodology, we have generated a 1-dimensional stochastic heat equation. We chose several values for α and D, with the values from α ranging from .01 to .1, and the values for D ranging from .1 to 1. We focused on a Cauchy approach for the equation, thus limiting $x \in [-L, L]$, and in this case we choose the value of L to be 2.5 to limit the amount of empty space around the diffusion of the equation. For the purpose of feeding these images (shown in 3.1 into CNN model, we have chosen to plot the images without axes, labels, or a title to have the model train better.

We first generated the stochastic portion of the 1-dimensional heat equation, which changed in its intensity depending on what value of σ was used to generate the stochastic term (shown in 3.2. We then added the stochastic term to the values of x, which then allowed us to calculate the solution to the heat equation normally.

3.2 CNN

For the CNN model, we calculated the mean average error of both the test and validation model. We also plotted the actual vs. predicted points with the model on a test dataset (the data was split 70 percent training, 30 percent testing). The model that performed the best for us was the model with 100 iterations and a batch size of 32. When the model was run with a higher number of epochs, the loss and mae tend to diverge and grow.

There was a tendency for the σ value in all of our models to have a greater difficulty being

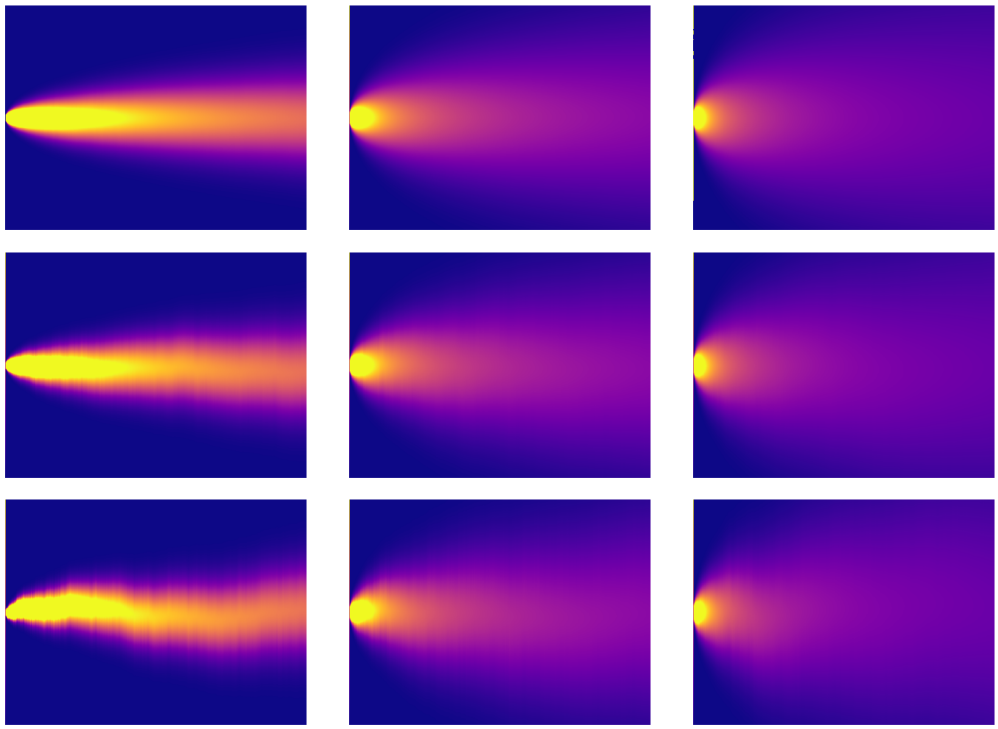

Figure 3.1: A compilation of U(x,t) Solutions with differing values of σ and *D*. Top Row: σ = 0.01, D = 0.1 ; σ = 0.01, D = 0.5 ; σ = 0.01, D = 1.0 ; Second Row: σ = 0.05, D = 0.1 ; σ = 0.05, D = 0.5 ; σ = 0.05, D = 1.0 ; Bottom Row: σ = 0.1, D = 0.1 ; σ = 0.1, D = 0.5 ; σ = 0.1, D = 1.0 ; Here, we solve the 1-D heat equation 2.5 with f(x) = x.

