Project 3: Differential Equations and Neural Networks

Josef Hisanawi, Higinio Joaquin Perez Parra, Mustafa Najjar University of Oslo

The Heat equation was solved using analytical methods, neural networks and finite differences. The results from the neural networks proved to be very similar to the analytical with an MSE of about 0.053. The finite differences method performed even better with an extraordinarily low MSE value of 10^{-9}. In addition, eigenvalues and eigenvectors were computed using the Jacobi method compared to Neural Networks' solution, the results were accurate to 10^{-7}. The bias-variance tradeoff analysis shows that the random forest provided the best result with an MSE of 0.1489 while decreasing both bias and variance.

Introduction

As Galileo Galieli once stated: "The universe is written in the language of mathematics, and the letters are triangles, circles and other geometrical figures, without which means it is humanly impossible to comprehend a single word". In fact, mathematics is the basis of all human knowledge of natural sciences, from the use of statistics in medical research to test the efficacy of a certain drug to differential equations used in physics to describe the motion and laws of the universe. It's the latter that we are going to focus on this report.

Galileo lacked the knowledge of differential equations, if he knew about differential calculus at his time, he would probably have stated that: "The universe is written in the language of differential equations". Differential equations (or DE) become extremely use full when dealing with the evolution of a certain system through time. In deed, DE describe how different magnitudes of a system depend on each other and their time evolution. Knowing the set of DE that govern a system and the state of the system at a specific moment in time, the state of the system is then univocally determined at any moment in time, future or even past. Among these fundamental differential equations, we can recall, Newton's second law of motion which includes the acceleration as a second derivative of position in respect to time, and is able to predict the motion of all bodies if we know all the forces that are involved:

$$\sum \vec{F} = m\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2}$$

Or the Euler-Lagrange equation that is used in analytical mechanics to obtain a set of DE that describes the motion of a system through time, if we know its Lagrangian:

$$L = T - V$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0$$

Or the Maxwell equations that are the fundamental equations in electromagnetism, that describes the source and the interaction between the magnetic and electric field. And in conjunction with the Lorentz force describes all the electromagnetic effects.

$$\vec{\nabla}.\vec{E} = \frac{\rho}{\epsilon_0}$$

$$\vec{\nabla}.\vec{B} = 0$$

$$\vec{\nabla} \times \vec{E} = -\frac{\partial \vec{B}}{\partial x}$$

$$\vec{\nabla} \times \vec{B} = \mu_0 \vec{J} + \mu_0\epsilon_0\frac{\partial \vec{E}}{\partial x}$$

Or the Navier-Stokes equations that are fundamental in the field of fluids mechanics. Or even the Schrödinger equation which describes the evolution of a quantum system in time if no measurement is performed on it:

$$i\hbar\frac{\partial}{\partial t}|\Psi_n(t)\rangle = \hat{H}|\Psi_n(t)\rangle$$

Since some integration is needed in order to obtain an analytical solution, this solution is only available in very simple cases that are usually studied in a physics class. Real problems are nearly all ways impossible to solve analytically. That is why we often use numerical methods to solve our differential equations. The most known method is Euler's method, that uses the definition of the derivative.

However, to solve more sophisticated problems we are given the finite differences method. This method enables us to approximate first and second order derivatives through the truncation of the Taylor expansion of the given function. Although these methods are quite simple to understand and implement, they suffer from stability issues, making some problems impossible to solve.

To avoid this stability problem, we can use neural networks to provide us the solution to some differential equations. This is possible due to the universal approximation theorem, which states that a neural network may approximate any function at a single hidden layer along with one input and output layer to any given precision. With some clever manipulation, we can use this fact to use neural networks to solve differential equations.

The aim of this report is to study the accuracy and advantages of using finite differences methods and neural networks algorithms to numerically solve differential equations. In particular, we will study how these methods perform on the one-dimensional heat equation. As a side quest, we will investigate and evaluate Neural Networks performance on calculating the eigenvalues and eigenvectors of a 6x6 symmetric real matrix and compare to the standard Jacobi method of finding it.

In addition to the previous analysis, some research was also done on the bias-variance tradeoff. The aim of which is to analyse different sets of algorithms, linear regression, neural networks and ensemble methods. The focus will be put on which methods perform the best by yielding lowest error values, including MSE, bias and variance.

Theory

In general, a differential equation is an equation that contains some function $f(x) : \mathbb{R}^n \to \mathbb{R}$ and its derivatives,

$$g(x, f(x), \nabla f(x), \nabla^2 f(x), \ldots) = 0$$

We can subdivide a differential equation into two sub-categories. A differential equation where $f : \mathbb{R} \to \mathbb{R}$ and where $n = 1$ is called an Ordinary Differential Equation (ODE), and only contains total derivatives of f, since it can only contain total derivatives of f. A differential equation where $f : \mathbb{R}^n \to \mathbb{R}$ and where $n > 1$, can contain partial derivatives, thus it is called a Partial Differential Equation (PDE). As a note, the order of a differential equation is given by the highest order derivative present in the equation.

The order of an ordinary differential equation (ODE) determines the number of degrees of freedom that can be obtained through integration constants. These degrees of freedom result in the existence of families of solutions rather than a single, exact solution. However, if we have additional information about the starting state of the system, such as initial conditions or boundary conditions, we may be able to find a specific solution.

Similarly, partial differential equations (PDEs) can be solved if we have sufficient information about the solution at the initial time or at the boundaries of the domain of the equation. This information may take the form of the solution itself (in the case of a Dirichlet problem) or the normal derivative of the solution (in the case of a Neumann problem). In general, we need as many conditions as the order of the differential equation in order to solve it.

Heat equation analytical solution

We are going to solve the homogeneous heat equation with homogeneous contour conditions (CC) and some initial condition (IC) using the variable separation method for a rod segment.

We consider the following problem:

$$PDE : u_{xx} - u_t = 0$$

$$x \in [0, L] = [0, 1]$$

$$IC : u(x, 0) = sin(\pi x)$$

$$CC \begin{cases} u(0, t) = 0 \\ u(L, t) = 0 \end{cases}$$

We assume that we can write the solution in the following form:

$$u(x, t) = X(x)T(t)$$

If we substitute that in our original PDE we get:

$$X''T - XT' = 0$$

$$\frac{X''}{X} = \frac{T'}{T}$$

As $\frac{X''}{X}$ only depends on x and $\frac{T'}{T}$ only depends on t, the only possibility for the equation to satisfy the previous equality is that they are both equal to some constant that we are going to call $-\lambda^2$. So:

$$\frac{X''}{X} = \frac{T'}{T} = -\lambda^2$$

With this we get two ordinary differential equations (ODE) that we have to solve, and that is:

$$X''(x) + \lambda^2 X(x) = 0$$
$$T'(t) + \lambda^2 T(t) = 0$$

In this case, the ODE are lineal with constant coefficients, and as general solution, we get:

$$X(x) = Acos(\lambda x) + Bsin(\lambda x)$$
$$T(t) = Ce^{-\lambda^2 t}$$

If we impose our contour conditions we have that:

$$X(0) = 0 \Rightarrow A = 0$$

$$X(L) = 0 \Rightarrow \lambda L = n\pi \rightarrow \lambda = \frac{n\pi}{L}$$

$$n \in \mathbb{N}$$

With that we get a fundamental solution for each value of n, in the form:

$$u_n(x,t) = B_n sin\left(\frac{n\pi}{L}x\right) e^{-\left(\frac{n\pi}{L}\right)^2 t}$$

