Solving a diffusion equation using neural networks

Sakarias Frette, William Hirst, Mikkel Metzsch Jensen
(Dated: December 16, 2021)

In this rapport we implement a neural network using Tensorflow framework in order to solve a 1D diffusion equation and a 6×6-matrix eigenvalue problem. When solving the diffusion equation we compare the performance, both accuracy and computation time, to an explicit solver using forward Euler for time discretization and central difference for spatial discretization. We generally found that the explicit solver drastically outperformed the neural network when considering the error vs computation time ratio. However, the neural network was able to produce very accurate results down to the order of 10^{-10} for the tested settings. Thus it serves as an interesting topic as the neural network might be more valuable for more complex partial differential equations. When using our network to solve the eigenvalue problem we successfully found an eigenvalue to an accuracy of order 10^{-5} and the corresponding eigenvector with mean squared error of order 10^{-13}, when comparing with alternative solvers.

I. INTRODUCTION

Differential equations are a cornerstone in understanding the physical processes around us. From the Schrödinger equation, which explains low energy quantum mechanical systems, to the Navier-Stokes equations that governs flow of fluids, they explain how the system evolves with respect to time, space or both. Traditional numerical solutions such as finite difference[3] have yielded good results, and they are computationally fast. However, they base themselves on discretization, and so the resulting solution is also discretized. Thus, one does not get a function which can be evaluated at any given point. With the Neural ordinary differential equation paper[1] from 2018 and other papers, it was shown that neural networks also can solve both ODE's and PDE's. With the neural network one can get an actual function that can be evaluated continuously throughout the training domain and in principle also outside the domain. There is however a potential cost of computation time.

In 2003 it was shown by Yi et al. [4] that neural networks can be used to solve eigenvalue problems. Eigenvalue problems occur in many physical phenomenon, such as the Schrödinger equation, where the energy of the system is the eigenvalue of the Hamilton operator, or in vibration analysis, where the eigenvalue corresponds to the eigen-frequency.

In this paper we will use both finite difference and neural network to solve the 1D diffusion equation. We investigate the solution for different discretization and number of epochs, and compare the total error and at specific time points. By also considering the computation time we study and compare the efficiency of the two methods. In addition we use the neural network to compute the eigenvalues and eigenvectors for a 6×6-matrix.

The report is structured in the sections theory, implementation, results and discussion, and conclusion. The theory includes the mathematics behind the diffusion equation, the finite difference method, neural networks for solving PDE's and eigenvalue problems using neural networks. The implementation includes the pseudo code algorithms for the finite difference method, the neural network PDE solver, as well as some of the choices used to solve the eigenvalue problem. The results and discussion includes results from the finite difference method, the neural network for the PDE and the eigenvalue problem, as well as discussion comparing the methods. Finally we summarize our findings in the conclusion.

II. THEORY

A. The diffusion equation

The generalized diffusion equation is given as

$$\frac{\partial}{\partial t}u(\mathbf{x},t) = \nabla \cdot [\alpha(u,\mathbf{x},t)(\nabla u(\mathbf{x},t))],$$

where u is the solution, $\mathbf{x}$ is a 3D vector containing spacial coordinates, t is temporal coordinate, and α is a function comprised of the temporal and spacial derivative constants and possibly the solution u. For this project we will look at a simpler example, namely the 1D diffusion equation with $\alpha = 1$. The 1D diffusion equation is written as

$$\frac{\partial}{\partial t}u(x,t) = \frac{\partial^2}{\partial x^2}u(x,t). \tag{1}$$

In our case u is defined in the domain of

$$x \in \Omega : [0,1],$$

for $t > 0$, with the initial condition

$$u(x,0) = \sin(\pi x), \quad 0 < x < L,$$

and the zero Dirichlet boundary conditions

$$u(0,t) = u(L,t) = 0, \quad t \geq 0.$$

1. Analytical solution

The 1D diffusion eq. (1) can be solved analytically, which can serve as a test for numerical solutions. By considering the specifics of the diffusion equation one can derive a solution on the following form that will satisfy eq. (1).