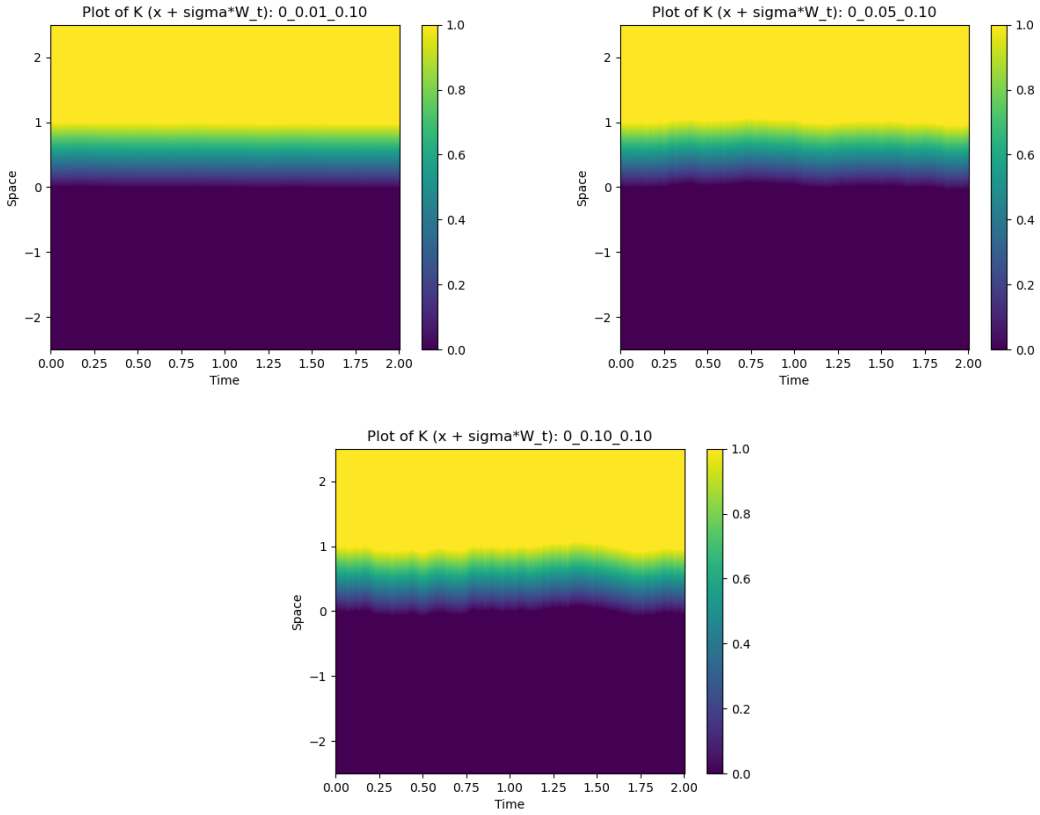

Figure 3.2: Depictions of K for different values of σ. K represents the Brownian motion $X(t) = X(0) + N(0, sigma**2*t; 0, t)$. Notice how higher values of σ show greater oscillations in the image.

	Model Name	D MAE	Sigma MAE	D vMAE	Sigma vMAE	D R2	S R2
0	100_iter_mm_lin_lr7_bs34	0.157715	0.333934	0.104279	0.358234	0.056856	-0.122344
1	200_iter_mm_st_lr7_bs64_lds	0.126652	0.305308	0.596034	0.927406	-9.427578	-6.634189
2	500_iter_mm_lin_lr7_bs64_lds	0.082451	0.257675	0.206211	0.878306	-2.052337	-1.582575
3	100_iter_mm_lin_lr7_bs32_lds	0.177157	0.317418	0.208459	0.484288	0.218106	-1.023076
4	100_iter_mm_lin_lr7_bs32_lds_4c	0.163480	0.319845	0.357276	0.314217	0.374022	-0.466732
5	100_iter_mm_lin_lr7_bs16_lds_4c	0.130520	0.288724	1.399606	0.337895	-3.967242	-0.176964
6	130_iter_mm_lin_lr7_bs32_lds_4c	0.075166	0.264950	0.296271	0.341565	-0.011688	-0.059439
7	130_iter_mm_lin_lr7_bs16_lds_4c	0.096155	0.279856	0.550155	0.474389	-3.400336	-2.973943
8	100_iter_lin_lr7_bs34	0.278807	0.302291	0.416264	0.338009	-1.291991	-4.027802
9	200_iter_lin_lr7_bs32	0.127744	0.252346	0.516816	0.257881	0.301017	-0.157616
10	100_iter_lin_lr7_bs32_weights	0.276803	0.386999	0.492170	0.485678	-0.447939	-0.189228
11	200_iter_mm_lin_lr7_bs16_lds_4c	0.096445	0.267572	0.087793	0.617799	-1.107557	-0.094979
12	100_iter_mm_lin_lr7_bs32_lds_2b	0.184048	0.314933	0.694165	0.459173	-2.102176	-2.721261
13	100_iter_mm_lin_lr7_bs64_lds_4c	0.220390	0.372539	0.314734	0.586401	-2.746418	-0.389009

Figure 3.3: A table showing the different results of several multi-output CNN models. The model that we believed performed best is highlighted in red. The values for D were usually predicted much better than that of σ.

accurately predicted with regression. The better models would predict values for σ in the correct domain, but with a low amount of accuracy. Poorer performing models would tend to generate negative values for σ. The values for D tended to be accurately predicted and in the correct domain, but models also tend to have more difficulty for smaller values of D, particularly for values of D around .1.

3.3 GAN

We had intended to utilize the parameters estimated from the CNN with the GAN. However, the model had difficulty recreating images even without parameter implementation. The standard model ran 10,000 epochs, with variations in the learning rate, parameters with the Adam Optimization, batch size, and normalization, with little effect.

Image generation tends to heavily focus on values of zero, likely a case of mode collapse of a WGAN. It is also observed that, while the original arrays have higher values towards the middle area of the image, generated images have higher values towards the top. This is a subject of further exploration for future works.