So the most general solution would be a lineal combination of all the fundamental solutions for the different n:

$$u(x,t) = \sum_{n=1}^{\infty} u_n(x,t) = \sum_{n=1}^{\infty} B_n sin\left(\frac{n\pi}{L}x\right) e^{-\left(\frac{n\pi}{L}\right)^2 t}$$

We can get the constant terms of B_n by imposing our initial condition:

$$f(x) = u(x,0) = \sum_{n=1}^{\infty} B_n sin\left(\frac{n\pi}{L}x\right)$$

This last expression is like the odd development of the Fourier function f(x) between -L and L, so we can obtain B_n as:

$$B_n = \frac{1}{L}\int_0^L sin\left(\frac{n\pi}{L}x\right) f(x) dx$$

Now, we substitute the values of our problem so that: $L = 1$ and $f(x) = sin(\pi x)$, and we get:

$$B_n = \int_0^1 sin(n\pi x)\, sin(\pi x)\, dx = 2\delta_{1n}$$

This expression can be found using the Euler's orthogonality relations for trigonometric functions. Now we recall our general solution and we substitute the value of B_n:

$$\begin{aligned} u(x,t) &= \sum_{n=1}^{\infty} B_n sin\left(\frac{n\pi}{L}x\right) e^{-\left(\frac{n\pi}{L}\right)^2 t} \\ &= \sum_{n=1}^{\infty} \delta_{1n} sin(n\pi x)\, e^{-(n\pi)^2 t} \\ &= sin(\pi x)\, e^{-\pi^2 t} \end{aligned}$$

And like this we have solve our problem of finding an analytical solution to the homogeneous heat equation.

Finite differences

The finite differences method we will use in this report is based on the truncation of the Taylor expansion of a certain function to approximate the derivative at hand.

Let us consider a function $f : \mathbb{R} \rightarrow \mathbb{R}$. Its Taylor expansion around a point $x + h$, where h is as small as possible is given as:

$$f(x+h) = f(x) + f^{'}(x)h + f^{''}(x)h^2 + ...$$

Considering only the first order term, we get an approximation for the first derivative

$$f^{'}(x) = \frac{f(x+h) - f(x)}{h}$$

Considering a second order approximation and using the result above for the first derivative, we can get an approximation of the second derivative

$$f^{''}(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

By having an equally spaced, discretized grid of $N+1$ points over $x \in [0, X]$, $x_0, x_1, x_2, \ldots, x_N$, given by

$$x_i = ih = i\frac{X}{N}$$

$i = 0, \ldots, N$, we can write the above expressions as

$$f^{'}(x) \approx \frac{f(x_{i+1}) - f(x_i)}{h}$$
$$f^{''}(x) \approx \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2}$$

Numerical methods for solving differential equations involve creating a grid of discrete points, called nodes, to approximate the solution of the equation. These methods have a certain range of values, called the stability zone, within which they will converge to a stable solution. To use these methods effectively, it is important to ensure that the grid is set up in such a way that it satisfies the given initial conditions and also falls within the stability zone. This is important because if the grid does not satisfy these requirements, the method may not converge to a stable solution.

Despite this, these methods can be used to solve a wide variety of systems, including those in which the temperature at the boundaries varies according to a non-differentiable function. It is worth noting that stability and stability requirements are important considerations when using numerical methods for differential equations, but we will not delve into them in detail in this report.

One of the main issues we previously mentioned is the stability criterion, which requires us to have a more finely discretized time domain compared to our space domain. This means that if we want a detailed space domain, our simulation will take a longer time to run.

Neural Network for Differential Equations

In a neural network, the cost function plays a key role in guiding the training process. It helps determine how well the network is performing in relation to the desired output. One way to choose a cost function is to define it as the square of a function g, which can be written in the form of a differential equation as:

$$g(x, f(x), \nabla f(x), \nabla^2 f(x), \dots) = 0$$

where x is a vector containing the parameters for the problem, and P represents the parameters of the network, such as weights and biases.

To find an approximation of the true solution, the neural network uses a trial solution that combines functions that satisfy certain conditions, such as initial and boundary conditions, with functions determined by the neural network itself. This trial solution is initialized with randomly assigned weights and biases, and the gradient of the cost function is calculated using high precision numerical differentiation with respect to P. The weights and biases are then adjusted based on this gradient during each epoch of training. The learning rate, which determines the size of these adjustments, can be kept constant or varied according to some predetermined schedule.

After enough epochs, the trial solution should have minimized the cost function, resulting in a good approximation of the true solution. To make predictions with the trained network, the input layer is fed with the appropriate values of x as nodes, and the network produces a single output value, which represents the predicted value of the true solution for those input parameters.

In the case of the one-dimensional heat equation we may write our trial solution as

$$S(x, t, NN(x, t; P)) = h_1(x) + h_2(x, t, NN(x, t; P))$$

with

$$h_1(x) = \sin(\pi x)$$

and

$$h_2(x, t, NN(x, t; P)) = x(1 - x)t \cdot NN(x, t; P)$$

Note that S satisfies the initial- and boundary conditions. This is because

$$S(0, t, NN) = \sin(\pi \cdot 0) + 0 \cdot t \cdot NN(0, t; P) = 0$$
$$S(1, t, NN) = \sin(\pi) + 1 \cdot 0 \cdot t \cdot NN(1, t; P) = 0$$

as given by $u(0, t) = u(1, t) = 0$, and

$$\begin{aligned} S(x, 0, NN) &= \sin(\pi x) + x(1 - x) \cdot 0 \cdot NN(x, 0; P) \\ &= \sin(\pi x) \end{aligned}$$

as given by $u(x, 0) = \sin(\pi x)$.

The trial solution S has the property of being differentiable with respect to time, as well as being twice differentiable with respect to position, which is a property that is also possessed by the actual solution. This construction method guarantees that the trial solution has these properties. These properties of differentiability are important because they allow us to analyze and understand the behavior of the trial solution over time and in different positions, which can help us to better approximate the true solution.

The founders of this method can be found here

Benefits and Shortcomings

The strength of this way of approximating a differential equation lies in its ability to train itself to a sufficient accuracy with very limited data, namely the grid of combinations (inputs) and the target that every point should be zero. Simply by calculating

the loss with the cost function, the model updates itself for the desired amount of epochs.