$$u_e(x,t) = A\sin(kx)e^{-\omega t}, \tag{2}$$

where we have introduced three degrees of freedom: A, k and ω. A is the amplitude of the sinusoidal function, k is the angular frequency of the spacial domain and ω is the rate of diffusion. By inserting the proposed solution eq. (2) into the diffusion eq. (1) we find the derivatives

$$\frac{\partial^2 u_e(x,t)}{\partial x^2} = -k^2 A\sin kx e^{-\omega t},$$
$$\frac{\partial u_e(x,t)}{\partial t} = -\omega A\sin(kx)e^{-\omega t},$$

which leads to the condition

$$\omega = k^2. \tag{3}$$

By expression ω in terms of k according to eq. (3) and implementing the zero Dirichlet boundary condition we can determine k as

$$u_e(0,t) = u_e(L,t),$$
$$0 = A\sin(kL)e^{-k^2 t}.$$

Thus we must have $kL = n\pi$, for any integer $n = \{0, \pm 1, \pm 2, \ldots\}$ such that

$$k = \frac{n\pi}{L}.$$

Finally we can determine A by the use of the initial condition. With the use of the previous findings of ω and k we get

$$u_e(x,0) = \sin(\pi x),$$
$$A\sin\left(\frac{n\pi}{L}x\right) = \sin(\pi x).$$

By choosing $n = 1$ and $L = 1$ we find $A = 1$. Our final analytical solution is then

$$u_e(x,t) = \sin(\pi x)e^{-\pi^2 t}. \tag{4}$$

2. Finite difference discretization

One common approach to solving the diffusion equation numerically is by the use of finite difference method. By discretizing the space and time domain respectively as

$$x = i\Delta x, \quad i \in \{0,1,2,\ldots,N_x\}, \quad N_x = L/\Delta x + 1,$$
$$t = n\Delta t, \quad n \in \{0,1,2,\ldots,N_t\}, \quad N_t = T/\Delta t + 1,$$

for a total time T, we can approximate the derivatives as finite differences. By convention we will use subscript indexing for spacial coordinates and superscript indexing for temporal coordinates such that the discretized solution at a given point is written $u(x_i,t_n) = u_i^n$. For the first order temporal derivative we will use the so-called explicit forward Euler scheme, while for the second order spacial derivative we use the second order central difference, that is

$$\frac{\partial u(x,t)}{\partial t} = \frac{u_i^{n+1} - u_i^n}{\Delta t} + \mathcal{O}(\Delta t),$$
$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \mathcal{O}(\Delta x^2).$$

Inserting the approximations into the diffusion eq. (1) we get the discretized equation

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2},$$

which can be rewritten on the form

$$u_i^{n+1} = u_i^n(1-2C) + C(u_{i+1}^n + u_{i-1}^n), \tag{5}$$

for $C = \Delta t/\Delta x^2$. Eq. (5) has a stability criteria, formalised by von Neumann[2], given as

$$\alpha C \leq \frac{1}{2}, \tag{6}$$

where α is the parameter multiplied with the spacial part of the diffusion equation. In our case $\alpha = 1$, which allows for reasonably choice of step sizes, while for large α this method requires an extensive fine discretization of the time domain.

B. Neural networks and PDE's

Another, but yet very different, approach to the numerical solution is by the use of a neural network.

1. Trial function

We begin by constructing a trial function $g_t(x,t)$ that will serve as the final prediction from our model. That is, for a well trained model the trial function will resemble the exact solution. We define the generic trial function as

$$g_t(x,t) = h_1(x,t) + h_2(x,t)N(x,t,P), \tag{7}$$

where $N(x,t,P)$ is the output from the neural network as a function of space coordinate x, time coordinate t and weights and bias P, while $h_1(x,t)$ and $h_2(x,t)$ are functions that ensures that initial conditions and boundary conditions are satisfied. Since we have zero Dirichlet

conditions on the boundaries we might choose a function like $x(1-x)$ which will ensure that we get zero on the boundaries. In addition we want to satisfy the initial condition such that $h_1(x,0) = u(x,0)$ and $h_2(x,0) = 0$. One way of achieving this is by the defining the functions as