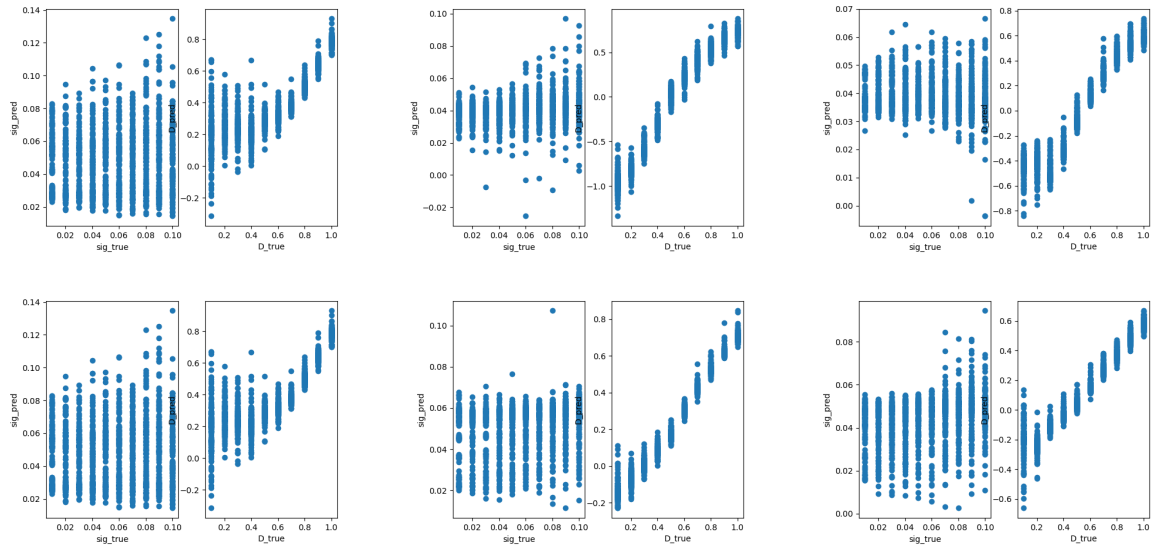

Figure 3.4: Plotting actual vs. predicted values with differing models. Top Row: 100 iterations, 32 batch size; 100 iterations, 16 batch size; 100 iterations, 64 batch size ; 100 iterations, 32 batch size; 130 iterations, 16 batch size; 200 iterations, 64 batch size ; The best model is the one on the upper left.

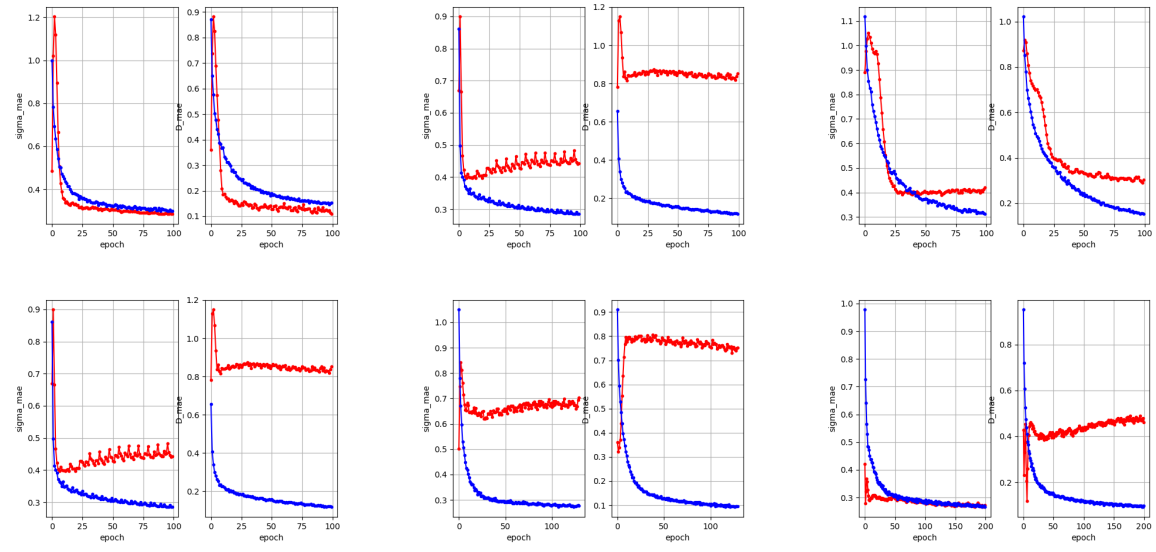

Figure 3.5: Mean Average Error with differing models. Top Row: 100 iterations, 32 batch size; 100 iterations, 16 batch size; 100 iterations, 64 batch size ; 100 iterations, 16 batch size; 130 iterations, 16 batch size; 200 iterations, 16 batch size ; The best model is the one on the upper left.