Another benefit is that it translates very well to problems of higher dimensionality. With traditional methods, going from a 2-dimensional problem to a 3-dimensional problem increases the computational load by one order of magnitude. With the neural network approach it might not be necessary to change the number of epochs or layer design at all. This, however, is very unlikely, some changes might be needed in order to ensure that the approximation is as good as possible.

These benefits do not come without some inconveniences though. In contrast with the finite differences method, the neural network does not allow for hard-coded boundary values. This is a result of the underlying difference between the two methods. The finite differences method calculates the next value based on its surroundings, the neural network tries to guess the underlying function which would solve the problem as a whole. Therefore the boundary conditions and initial conditions needs to be contained in the trial solution, as it would have been in the true, analytical solution. This does restrict the complexity of our conditions.

For instance, solving the heat equation of a one dimensional rod with $u(1,0) = T$ and $u(1,t) = 0$ for $t > 0$ is not possible, due to the discontinuous nature of the temperature applied to the end of the rod.

Additionally, the initial and boundary conditions decide the properties of the functions which comprise S, i.e. $h_1, h_2, \ldots, h_n$. With complicated conditions finding appropriate h_i functions might prove to be quite troublesome.

One advantage of using this method for approximating differential equations is that it can achieve a high level of accuracy with a small amount of data. This data consists of a grid of input combinations and the target value of zero for each point on the grid. The model updates itself through the use of a cost function to calculate the loss and adjusts itself for a specified number of epochs. Another benefit of this method is its scalability to problems with higher dimensions. While traditional methods may require significantly more computational resources to solve a 3-dimensional problem compared to a 2-dimensional problem, this neural network approach may not require any changes to the number of epochs or layer design. However, it is worth noting that some adjustments may be necessary to ensure the best possible approximation.

On the other hand, there are also some limitations to this method. Unlike the finite differences method, this neural network approach does not allow for hard-coded boundary values. This is because the neural network tries to guess the underlying function that would solve the problem as a whole, rather than calculating the next value based on the values around it. As a result, the initial and boundary conditions must be incorporated into the trial solution, similar to how they would be included in the true, analytical solution. This can restrict the complexity of the conditions that can be handled by the model. For example, it would not be possible to use this method to solve the heat equation for a one-dimensional rod with a temperature of T at $u(1,0)$ and a temperature of 0 at $u(1,t)$ for all $t > 0$ due to the discontinuous nature of the temperature at the end of the rod. Additionally, finding appropriate functions $h_1, h_2, \ldots, h_n$ to represent the initial and boundary conditions may be challenging in cases with complex conditions.

More in depth can be found in those citations, which include the researchers who discovered the practice and tested it themselves[6][8][11][12]

Bias-Variance Tradeoff

Bias and variance are both errors that can arise in predictive models. Bias is a measure of the difference between the average prediction of the model and the true value, it is analogous to the standard scientific definition of "accuracy" of the model. A high bias tends to underfit the data and oversimplifies it, leading to a high error. Variance indicates how sensitive the model is and shows the variability of the data and thus the spread. High variance models are typically highly affected by small fluctuations and noise, leading to overfitting.

Variance is in general defined as

$$\begin{aligned} \mathrm{Var}(x) &= \mathbb{E}\left[(x - \mathbb{E}[x])^2\right] && (1)\\ &= \mathbb{E}[x^2] - \mathbb{E}[x]^2 && (2) \end{aligned}$$

The bias of our prediction $\tilde{y}$ can be mathematically written as follows:

$$\begin{aligned} \mathrm{Bias}(\tilde{\boldsymbol{y}}) &= \mathbb{E}[\tilde{\boldsymbol{y}} - f] && (3)\\ &= \mathbb{E}[\tilde{\boldsymbol{y}}] - f && (4) \end{aligned}$$

The MSE, which is also the basis of the cost function for the methods used is defined below:

$$\mathrm{MSE} = C(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right]. \quad (5)$$

We will assume that $y = f(x) + \varepsilon$, where ε is the noise with zero mean and a variance of σ^2. This means that $\mathbb{E}[\varepsilon] = 0$ and $\text{Var}(\varepsilon) = \sigma^2$. We also have that $\mathbb{E}[f] = f$ since f is deterministic. Furthermore, the variance of y can be written as:

$$\begin{aligned}
\text{Var}(y) &= \mathbb{E}\left[(\tilde{\boldsymbol{y}} - \mathbb{E}[\tilde{\boldsymbol{y}}])^2\right] && (6)\\
&= \mathbb{E}\big[(y - f)^2\big] = \mathbb{E}\big[f + \varepsilon - f)^2\big] && (7)\\
&= \mathbb{E}\big[\varepsilon^2\big] && (8)\\
&= \text{Var}[\varepsilon] + \mathbb{E}[\varepsilon]^2 && (9)\\
&= \sigma^2 && (10)
\end{aligned}$$

In the last equality we use the rearranged version of equation 2: $\mathbb{E}[x^2] = \text{Var}(x) + \mathbb{E}[x]^2$. The information above can then be used to decompose the MSE from equation 5:

$$\begin{aligned}
\mathbb{E}\big[(y - \tilde{y})^2\big] &= \mathbb{E}\big[(f + \varepsilon - \tilde{y})^2\big]\\
&= \mathbb{E}\big[(f + \varepsilon - \tilde{y} + \mathbb{E}[\tilde{y}] - \mathbb{E}[\tilde{y}])^2\big]\\
&= \mathbb{E}\big[(f - \mathbb{E}[\tilde{y}])^2\big] + \mathbb{E}[\varepsilon^2] + \mathbb{E}\big[(\mathbb{E}[\tilde{y}] - \tilde{y})^2\big]\\
&\quad + 2\mathbb{E}\big[(f - \mathbb{E}[\tilde{y}])\varepsilon\big] + 2\mathbb{E}\big[\varepsilon(\mathbb{E}[\tilde{y}] - \tilde{y})\big]\\
&\quad + 2\mathbb{E}\big[(\mathbb{E}[\tilde{y}] - \tilde{y})(f - \mathbb{E}[\tilde{y}])\big]\\
&= (f - \mathbb{E}[\tilde{y}])^2 + \mathbb{E}[\varepsilon^2] + \mathbb{E}\big[(\mathbb{E}[\tilde{y}] - \tilde{y})^2\big]\\
&\quad + 2(f - \mathbb{E}[\tilde{y}])\mathbb{E}[\varepsilon] + 2\mathbb{E}[\varepsilon]\mathbb{E}\big[\mathbb{E}[\tilde{y}] - \tilde{y}\big]\\
&\quad + 2\mathbb{E}\big[\mathbb{E}[\tilde{y}] - \tilde{y}\big](f - \mathbb{E}[\tilde{y}])\\
&= (f - \mathbb{E}[\tilde{y}])^2 + \mathbb{E}[\varepsilon^2] + \mathbb{E}\big[(\mathbb{E}[\tilde{y}] - \tilde{y})^2\big]\\
&= (f - \mathbb{E}[\tilde{y}])^2 + \sigma^2 + \text{Var}\left(\tilde{y}\right)\\
&= \text{Bias}(\tilde{y})^2 + \sigma^2 + \text{Var}\left(\tilde{y}\right)
\end{aligned}$$

The decomposition above shows us that to minimize the total error, we need to have both low bias and low variance (σ^2 is the irreducible error), i. e. we want to have a model that captures the trends in the training data while also performing well on test data that have not been included in training.