$$h_1 = (1-t)u(x,0) = (1-t)\sin(\pi x),$$
$$h_2 = x(1-x)t.$$

A side effect of the above definitions is a linear decrease of h_1 and a linear increase of h_2. However this is not an issue since $N(x,t,P)$ should simply adjust accordingly. By combining the expressions of h_1 and h_2 the trial function reads

$$g_t(x,t) = (1-t)\sin(\pi x) + x(1-x)t\,N(x,t,P).$$

2. *Cost function*

When using the neural network for supervised learning we essentially wants an output $N(x,t,P)$ for which the trial functions satisfy the PDE as well as possible. Thus we must define a cost function that assures this. Since it is unusual to know the exact solution we will instead use the residual. The residual is defined as putting all the terms on one side of the equation, and in the case of the diffusion equation, we have the residual

$$R = \frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{8}$$

which should be equal to zero in the whole domain for a perfect solution of the PDE. Thus we choose the cost function to be the mean squared residual

$$C(x,t) = \langle R^2 \rangle = \left\langle \left(\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2} \right)^2 \right\rangle. \tag{9}$$

This definition of the cost function leads to typical minimization problem which can be used to optimize the weights and biases of the network. The practical aspect of evaluating the cost function is further explained in the implementation section III.

C. Finding eigenvalues through a neural network

The work of Yi et al. [4] proposes a method to compute the eigenvectors of a symmetric matrix A using a neural network. The idea is to define the eigenvectors of A as solutions of a simple ODE. In this section we will briefly describe the essentials needed to implement such a method.

For a given matrix A, an eigenvector v with eigenvalue λ must satisfy the eigenvalue equation

$$Av = \lambda v. \tag{10}$$

For a vector x it follows from eq. (10) that

$$x^T x A x - x^T A x x = 0, \tag{11}$$

if and only if $x = v$ or $x = \mathbf{0}$ (the null vector). Finding the eigenvectors of A is therefore a question of solving eq. (11). By choosing eq. (11) as the cost function, we can solve the eigenvalue problem with respect to x and subsequently solving for the eigenvalue using eq. (10) and eq. (11) as

$$\begin{aligned} x^T x\lambda x - x^T A x x &= 0 \\ x^T x\lambda - x^T A x &= 0 \\ x^T x\lambda &= x^T A x \\ \iff \qquad \lambda &= \frac{x^T A x}{x^T x}, \end{aligned} \tag{12}$$

However, in order to avoid that the solution gravitates towards the trivial solution $x = \mathbf{0}$, we instead use the following ODE (suggested by Yi et al. [4]).

$$\frac{dx(t)}{dt} = f(x) - x(t), \tag{13}$$

where

$$f(x) = [x^T x A + (1 - x^T A x)I]x, \tag{14}$$

A is a symmetric matrix, I is the identity matrix and t is a variable.

We can study the derivative of of the length

$$\begin{aligned} \frac{d(x^T x)}{dt} &= 2x(t)\frac{dx(t)}{dt} \\ &= 2x(f(x) - x) \\ &= 2x(x^T x A x - x^T A x x) = 0, \end{aligned} \tag{15}$$

where we used eq. (11) and eq. (13) for the last transition. From eq. (15) we see that the derivative of the length must be constant, and by substituting $x(t)^T x(t) = x(0)^T x(0)$ into the ODE eq. (13) we arrive at

$$\frac{dx(t)}{dt} = x(0)^T x(0) A x - x^T A x^T x, \tag{16}$$

which is the final formulation of the ODE that we are going to use for the solving of the eigenvalue problem. At this point we state the following theorem.

Theorem 1. *Given an initial state, x(0) every solution of eq. (16) will converge towards an eigenvector of A corresponding to the largest eigenvalue, v_{max} when $t \to \infty$ given that the following is true.*

- *x(0) is non-zero,*
- *x(0) is not orthogonal to the v_{max}.*

For the proof of this theorem we refer again to the text of Yi [4], specifically theorems 3 and 4 on pages 1159 and 1160.

III. IMPLEMENTATION

The code can be found on Github.

A. Solving a PDE using finite difference

Before we solve our PDE with the neural network, we first want to solve it using an explicit solver. The explicit solver sets up a recursive algorithm to step forward in time using eq. (5). First, we initialize all arrays and constants, and insert the initial condition from section II A. For each time step we calculate the solution for each internal point, excluding the external points as the boundary points are fixed. After the boundary points are inserted, we update the arrays u and u_1, and store u in an array $u_{complete}$. The algorithm is outlined in algorithm 1, while the class can be found in ExplicitSolver.py (see github) .