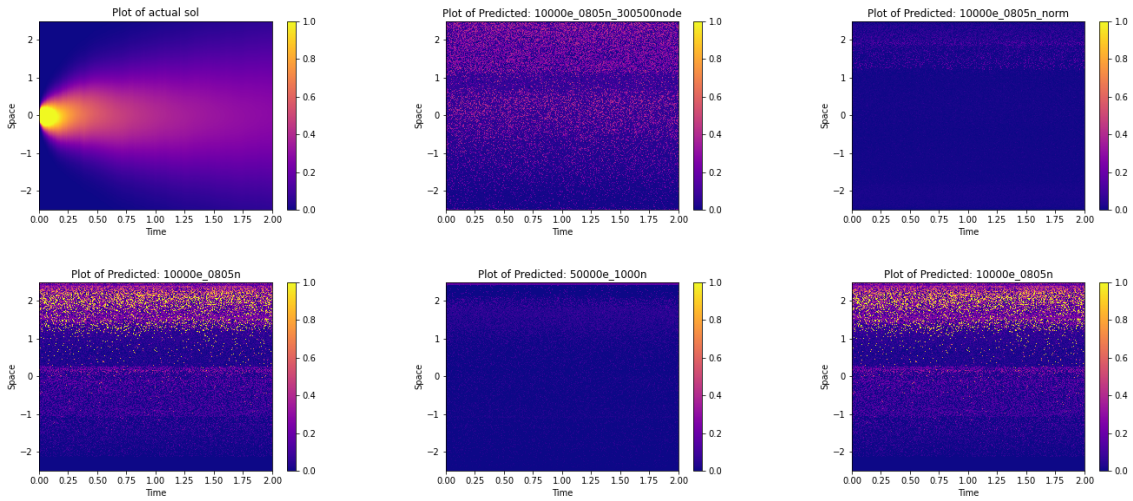

Figure 3.6: Top right image is actual solution of 1-D diffusion equation with the $\sigma = 0.08$, $D = 0.5$. Other images show attempts of recreating image with WGAN with gradient penalty models. The model seems to fix on the values of zero on image generation. Higher values also tend to be generated towards the top of the graphs rather than towards the middle.

CHAPTER IV

CONCLUSION AND DISCUSSION

While initially this project started off on physics informed partial differential equations, over time the project morphed to what it is today. The CNN did relatively well in parameter prediction for the 1-D diffusion equation, and while the GAN results where nothing really special, the author believes that with more time and examination that the direction is a promising one.

The generated images of the heat equation was a great start. Eventually, the idea would be not only to work with 1-dimensional heat equations, but also with multi-dimensional equations as well. It was decided to start small, and then build from the results. We could also look into experimenting with other types of stochastic equations, as well as different parameters for the stochastic and diffusion terms.

In terms of the CNN, the results were positive in that the D parameter of the inverse problem-solving can be estimated well with some of our models. There is reason to suspect that the models have difficulty training on parameters of smaller values, as noted in the results section. As aforementioned, the model has difficulty estimating parameters when their values are smaller than .1. We can also look into restricting the CNN to generate only positive terms in generation.

The results of the WGAN were lackluster, due to the inability of the models to generate an image that resembled the 1-D heat equation. There are paths to explore in terms of improving the model, such as changing the type of loss of the model, to changing the scope of input into the model. We currently pass one array, so it may be feasible to pass multiple arrays in hopes of a better output.

The overall goal of the research was to perform parameter estimation, and with estimated parameters, create a generator model based on the given stochastic differential equation and parameters. While a portion of our results were successful, for the generation we can continue with the

work of GANS, and perhaps look into different types of deep learning models such as recurrent neural networks. Eventually, once we hone in on the models, we intend to experiment with more types of stochastic differntial equations as well.

REFERENCES

Albawi, Saad, Tareq Abed Mohammed, and Saad Al-Zawi (2017). "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

Alzubaidi, Laith et al. (2021). "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". In: *Journal of Big Data* 8.1, p. 53. DOI: 10.1186/s40537-021-00444-8. URL: https://doi.org/10.1186/s40537-021-00444-8.

Bahuguna, D. and J. Dabas (2008). "Partial functional differntial equation with and integral condition and applications to population dynamics". In: *Nonlinear Analysis: THeory, Methods & Applications* 69.8, pp. 2623–2635.

Bressan, Rodrigo (2020). *Building a multi-output Convolutional Neural Network with Keras*. URL: %5Curl%7Bhttps://towardsdatascience.com/building-a-multi-output-convolutional-neural-network-with-keras-ed24c7bc1178%7D.

Brownlee, Jason (2019). *Generative Adversarial Networks with Python*. 1.5. Machine Learning Mastery.

"Coefficient of Determination, R-squared" (n.d.). In: (). URL: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html.

Flores, Stephanie et al. (2020). "Exact and Numerical Solution of Stochastic Burgers Equations with Variable Coefficients". In: *Discrete and Continuous Dynamical System Series S* 13.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. Cambridge, Massachusetts, London, England: The MIT Press.

Gulrajani, Ishaan et al. (2017). *Improved Training of Wasserstein GANs*. arXiv.

Handel, Ramon van (n.d.). "Stochastic Calculus, Filtering, and Stochastic Control". In: (). URL: https://byjus.com/jee/differential-equations/.

Kumar, V.V.K Srinivas (n.d.). “The Heat Equation”. In: (). URL: https://web.iitd.ac.in/~vvksrini/Oldhomepage/1d-fund-heat.pdf.

LeCun, Yann, Yoshua Benglo, and Geoffrey Hinton (2015). “Deep Learning”. In: *Nature* 521, pp. 436–444.

Liu, Weibo et al. (2017). “A survey of deep neural network architectures and their applications”. In: *Neurocomputing* 234, pp. 11–26.

Lysaker, M., A. Lundervold, and Xue-Cheng Tai (2003). “Noise removal using fourth-order partial differential equation with applications to medical magnetic resonance images in space and time”. In: *IEEE Transactions on Image Processing* 12.12, pp. 1579–1590. DOI: 10.1109/TIP.2003.819229.

Mwiti, Derrick (n.d.). *Introduction to Generative Adversarial Networks (GANs): Types , Applications, and Implementations*. https://heartbeat.fritz.ai/introduction-to-generative-adversarial-networks-gans-35ef44f21193.