Ideally, the variance and bias of the models would be as low as possible to ensure both high accuracy and precision, though this is easier said than done as often decreasing the value of one error might increase the value of the other. This can especially be a problem with more complex models that are prone to overfitting. The bias-variance tradeoff is a property of the model that describes the relationship between the bias and variance and can be used to indicate the the complexity at which the bias and variance are lowest.

Methods

Discrete solution to the heat equation

There is another method to find a solution to our problem, and that is using numerical tools and discrete differential equations. So, in order to solve the heat equation that we solve previously, we have to bare in mind our contour conditions:

$$CC \begin{cases} u(0,t) = 0 \\ u(L,t) = 0 \end{cases}$$

And the initial condition:

$$IC : u(x,0) = sin(\pi x)$$

And now, in order to solve the equation we are going to use an explicit method, that is a method that in order to obtain the value of the function at a certain moment in time t+1, we can find it using the known values of the function at a previous moment in time t. Our frist step would be to discretize the heat equation, that is:

$$\frac{\partial^2 u}{\partial x^2} - \frac{\partial u}{\partial t} = 0$$

We use the expression of partial derivatives by their equivalent in finite differences:

$$\frac{\partial u}{\partial t} = \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{\Delta t} + \mathcal{O}\left(\Delta t\right)$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x_{i-1}, t_j) - 2u(x_i, t_j) + u(x_{i+1}, t_j)}{(\Delta t)^2} + \mathcal{O}\left(\Delta t^2\right)$$

After substituting in the original differential equation the discrete expresions of the derivatives we get:

$$u(x_i, t_{j+1}) = u(x_i, t_j) + \frac{\Delta t}{\Delta x^2}\left[u(x_{i-1}, t_j)\right.$$

$$\left. -2u(x_i, t_j) + u(x_{i+1}, t_j)\right.$$

And like this we got our expression for an explicit method to solve the heat equation. This expression allows us to find the values u in each point of the rod for any time, since the values of u at the beginning where given by our initial condition. Now, we may find the values of u at any moment of time if we perform the necessary number of iterations.

We have not discussed about the size of the spatial and time steps, that is Δt and Δx^2. The algorithm that we are going o program, in order to satisfy the

stability criterion for the explicit scheme, requires that Δt and Δx^2 must not be chosen arbitrarily. The algorithm to obtain $u(x_i, t_j)$ is stable when:

$$\Delta t \leq \frac{\Delta x^2}{2}$$

Or in the extreme case we have:

$$\Delta t = \frac{\Delta x^2}{2}$$

In that case our previous the equation is tremendously simplified, and it is reduced to:

$$u(x_i, t_{j+1}) = \frac{1}{2}(u(x_{i+1}, t_j) + u(x_{i-1}, t_j)$$

Eigenvalues of a symmetrical matrix

If we have a matrix A that is symmetric, the eigenvalues can be obtain using a similarity transformation, that is:

$$A \to Z^{-1}AZ$$

where Z is the transformation matrix. The similarity transformation leave invariant the eigenvalues of the matrix. A strategy widely used to obtain the eigenvalues consists on applying a sequence of similarity transformations of the matrix A until we obtain a diagonal matrix.

$$\begin{aligned} A &\to Z_1^{-1}AZ_1 \\ &\to Z_2^{-1}Z_1^{-1}AZ_1Z_2 \\ &\dots \\ &\to Z_n^{-1}\dots Z_2^{-1}Z_1^{-1}AZ_1Z_2\dots Z_n \\ &\to diag(\lambda_1, \dots, \lambda_n) \end{aligned}$$

The elements of the diagonal matrix are the eigenvalues of the original matrix A.

The transformation matrices Z are chosen in a way that when we apply them, they eliminate the non diagonal elements. This elimination process is apply iteratively until we get a matrix that is almost diagonal. That means, until the non diagonal elements are nearly zero, with a certain tolerance.

The Jacobi method, which is the method that we are going to use to obtain the eigenvalues of the symmetrical matrix; consists on a sequence of similarity orthogonal transformation. Each transformation consists on a rotation that cancels one of the non diagonal elements of the matrix. This procedure is infallible for real symmetrical matrices, even though it converges slowly as the size of the matrix gets bigger and bigger. However, since this method is much simpler than other methods more efficient, it is still usually implemented for moderated size. The number of Jacobi rotations required to reach the convergence is usually between $3N^2$ and $5N^2$ rotations.

To implement the Jacobi method, we are going to use a famous subroutine in Fortran with the same name, that is going to help us with the heavy calculus.

Neural Network Architecture

The following architecture used was the adopted from [11].

We assume a feedforward neural network of the following structure:

- The input is scaled elementwise to lie in the interval $[-1, 1]$
- Followed by 8 fully connected layers each containing 20 neurons and each followed by a hyperbolic tangent activation function
- One fully connected output layer.

This setting results in a network with 3021 trainable parameters (first hidden layer: $2 \cdot 20 + 20 = 60$; seven intermediate layers: each $20 \cdot 20 + 20 = 420$; output layer: $20 \cdot 1 + 1 = 21$). We set the learning rate to the step function:

$$\delta(n) = 0.01\,\mathbf{1}_{\{n<1000\}} + 0.001\,\mathbf{1}_{\{1000\leq n<3000\}} + 0.0005\,\mathbf{1}_{\{3000\leq n\}}$$

which decays in a piecewise constant fashion and fed to the ADAM optimizer. 2500 Epochs were used.

Finding Eigenvalues and Eigenvectors using Deep Learning

Computing eigenvectors of a matrix is very important and an interesting problem in many fields.

Let A be a $n \times n$ real symmetric matrix. The dynamics of the proposed neural network model is described by

$$\frac{\mathrm{d}x(t)}{dt} = -x(t) + f(x(t))$$

for $t \geq 0$, where

$$f(x) = [x^{\mathrm{T}}xA + (1 - x^{\mathrm{T}}Ax)I]x$$

and $x = (x_1, ..., x_n)^{\mathrm{T}} \in R^n$ represents the state of the network, I is the identity matrix with square dimensions n.