Algorithm 1: Finite difference solution

```
Initialize arrays for solution and domain;
Insert initial condition ;
for 1 → N_t do
    Calculate the inner spacial points for u;
    Insert boundary conditions;
    Update u, u_1;
    Insert solution to u_complete;
end
Return u_complete
```

Note here the use of three arrays, namely the final solution $u_{complete}$, u and u_1. $u_{complete}$ stores the solution for every combinations of x and t. u_1 stores the values of the previously loop and is set equal to the initial condition, eq. (II A). u stores the values calculated in the current iteration.

B. Solving a PDE with an neural network

For the setup of the neural network we decide to use a TensorFlow framework. The solver is implemented as a class in NN_PDE_solver.py (see github). The main structure of the code is divided into two parts; creating the neural network model and training of the model.

1. Model

The model is built according to a fixed network architecture as shown in table I.

Table I: Network architecture, for each hidden layer showing the number of number of hidden nodes, the number of trainable parameters given by the weights and biases, and the activation function.

Layer	Nodes	Params	Activation
1	20	60	Sigmoid
2	20	420	Sigmoid
3	20	420	Sigmoid
Output	1	21	None

This specific framework was decided upon after an extensive testing of the network performance on the PDE. The data, which is fed into the first hidden layer of the model, has two features, namely position x and time t. Since we want the network to predict the solution in the whole spacial and temporal domain we need to train on every combination of the discretized spacial and temporal points. To achieve this, we create a design matrix X given as

$$X = \begin{bmatrix} t_0 & x_0 \\ t_0 & x_1 \\ \vdots & \vdots \\ t_1 & x_0 \\ t_1 & x_1 \\ \vdots & \vdots \\ t_{N_t} & x_{N_x} \end{bmatrix} \Bigg\} (N_x \cdot N_t). \tag{17}$$

2. Training

The training of the network is executed as a custom training, since TensorFlow's own training functionality can not easily adapt to the cost function of this problem. Hence, for each epoch we calculate the loss and gradient of the loss with respect to model parameters using TensorFlow GradientTape. The parameters are then updated using the ADAM optimizer, which uses an adaptive momentum. The algorithm is outlined in algorithm 2.

Algorithm 2: Training of the network.

```
Initialize model ;
for epoch in epochs do
    Calculate the loss and the gradient of the loss
      with respect to model parameters ;
    Update model parameters using ADAM
      optimizer ;
    Store loss;
end
```

C. Eigenvalue and eigenvectors

In order to calculate eigenvalues and eigenvectors we first need to specify the matrix A of interest. In this report we will limit ourselves to a symmetric 6×6-matrix. We generate intially a 6×6 matrix, Q by drawing random points from a standard normal distribution. We insure that the final matrix A is symmetric by defining it as

$$A = \frac{Q^T + Q}{2}. \tag{18}$$

From this point on we can define the ODE according to eq. (13) and solve for the eigenvectors using a similar approach to what we have done for the diffusion equation in section III B.

The main distinction from section III B is the sizes of the input and output layer, and the cost function. For the PDE case the input was two-dimensional (for x and t) and the output was a single number. In our ODE case we only have one variable, t and our output is a 6-dimensional vector. Therefore the input layer had one node and the output layer has 6 nodes.

As for the cost function we will again use the residual of our equation (eq. (16)) given as

$$C(x) = \left\langle \left(x(0)^T x(0) Ax - x^T A x^T x - \frac{dx(t)}{dt} \right)^2 \right\rangle.$$

Finally we must satisfy the criteria in theorem 1. Firstly, as we have no pre-knowledge on v_{max}, there is no way of guarantying that our choice of $x(0)$ will not be orthogonal to v_{max}. Therefore we instead use randomly generated values for $x(0)$, again sampling from a "standard normal" distribution, in order to get some values that fit the criteria. Secondly, we need the final solution for $t \to \infty$. We approximate this by taking the final solution for a large enough t-domain given as $t \in [0, 10^4]$.