Øksendal, Bernt (1998). *Stochastic Differential Equations- An Introduction with Applications*. University of Oslo, Box 1053, Blindern, N-0316 Oslo, Norway: Springer.

Paul, Wolfgang and Jörg Bashchnagel (2000). *Stochastic Processes*. Switzerland: Springer.

Qi, Jun et al. (2020). “On Mean Absolute Error for Deep Neural Network Based Vector-to-Vector Regression”. In: *IEEE Signal Processing Letters* 27, pp. 1485–1489. DOI: 10.1109/LSP.2020.3016837.

Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis (2017). *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. arXiv.

Sarle, Warren S. (1994). *Neural Networks and Statistical Models*.

Siemrod, Marshall (1976). “A hereditary partial differential equation with applications in the theory of simple fluids”. In: *Archive for Rational Mechanics and Analysis* 62, pp. 303–321.

“The Diffusion Equation” (2018). In: URL: https://www.uni-muenster.de/imperia/md/content/physik_tp/lectures/ws2016-2017/num_methods_i/heat.pdf.

Trench, William F. (2013). *Elementary Differential Equations with Boundary Value Problems*. Trinity Unversity, San Antonio, Texas, USA.

"What is the Convolutional Neural Network Architecture?" (2020). In: URL: https://www.analyticsvidhya.com/blog/2020/10/what-is-the-convolutional-neural-network-architecture/.

Yang, Liu, Dongkun Zhang, and George Em Karniadakis (2018). *Physics-Informed Generative Adversarial Networks for Stochastic Differential Equations*. arXiv.