This was proposed in 2004[10]. Now we are able to effectively solve differential equations using neural networks, meaning this problem is reduced to a constraint problem where the loss function is simply to minimize the function above.

We then approach a convergence point [10], where the eigenvector v belonging to A. We can then compute the corresponding eigenvalues as such $\lambda = v^{\mathrm{T}}Av/v^{\mathrm{T}}v$ In our implementation we find the largest eigenvalue and its corresponding eigenvector entries. This method can easily be generalized to find all eigenvalues and eigenvectors.

The architecture of the neural net was a simple 1 layer with 50 hidden neurons and 10000 epochs.

Benefits and Shortcomings

Benefits

1. PINNs can solve eigenvalues and eigenvectors accurately and efficiently, even for large and complex systems.
2. PINNs can handle noisy or incomplete data and can learn from multiple sources of data.
3. PINNs are a flexible and versatile tool, capable of handling a wide range of problems, including nonlinear and time-dependent systems.

Shortcomings

1. PINNs require a large amount of data and computational resources, which may not be practical for some problems.
2. PINNs can be sensitive to hyperparameters, such as the network architecture and learning rate, which may require careful tuning.
3. PINNs can be difficult to interpret, as the learned model is not always transparent or easy to understand.

Setting Up the Bias-Variance Tradeoff Analysis

The bias-variance tradeoff analysis was done on three different sets of algorithms, including linear regression, deep learning and ensemble methods. To improve the accuracy of the results and reduce the bias, resampling was used. This involves iteratively resampling the data throughout each cycle, using the bootstrap resampling method. To ensure consistency, various functions from SciKit-Learn were used for resampling, the various regressors as well as the splitting of data into test and training data. The mlxtend library was used for the bias-variance decomposition. One issue that arose was the decision of what was chosen as complexity for each of the sets of algorithms, as they do not share all of the same features. The complexity for the linear regression and ensemble methods was chosen to simply be the size of the data, the higher the complexity, the larger the dataset. For neural networks, the complexity was indicated by the size of the hidden layer, higher complexities allowed for more hidden nodes. It is also worth noting that the number of bootstrap cycles was kept consistent for linear regression and decision tress, but must be much lower for random forests and neural networks due to the significantly longer computation times. The data used for this analysis was the Boston housing data.

Results

Differential Equations

Now we are going to plot our results for the heat equation using the method of finite differences. We are going to make two different plots with two different values for the space between two nodes that are. $\Delta x = \frac{1}{10}$ (figure 1) and $\Delta x = \frac{1}{100}$ (figure 2). For the increment in time we have used the stability criterion which states that:

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$$

In both cases for the increment of time Δt we are going to use the following expression:

$$\Delta t = \frac{\Delta x^2}{2}$$

In the two different plots we are going to display the initial distribution of temperature which is: $sin(\pi x)$ and the solutions in two points in time t_1 and t_2 where $u(x, t_1)$ is smooth but still significantly curved and $u(x, t_2)$ is almost linear, close to the stationary state.

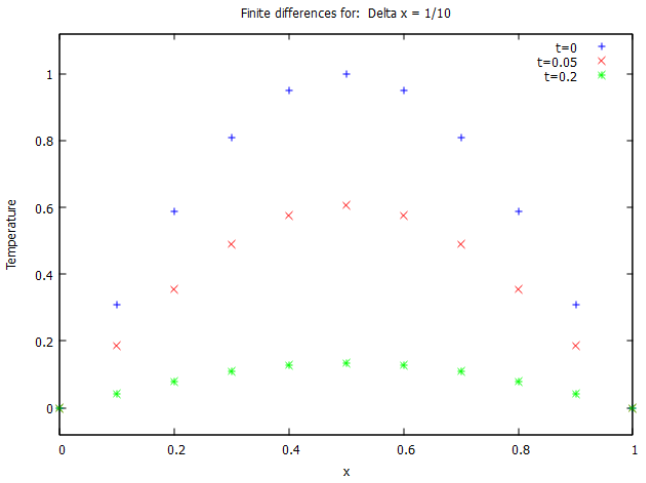

Figure 1: *Finite differences method for the 1D heat equation with* $\Delta x = \frac{1}{10}$*. Tolerance as a function of* x *and* $t = 0$*,* $t = 0.05$ *and* $t = 0.2$

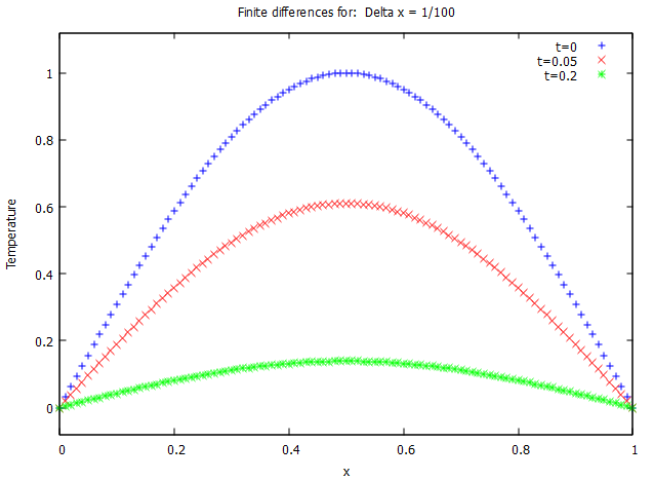

Figure 2: *Finite differences method for the 1D heat equation with* $\Delta x = \frac{1}{100}$*. Tolerance as a function of* x *and* $t = 0$*,* $t = 0.05$ *and* $t = 0.2$

The NN approach to differential equations proved to work quite well. Figure 3 shows the analytical solution, while the simulation using the neural network is shown in figure 4. The graphs look nearly identical, though the discrepancies are apparent as can be seen from the MSE values shown in table 1, where the error is the deviation from the analytical solution.