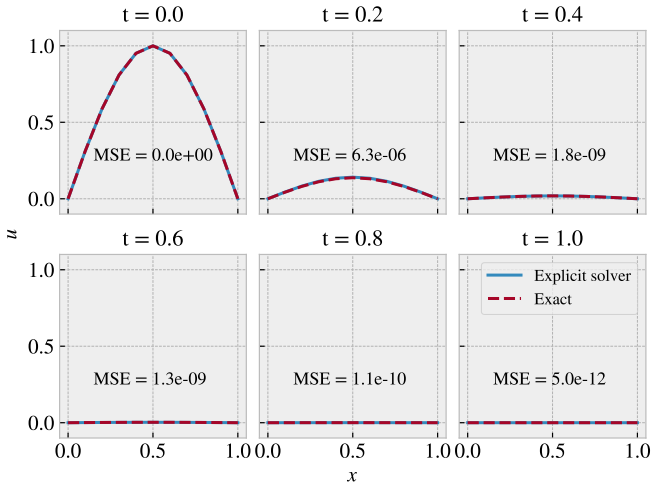
IV. RESULTS AND DISCUSSION

A. Diffusion Equation

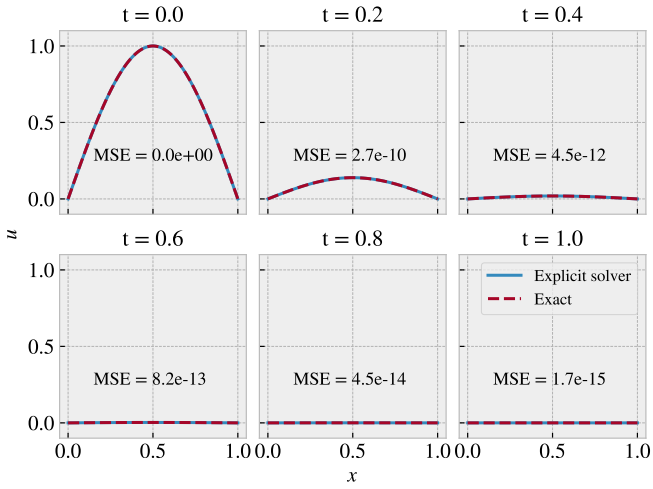
In this section we address the numerical solutions of the diffusion equation eq. (1) discussed in section II A using finite difference and the neural network respectively[1].

1. *Finite Difference*

We are going to investigate the explicit solution for two spatial step sizes: $\Delta x = 0.1$ and $\Delta x = 0.001$. For the time step we choose $\Delta t = \Delta x^2/2$ according to the von Neuman condition stability criteria in eq. (6). The resulting numerical solution is shown against the exact solution for six selected time points in figure 1.



(a) $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$.



(b) $\Delta x = 0.01$, $\Delta t = 5 \times 10^{-5}$.

Figure 1: Explicit solution vs. exact solution for the diffusion equation eq. (1) with $L = 1$ and $T = 1$ for different combinations of Δx and Δt. MSE denotes the mean squared error between the numerical and analytical solution.