APPENDIX A

APPENDIX A

CNN CODE

1.1 CNNCode

```
### Preliminaries

import glob
import pathlib
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from keras.constraints import nonneg
from sklearn.preprocessing import MinMaxScaler
import random
random.seed(123)

#tf.debugging.disable_traceback_filtering()

TRAIN_TEST_SPLIT = 0.7
```

```
IM_WIDTH = IM_HEIGHT = 198 #255


############### Parameters
test_batch_size = 32
num_model_runs = 5
init_lr = 1e-7
epochs = 200
batch_size = 32
valid_batch_size = 32




model_name = "130_iter_mm_lin_lr7_bs32_lds_4c"
data_dir = '/Users/stephanieflores/PycharmProjects/research_proj/
    images/u_trim_50'
data_dir = pathlib.Path(data_dir)

#####################################
#Parse Dataset function
# Read and label the data properly

def parse_dataset(dataset_path, ext='png'):
    def parse_info_from_file(path):
        try:
            filename = os.path.split(path)[1]
            filename = os.path.splitext(filename)[0]
            fiter, sig_val, D_val = filename.split('_')
```

```
            return int(fiter), float(sig_val), float(D_val)
        except Exception as ex:
            return None, None, None

    files = glob.glob(os.path.join(dataset_path, "*/*.%s" % ext))

    records = []
    for file in files:
        info = parse_info_from_file(file)
        records.append(info)

    df = pd.DataFrame(records)
    df['file'] = files
    df.columns = ['fiter', 'sigma', 'D', 'file']
    df = df.dropna()

    return df


df = parse_dataset(data_dir)
df.head()


#####################################
# Data Generator Class
# Create a data generator
```

```
from PIL import Image

class UDataGenerator():
    '''Generates data for the U dataset'''
    def __init__(self, df):
        self.df = df

    def generate_split_indexes(self):
        p = np.random.permutation(len(self.df))
        train_up_to = int(len(self.df) * TRAIN_TEST_SPLIT)
        train_idx = p[:train_up_to]
        test_idx = p[train_up_to:]

        train_up_to = int(train_up_to * TRAIN_TEST_SPLIT)
        train_idx, valid_idx = train_idx[:train_up_to], train_idx
    [train_up_to:]

        self.max_sigma, self.min_sigma = (self.df['sigma'].max(),
     self.df['sigma'].min())
        self.max_D, self.min_D = (self.df['D'].max(), self.df['D'
    ].min())

        return train_idx, valid_idx, test_idx

    def preprocess_image(self, img_path):
        im = Image.open(img_path)
        im = im.resize((IM_WIDTH, IM_HEIGHT))
```

```
        im = np.array(im) / 255.0

        return im

    def generate_images(self, image_idx, is_training, batch_size
=16):
        images, s, D = [], [], []
        while True:
            for idx in image_idx:
                equat = self.df.iloc[idx]

                sigma_v = equat['sigma']
                D_v = equat['D']
                file_v = equat['file']

                im = self.preprocess_image(file_v)

                self.sig_denom = self.max_sigma - self.min_sigma
                self.D_denom= self.max_D - self.min_D

                #s.append(sigma_v / self.max_sigma)
                #D.append(D_v / self.max_D)

                s.append(((sigma_v - self.min_sigma)/self.
sig_denom))
                D.append(((D_v - self.min_D)/self.D_denom))
```

```
                images.append(im)

                # yielding condition
                if len(images) >= batch_size:
                    yield np.array(images), [np.array(s), np.
    array(D)]
                    images, s, D = [], [], []

            if not is_training:
              break


data_generator = UDataGenerator(df)
train_array, valid_array, test_array = [], [], []

for i in range(0,num_model_runs):
    train_idx, valid_idx, test_idx = data_generator.
    generate_split_indexes()
    train_array.append(train_idx)
    valid_array.append(valid_idx)
    test_array.append(test_idx)

#####################################
# Multi-Output Class

from keras.models import Model
from keras.layers import BatchNormalization
```

```
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Dropout
from keras.layers.core import Lambda
from keras.layers.core import  Dense
from keras.layers import Flatten
from keras.layers import Input
import tensorflow as tf

class UMultiOutputModel():
    """Contain 2 branches, one for signma, one for D"""
    def make_default_hidden_layers(self,inputs):
        """Conv2D -> BatchNormalization -> Pooling -> Dropout"""

        x = Conv2D(16,(3,3), padding="same")(inputs)
        x = Activation("relu")(x)
        x = BatchNormalization(axis=-1)(x)
        x = MaxPooling2D(pool_size=(3,3))(x)
        x = Dropout(0.25)(x)

        #x = Conv2D(32, (3, 3), padding="same")(x)
        #x = Activation("relu")(x)
        #x = BatchNormalization(axis=-1)(x)
        #x = MaxPooling2D(pool_size=(2, 2))(x)
        #x = Dropout(0.25)(x)
```

```
        #x = Conv2D(32, (3, 3), padding="same")(x)
        #x = Activation("relu")(x)
        #x = BatchNormalization(axis=-1)(x)
        #x = MaxPooling2D(pool_size=(2, 2))(x)
        #x = Dropout(0.25)(x)

        return x

    def build_sigma_branch(self, inputs):
        x = self.make_default_hidden_layers(inputs)

        x = Flatten()(x)
        x = Dense(32)(x)
        x = Activation("relu")(x)
        x = BatchNormalization()(x)
        x = Dropout(0.25)(x)
        x = Dense(1)(x)
        x = Activation("linear", name = "sigma_output")(x)

        return x

    def build_D_branch(self, inputs):
        x = self.make_default_hidden_layers(inputs)

        x = Flatten()(x)
        x = Dense(32)(x)
        x = Activation("relu")(x)
```

```
        x = BatchNormalization()(x)
        x = Dropout(0.25)(x)
        x = Dense(1)(x)
        x = Activation("linear", name="D_output")(x)

        return x

    def assemble_full_model(self, width, height):
        input_shape = (height, width, 4)

        inputs = Input(shape=input_shape)