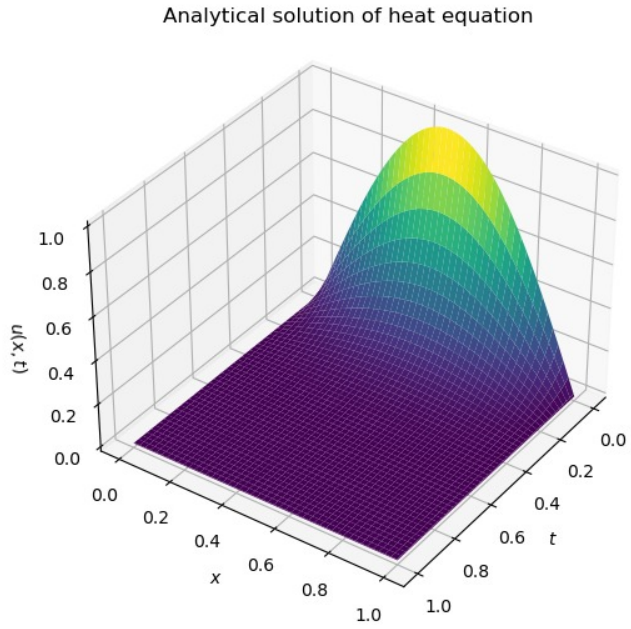

Figure 3: *Analytical solution to heat equation. Temperature is displayed on the vertical axis as* $u(x, t)$*,* t *is time and* x *is the spatial coordinate.*

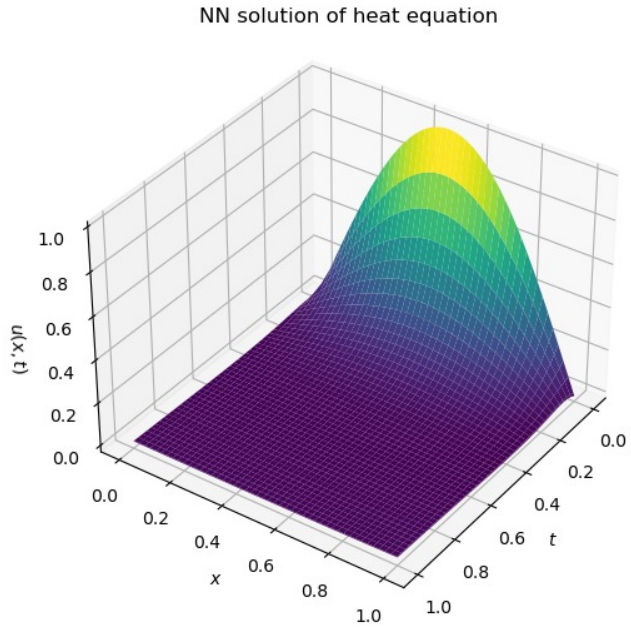

Figure 4: *Neural network solution to heat equation. Temperature is displayed on the vertical axis as* $u(x, t)$*,* t *is time and* x *is the spatial coordinate.*

Table 1: *Minumum MSE values for Neural Network and Finite Differences.*

Method	MSE
Neural Network	0.0533
Finite Differences	10^{-9}

Eigenvectors and Eigenvalues

To determine the eigenvalues and eigenvectors of a 6×6 symmetrical matrix, we are going to use the following matrix as an example:

$$\begin{pmatrix} 0.192 & 0.449 & 0.561 & 0.779 & 0.857 & 0.571 \\ 0.449 & 0.802 & 0.835 & 0.879 & 0.505 & 0.469 \\ 0.561 & 0.835 & 0.370 & 0.463 & 0.450 & 0.408 \\ 0.779 & 0.879 & 0.463 & 0.615 & 0.432 & 0.256 \\ 0.857 & 0.505 & 0.450 & 0.432 & 0.317 & 0.636 \\ 0.571 & 0.469 & 0.408 & 0.256 & 0.636 & 0.705 \end{pmatrix}$$

This matrix had is truncated to 3 significant figures. For the exact matrix please refer to the code provided.

Using our explained Jacobi method we get the following eigenvalues:

$$\begin{pmatrix} -0.33559901 \\ 3.3783123 \\ -0.75755364 \\ -0.09221539 \\ 0.18553632 \\ 0.62197524 \end{pmatrix}$$

As an extra we can also get the eigenvectors associated with its corresponding eigenvalue. For clarity purpose, we are only going to display the eigenvector associated with the largest eigenvalue (which is 3.3783123):

$$\begin{pmatrix} 0.40661031 \\ 0.48043618 \\ 0.38109347 \\ 0.42498845 \\ 0.38403463 \\ 0.36127427 \end{pmatrix}$$

To determine the eigenvalues and eigenvectors of a 6×6 symmetrical matrix, we are going to use the following matrix as an example

Using the neural network to compute the eigenvector corresponding to the largest eigenvalue. We got this eigenvector:

$$\begin{pmatrix} 0.40661035 \\ 0.48043619 \\ 0.38109342 \\ 0.42498848 \\ 0.38403463 \\ 0.36127426 \end{pmatrix}$$

with the corresponding eigenvalue $\lambda = 3.37831241$

Bias-Variance Tradeoff

The values for MSE will be lowest where the variance and bias are the lowest due to the way in which the mlxtend decomposition works. The MSE will be the main predictor of how well a model performs, and will be compared to the relevant bias and variance at that complexity. For all of the simulations, the number of rounds was 200 rounds for the bias-variance decomposition as well as MSE as the cost function. The simulations for the linear regression were done using 50 bootstrap cycles. Figure 5 shows the MSE as a result of complexity for standard OLS.

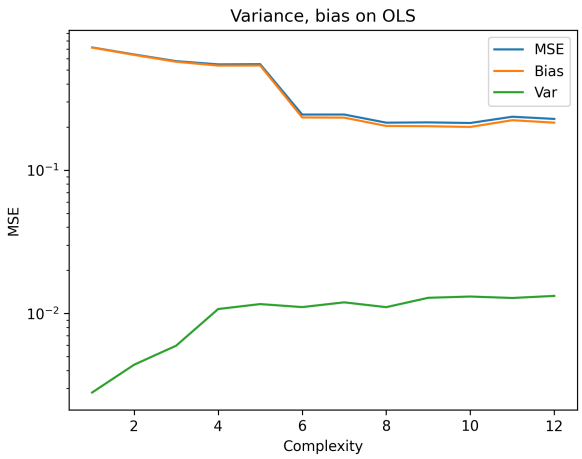

Figure 5: *Variance and bias on OLS as a function of complexity and MSE. 50 Bootstrap cycles.*

There is a gradual decrease in MSE as the complexity increases, up until a complexity of about 10, after which the MSE slowly increases again. The variance increases as the complexity increases. The minimum value for MSE is 0.2135 at complexity 10, which corresponds to a bias of 0.2004, which is also a minimum for the dataset. The variance is on the higher end of about 0.0131.

The ridge parameter λ was chosen to be 0.01, which proved to provide good scores, while the lasso parameter α was chosen to be 0.01 for the same reason. Both Ridge and Lasso regularization show similar trends to standard OLS with general decreases in MSE and bias corresponding with an increase in variance. The minimum values for the MSE are 0.2135 at complexity 10 and 0.2154 at complexity 10 for Ridge and Lasso respectively. The minimum values for bias are 0.2004 at complexity 10 for ridge and 0.2026 at complexity 10 for Lasso. The simulations for the bias-variance tradeoff for ridge and lasso are shown in figures 6 and 7.