        sigma_branch = self.build_sigma_branch(inputs)
        D_branch = self.build_D_branch(inputs)

        model = Model(inputs=inputs,
                      outputs=[sigma_branch, D_branch],
                      name="pde_net")

        return model


try:
    os.makedirs('./%s/arrays' % model_name)
except OSError as error:
    print(error)

sig_mae, sig_vmae, D_mae, D_vmae = [],[],[],[]
```

```
m_loss, val_loss = [], []

opt = tf.keras.optimizers.Adam(learning_rate=init_lr, decay=
    init_lr / epochs) #, decay=init_lr / epochs

for i in range(0, len(test_array)):

    model_f = UMultiOutputModel().assemble_full_model(IM_WIDTH,
    IM_HEIGHT)

#####################################
# Model Parameters
# Train the model

    model_f.compile(optimizer=opt,
                    loss= {'sigma_output':'mse',
                           'D_output':'mse'},
                    loss_weights={'sigma_output':1.4, #1.7
                                  'D_output':1.7}, #1.7
                    metrics={'sigma_output':'mae',
                             'D_output':'mae'})

    from keras.callbacks import ModelCheckpoint

    train_gen = data_generator.generate_images(train_array[i],
    is_training=True, batch_size=batch_size)
```

```
 valid_gen = data_generator.generate_images(valid_array[i],
is_training=True, batch_size=valid_batch_size)

 callbacks = [
     ModelCheckpoint("./model_checkpoint", monitor='val_loss')
 ]

 history = model_f.fit(train_gen,
                                    steps_per_epoch=len(
train_array[i])//batch_size,
                                    epochs=epochs,
                                    callbacks=callbacks,
                                    validation_data=valid_gen,
                                    validation_steps=len(
valid_array[i])//valid_batch_size)

 ######################################
 # SIGMA and D Absolute Error

 sig_mae.append(history.history['sigma_output_mae'])
 sig_vmae.append(history.history['val_sigma_output_mae'])
 D_mae.append(history.history['D_output_mae'])
 D_vmae.append(history.history['val_D_output_mae'])

 hist_df = pd.DataFrame(history.history)

 # or save to csv:
```

```
    hist_csv_file = './%s/history_%s.csv' % (model_name,str(i))
    with open(hist_csv_file, mode='w') as f:
        hist_df.to_csv(f)

np.save('./%s/arrays/sig_mae.npy' % model_name, sig_mae)
np.save('./%s/arrays/sig_vmae.npy' % model_name, sig_vmae)
np.save('./%s/arrays/D_mae.npy' % model_name, D_mae)
np.save('./%s/arrays/D_vmae.npy' % model_name, D_vmae)



####################################
# Evaluate on a Test Set

sig_true_arr, sig_pred_arr, D_true_arr, D_pred_arr = [],[],[],[]



from sklearn.metrics import r2_score
sig_r2, D_r2 = [],[]

for i in range(0,len(train_array)):

    test_generator = data_generator.generate_images(test_array[i
   ],
                                                    is_training=
   False,
                                                    batch_size=
   test_batch_size
```

```
                                                    )

 sig_pred, D_pred = model_f.predict(test_generator,
                                                    steps=len(
test_array[i])//test_batch_size)

 test_generator = data_generator.generate_images(test_array[i
], is_training=False, batch_size=test_batch_size)

 images, sig_true, D_true = [], [], []

 for test_batch in test_generator:
   image = test_batch[0]
   labels = test_batch[1]

   images.extend(image)
   sig_true.extend(labels[0])
   D_true.extend(labels[1])

 sig_true = np.array(sig_true)
 D_true = np.array(D_true)

 #sig_true = sig_true * data_generator.max_sigma
 #sig_pred = sig_pred * data_generator.max_sigma

 #D_true = D_true * data_generator.max_D
 #D_pred = D_pred * data_generator.max_D
```

```
    sig_true = sig_true * (data_generator.sig_denom) +
   data_generator.min_sigma
    sig_pred = sig_pred * (data_generator.sig_denom) +
   data_generator.min_sigma

    D_true = D_true * (data_generator.D_denom) + data_generator.
   min_D
    D_pred = D_pred * (data_generator.D_denom) + data_generator.
   min_D

    sig_true_arr.append(sig_true)
    sig_pred_arr.append(sig_pred)
    D_true_arr.append(D_true)
    D_pred_arr.append(D_pred)

    sig_r2.append(r2_score(sig_true, sig_pred))
    D_r2.append(r2_score(D_true, D_pred))
    #print('R2 score for D: ', )
    #print(sig_pred)
    #print(D_pred)

np.save('./%s/arrays/D_pred_arr.npy' % model_name, D_pred_arr)
np.save('./%s/arrays/D_true_arr.npy' % model_name, D_true_arr)
np.save('./%s/arrays/sig_r2.npy' % model_name, sig_r2)
np.save('./%s/arrays/D_r2.npy' % model_name, D_r2)
```

```
    #rint('hello')


for i in range(0,len(sig_mae)):
    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.plot(np.arange(len(sig_vmae[i])), sig_vmae[i], marker='.'
   , c='red')
    ax1.plot(np.arange(len(sig_mae[i])), sig_mae[i], marker='.',
   c='blue')
    ax1.grid()
    plt.setp(ax1, xlabel='epoch', ylabel='sigma_mae')

    ax2.plot(np.arange(len(D_vmae[i])), D_vmae[i], marker='.', c=
   'red')
    ax2.plot(np.arange(len(D_mae[i])), D_mae[i], marker='.', c='
   blue')
    ax2.grid()
    plt.setp(ax2, xlabel='epoch', ylabel='D_mae')

    try:
        os.mkdir('./%s/metric' % model_name)
    except OSError as error:
        print(error)

    plt.savefig('./%s/metric/model_iter_%s.png' % (model_name,str
   (i)))
    plt.close()
```

```
for i in range (0,len(sig_true_arr)):
    try:
        os.mkdir('./%s/act_pred' % model_name)
    except OSError as error:
        print(error)
    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.scatter(sig_true_arr[i], sig_pred_arr[i])
    #ax1.set_ylim([.01, .1])
    plt.setp(ax1, xlabel='sig_true', ylabel='sig_pred')

    ax2.scatter(D_true_arr[i], D_pred_arr[i])
    #ax1.set_ylim([.1, 1])
    plt.setp(ax2, xlabel='D_true', ylabel='D_pred')
    plt.savefig('./%s/act_pred/m_iter_%s.png' % (model_name,str(i
   )))
    plt.close()
```

APPENDIX B

APPENDIX B

U SOLUTION CODE

2.1 U solution Code

```
import os

os.getcwd()
curr_direct = '/Users/stephanieflores/PycharmProjects/
   research_proj'
os.chdir('/Users/stephanieflores/PycharmProjects/research_proj')

"""Files of the arrays/images are in the form #of pair, sigma, D
Directory files are named as sigma, D"""

# File: brownian.py

from math import sqrt
import math
import matplotlib
from brownian_function import brownian
from create_im_directories import create_im_directories
```

```
import scipy.integrate as integrate
import numpy as np
from pylab import plot, show, grid, xlabel, ylabel
import matplotlib.pyplot as plt

################################
# Define Parameters

# The Wiener process parameter.
#sigma = 1 # Wiener Coefficient
#d_array = np.linspace(.1,1,10) #10 (x(.1,1,10))
#sig_array = np.linspace(.01,.1,10) #10
d_array = np.array([.1, .5, 1.0]) #10 (x(.1,1,10))
sig_array = np.array([.01, .05, .1])
#D = 2 # Diffusion Coefficient
D = 2
sigt = 1
# Total time.
T = 2 # 1
# Number of time steps.
N = 500
# Time step size
dt = T / N
# Number of realizations to generate.
m = 1
# Create an empty array to store the realizations.
x = np.empty((m, N + 1))
```

```
# Initial values of x.
x[:, 0] = 0
M = 1000 # Number of space steps
colors = 'plasma'

#xrange = np.append(np.arange((-e_point), e_point, dx), [1])
trange = np.linspace(0.0, N * dt, N + 1)

################################################

# sigma will change/non-constant
def create_k(map_name,sigma, xrange, visualize = False, save =
    False):
    '''Creates the array x + sigma * wt'''
    # Visualize Wt
    brownian(x[:, 0], N, dt, sigma, out=x[:, 1:])
    if visualize is True:
        for k in range(m):
            plot(trange, x[k])
        xlabel('t', fontsize=16)
        ylabel('x', fontsize=16)
        grid(True)
        show()
    b_calc = np.repeat(x, M + 1, axis=0) # Wt
    x_val = np.repeat(xrange, N + 1, axis=0).reshape([M + 1, N +
    1]) #x
    K_array = x_val + b_calc
```

```
    if save is True:
        np.save('./arrays/k/' + map_name, K_array)
    return K_array

## You must get xrange, trange, K in which each row is for an x
    from 0 to 1 and each column is for t from 0 to 1
#  and K=x+sigma*W_t

def plot_k(array,map_name, xrange,impath, show_plot = False):
    fig, ax = plt.subplots()
    cmap = plt.get_cmap(colors)
    im = ax.pcolormesh(trange,xrange, array, cmap='viridis', norm
    = matplotlib.colors.Normalize(vmin=0, vmax = 1))
    plt.xlabel('Time')
    plt.ylabel('Space')
    plt.title('Plot of K (x + sigma*W_t): ' + map_name)
    plt.colorbar(im)
    plt.savefig('./images/k_thesis/' + map_name +'.png')
    if show_plot is True:
        plt.show()
    plt.close('all')

def find_u(array, map_name,dtil):
#### Solving the series using K
    u_array = np.empty((1001,501))
    for i in range(1,trange.shape[0]):
        const = 1 / ((4 * math.pi * dtil * trange[i]) ** (1 / 2))
```

```
        for j in range(0,array.shape[0]):
            exp_const = math.exp(-((array[j,i])**2)/(4*dtil*
    trange[i]))
            u_array[j,i] = const*exp_const
     u_array[0,0] = 0
     #np.save('./arrays/usol/' + map_name, u_array)
     return u_array

def plot_u(array,map_name,xrange,impath, show_plot = False):
     fig, ax = plt.subplots()
     cmap = plt.get_cmap(colors)
     im = ax.pcolormesh(trange,xrange,array, cmap=cmap, norm =
    matplotlib.colors.Normalize(vmin=0, vmax = 1) ,shading='
    nearest')
     #plt.xlabel('Time')
     #plt.ylabel('Space')
     #plt.title('Plot of u(x,t): ' + map_name)
     #plt.colorbar(im)
     plt.axis('off')
     plt.savefig(impath + '/' + map_name + '.png', bbox_inches='
    tight')
     if show_plot is True:
         plt.show()
     plt.clf()


def __main__():
```

```
    for k in sig_array:
        sig = k
        for j in d_array:
            D = j
            #e_point = 3 * math.sqrt(2 * D)
            #dx = ((e_point * 2)) / M
            xr = np.linspace(-2.5, 2.5, M + 1)
            D_tilda = D - (1 / 2) * (sig ** 2)
            pathc = '/Users/stephanieflores/PycharmProjects/
research_proj/images/u_thesis/{sigp:.2f}_{dp:.2f}' #changed
testing/u
            pathc = pathc.format(sigp = k, dp = j)
            if os.path.isfile(pathc) is False:
                create_im_directories(pathc)
            for i in range(0,1):
                k_map = '{read}_{rsig:.2f}_{rd:.2f}'
                k_map = k_map.format(read = i,rsig = sig,rd = D)
                uxt_map = '{read}_{rsig:.2f}_{rd:.2f}'
                uxt_map = uxt_map.format(read=i, rsig=sig, rd=D)
                K = create_k(k_map,sig,xr)
                plot_k(K,k_map, xr,pathc)
                u_sol = find_u(K,uxt_map, D_tilda)
                plot_u(u_sol,uxt_map,xr,pathc)


__main__()


def test_gen(sigma_val, D_val):
```

```
D_tilda = D_val - (1 / 2) * (sigma_val ** 2)
k_map = 'k_test s: {rsig:.2f}, D: {rd:.2f}'
k_map = k_map.format(rsig=sigma_val, rd=D_val)
uxt_map = 'uxt_test s: {rsig:.2f}, D: {rd:.2f}'
uxt_map = uxt_map.format(rsig=sigma_val, rd=D_val)
K = create_k(k_map, sigma_val)
plot_k(K, k_map, show_plot=True)
u_sol = find_u(K, uxt_map, D_tilda)
plot_u(u_sol, uxt_map, show_plot=True)
```

BIOGRAPHICAL SKETCH

Stephanie Flores graduated from Donna High School in 2015, where she earned the Coca-Cola First Generation Scholarship. She has since then earned a Bachelor's Degree in Applied Mathematics in 2019 and a Master's Degree in Applied Statistics and Data Science in 2022 from the University of Texas of Rio Grande Valley. Upon the beginning of her graduate career, Stephanie had earned the Presidential Grant in Research Assistantship.

Stephanie has participated in the Louis Stokes Alliances for Minority Participation in 2017 with research in Integer Partitions with Dr. Kronholm, and has researched on behalf of the National Science Foundation in REU in 2018 on the topic of Stochastic Differential Equations with Dr. Oraby, Dr. Suazo, and Dr. Yoon. She had interned for the Department of Defense during 2019 on a data analyst project and at Apple as a data science intern during 2021.

To contact Stephanie, feel free to message her with the following email:

stephflores721@gmail.com.