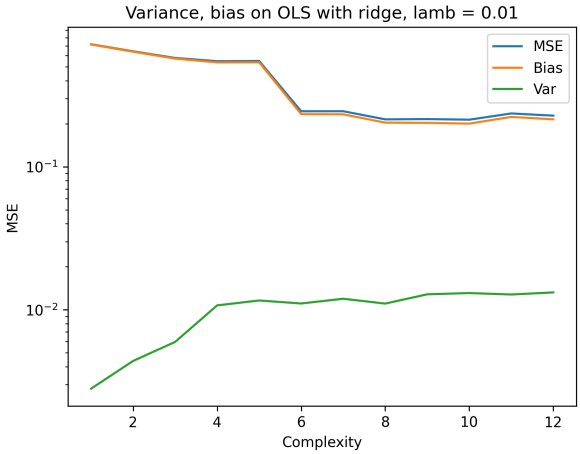

Figure 6: *Variance and bias on OLS with ridge as a function of complexity and MSE. 50 Bootstrap cycles.*

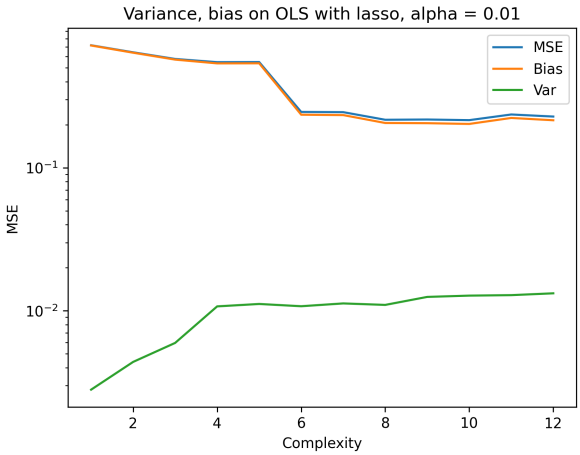

Figure 7: *Variance and bias on OLS with lasso as a function of complexity and MSE. 50 Bootstrap cycles.*

For the decision tree simulations, 50 bootstrap cycles were used as well as 200 rounds for the bias-variance decomposition. The expected trend of a decreasing MSE and bias corresponding to an increase in variance until a complexity of 4, after which the MSE and variance continue to decrease while the bias shows a general decrease. Curiously however, the bias decreases at the same time as the variance for complexities 4 through 7, then increases slightly up to a complexity of 10. The minimum value for MSE is 0.3508 at complexity 12, which corresponds to a bias of 0.1765, which is the lowest in the dataset. The variance at complexity 12 is 0.1743. Figure 8 shows this.

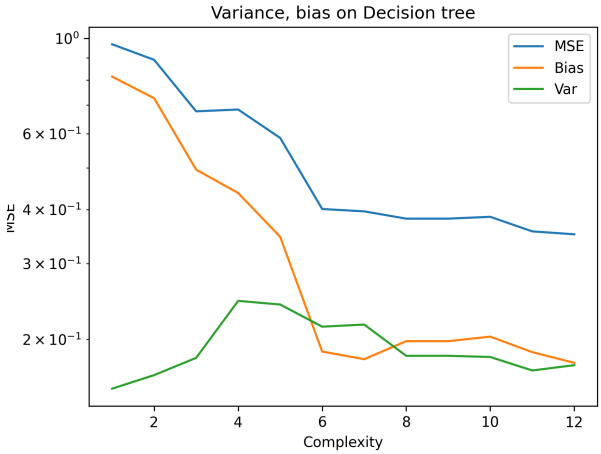

Figure 8: *Variance and bias on Decision trees as a function of complexity and MSE. 50 Bootstrap cycles.*

Only 5 bootstrap cycles were used on random forests due to the long computation times. Random forests performed consistently well beyond a complexity of 5 with an MSE hovering about 0.15. The variance and bias followed the same trends throughout each complexity with slight deviations. The minimums were 0.1217 at a complexity of 10 and 0.0265 at a complexity of 6 for bias and variance respectively, while the minimum for MSE was 0.1489 at complexity 10. This is shown in figure 9.

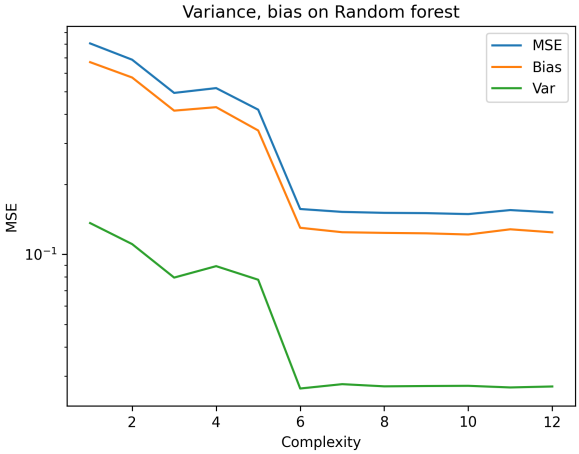

Figure 9: *Variance and bias on Random Forests as a function of complexity and MSE. 5 Bootstrap cycles.*

Similarly to random forests, the bootstrap cycles for the neural network also had to be limited to 5 due to computational restraints. The bias decreases as the variance increases until a complexity of 4, after which both the variance and bias continue to increase until a complexity of 5. Thereafter, the

variance and bias both steadily decrease. The MSE reaches a minimum value of 0.4338 at a complexity of 11 which corresponds to a bias of 0.4160 which is also the minimum, and a variance of 0.0178. The variance is rather high compared to the surrounding results. This is shown in figure 10.

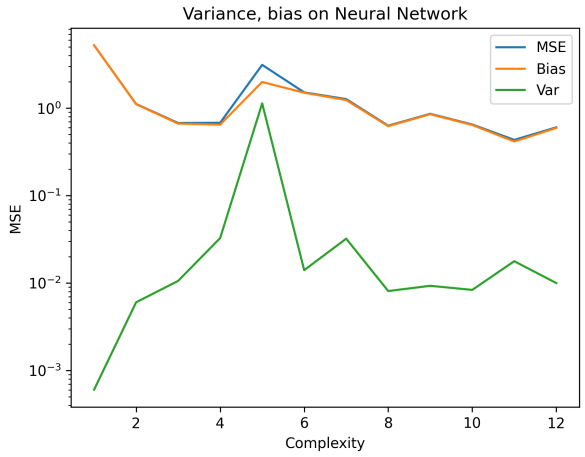

Figure 10: *Variance and bias on Neural networks as a function of complexity and MSE. 5 Bootstrap cycles.*

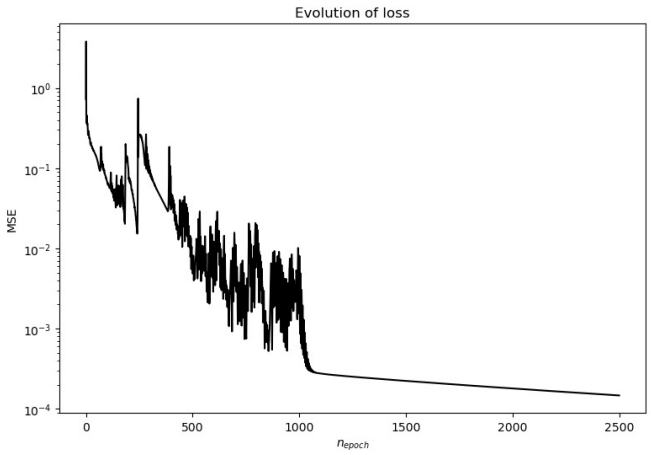

Figure 11: *Evolution of loss as a function of number of epochs.*

Discussion

Differential Equations

The results we have show that the finite difference method worked exceptionally well with a very low MSE. The Neural Net did not perform as well as the finite difference method. A problem like this one is very well suited for a finite difference method where it is simple enough to be stable, amongst other factors. The results demonstrate to us that when given a problem related to differential equations, one must remember that there are different methods that are suited for different reasons.

This analysis could have been done better by extending the time domain and comparing how the two methods compare. Another useful analysis that we suggest future reaserch in: Using very complicated geometries and seeing how Neural Networks perform. This analysis is already done [13] [14] [15] [16] [**?**] which means comparisons can be done to established theory.

Eigenvectors and Eigenvalues

For finding the eigenvalues of a symmetrical matrix both used methods worked really good. The differences between the results obtain analytically, in the Jacobi method and in the NN is in the order of magnitude 10^{-7}, which is almost the inherent numerical error in which the computers work. However, in the Jacobi method, the number of Jacobi rotations required to reach the convergence is usually between $3N^2$ and $5N^2$ rotations. Which, as we can appreciate, escalates very quickly and for higher dimensional situations, the problem can turn unapproachable. That is where the Neural Networks would become more useful rather than the traditional algebra methods.

Bias-Variance Tradeoff

The analysis for the bias-variance tradeoff shows results that are somewhat unexpected for the most part. The OLS trends for bias and variance follow the expected path of decreasing bias as the variance increases for higher complexities, though the poor performance of the lasso regularization and the middling increase in performance of the ridge regularization were surprising. This was likely due to a lack of fine tuning of the parameters, especially the lasso parameter. The decision tree simulation showed results that were similarly poor compared to the lasso regularized OLS. This could again be improved by fine tuning or pruning of the decision tree. Of note however, is the trend of bias and variance both decreasing between complexities 4 and 7 and staying similarly low. Rather than a good model fit, this more likely suggests mediocre values for both bias and variance.

The random forests regressor displayed the best performance out of the methods tested. It reached the lowest minimum of 0.1489 at a complexity of 10 and the trends for the bias and variance both decreased as the complexity increased. The model

achieved both a low bias and low variance, meaning that the model was not very overfit or underfit but instead well balanced. The neural network performance was also poor, though the trends are promising. This again is likely due to poor implementation and low sizes of the hidden layer. An improvement here would have been to change the definition of complexity of the neural network to number of hidden layers, though there were issues implementing this, and it was difficult under the time constraints. Overall, the poor results could be attributed to poor implementation of the models, but most of the trends of the models are promising. It may also be difficult to compare different methods due to the rather vague definition of complexity that changes depending on the method.

Conclusion

It's difficult to say whether finite difference or physics-informed neural networks (PINNs) is generally "better" for solving differential equations, as the suitability of each method depends on the specific problem at hand. Both methods have their own strengths and weaknesses, and the best approach may depend on factors such as the complexity of the differential equation, the available data, and the desired accuracy of the solution.

Finite difference is a well-established numerical method for solving differential equations. It involves approximating the derivative of a function using a finite difference formula, and then using this information to advance the solution from one time step to the next. Finite difference is a relatively simple and robust method, but the accuracy of the solution may depend on the choice of discretization parameters. Also it is worth noting that finite difference is very costly to implement on complex geometries such as in fluid mechanics.

On the other hand, PINNs are a relatively recent development in the field of machine learning, and they have been used to successfully solve a wide range of differential equations. PINNs involve training a neural network to approximate the solution of a differential equation by minimizing a loss function that measures the discrepancy between the neural network's predictions and the known physics of the problem. PINNs can be very accurate, but they may require more data and computational resources than finite difference, and they may be more sensitive to the choice of hyperparameters. However, since PINNs do not solves the problem directly and solves it in a unknown method that is more direct, it becomes much more efficient when dealing with multi dimensional problems that might be unapproachable using more traditional methods in a reasonably amount of time.

When it comes to bias-variance, the various methods analyzed for bias-variance tradeoff show that an increase in bias typically results in a decrease in variance. There are certain complexities for some methods, primarily random forests that can have both low bias and variance.

References

[1] T. Hastie, R. Tibshirani, and J. Friedman., *The Elements of Statistical Learning.*, Springer, New York, 2009.

[2] Aurelien Geron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow,*, O'Reilly, 2019.

[3] Kevin Murphy, *Machine Learning: A Probabilistic Perspective*, MIT Press, 2022.

[4] I.E. Lagaris, et al., *Artificial neural networks for solving ordinary and partial differential equations*, IEEE, 1998.

[5] Alex Honchar, *Neural networks for solving differential equations*, Becoming Human: Artificial Intelligence Magazine, 2017.

[6] M.M Chiaramonte and M. Kiener, *Solving differential equations using neural networks*, Stanford, 2013.

[7] A. Tveito and R. Winther, *Introduction to Partial Differential Equations*, Springer, 1998.

[8] M. Kumar and N. Yadav, *An Introduction to Neural Network Methods for Differential Equations*, Springer, 2015.

[9] D. Rajnarayan and D. Wolpert, *Bias-Variance Trade-offs: Novel Applications*, Springer, 2011.

[10] Z. Yi and Y. Fu, *Neural Networks Based Approach for Computing Eigenvectors and Eigenvalues of Symmetric Matrix*, Elsevier, 2004.

[11] M. Raissi, et al., *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*, Cornell University, 2017.

[12] M. Raissi, et al., *Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*, Cornell University, 2017.

[13] D. Li, et al., *A Physics-Informed Neural Network Approach for Solving Burgers' Equation with Complicated Geometry and Boundary Conditions*, IEEE, 2021.

[14] C. Cao, Y. Want, and Y. Lu, *Solving the Heat Equation with Complicated Boundary Conditions Using a Physics-Informed Neural Network*, Journal of Computational Physics, 2020.

[15] J. Wang, Y. Zhang, and L. Li, *Physics-Informed Neural Networks for Solving Partial Differential Equations with Complex Geometry Boundary Conditions*, IEEE, 2020.

[16] Y. Zhang, J. Wang, and L. Li, *Solving the Navier-Stokes Equations with Complex Boundary Conditions Using a Physics-Informed Neural Network*, Journal of Computational Physics, 2019.