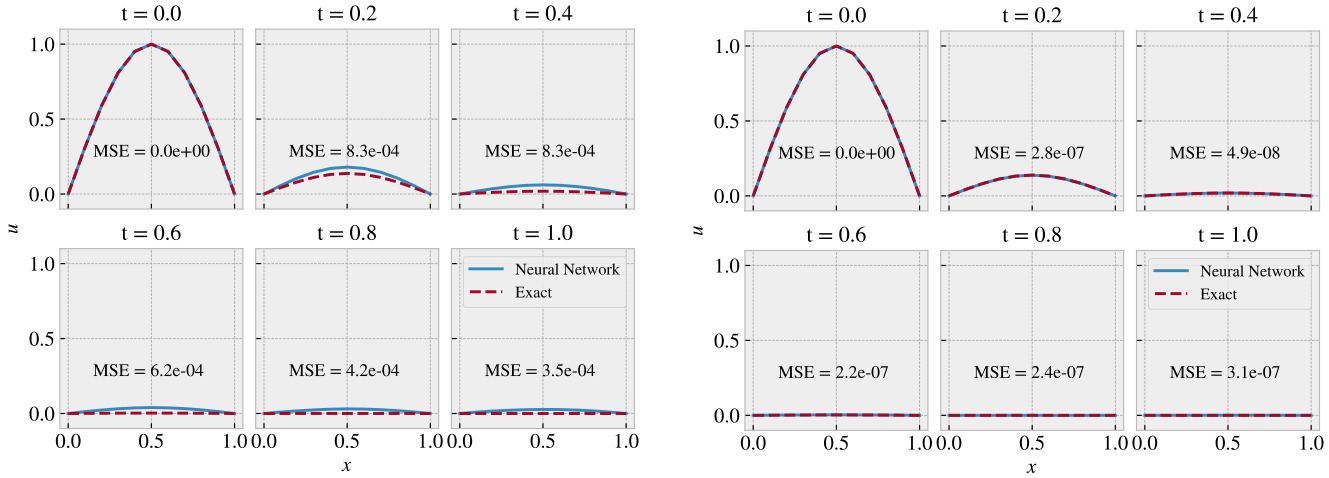
From the results in figure 1 we observe immediately a relatively strong fit between the numerical and analytical solution. As expected, we see that the numerical solution is improved when decreasing Δx from $\Delta x = 0.1$ (1a) to $\Delta x = 0.01$ (1b). When inspecting each time frame we first observe in both cases that the solution is exact for $t = 0$ which is also expected since the numerical solution takes a starting point from the initial condition.

[1] A set of gifs for some of the solutions can be found here.

Secondly, we observe that, among the six time points, the error peaks at $t = 0.2$. We know that the analytical solution decays from a sine curve into a horizontal line, for which the change of the solution is more radical at the beginning of the solution. Thus it is not so strange that the numerical solution will miss the most in the first part of the time domain. As time goes on and the solution approach the horizontal line we see that the MSE becomes extremely small, which supports the idea of the error being linked to the rate of change of the analytical solution.

2. *Neural Network*

Next we investigate the numerical solution using the neural network approach. We use the network architecture as described in the implementation section III B, and an initial learning rate of 0.05 which proved to be a sturdy choice. Naturally we want to start off by using the same discretization as used in the finite difference analysis, namely $\Delta x = 0.1$ and $\Delta x = 0.01$ with $\Delta t = \Delta x^2/2$ corresponding to the von Neumann stability criteria. However, by running the computation with both the Neumann stability criteria and equal step size $\Delta x = \Delta t$ we saw that the equal step size performed significantly better. This is shown in figure 2 for $\Delta x = 0.1$ and 2000 epochs.



(a) Neumann stability criteria: $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$.

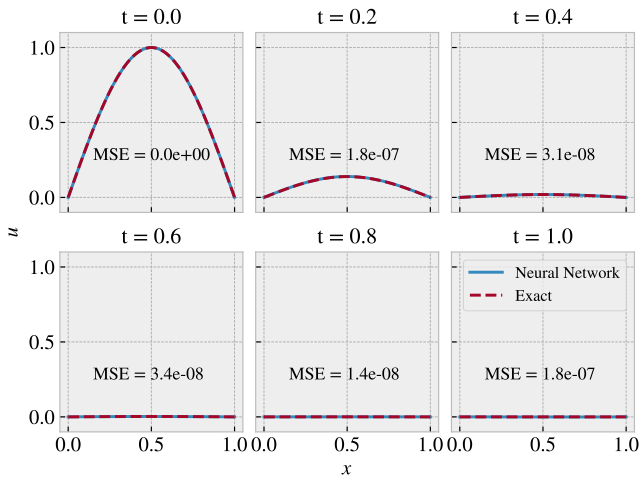
(b) Equal step size : $\Delta x = \Delta t = 0.1$.

Figure 2: Neural network solution vs. exact solution for the diffusion eq. (1) with $L = 1$ and $T = 1$. The discretization is done with $\Delta x = 0.1$ combined with $\Delta t = 5 \times 10^{-3}$ and $\Delta t = 5 \times 10^{-3}$ respectively. We used 2000 epochs with a learning rate of 0.05. MSE denotes the mean squared error between the numerical and analytical solution at a given time point.

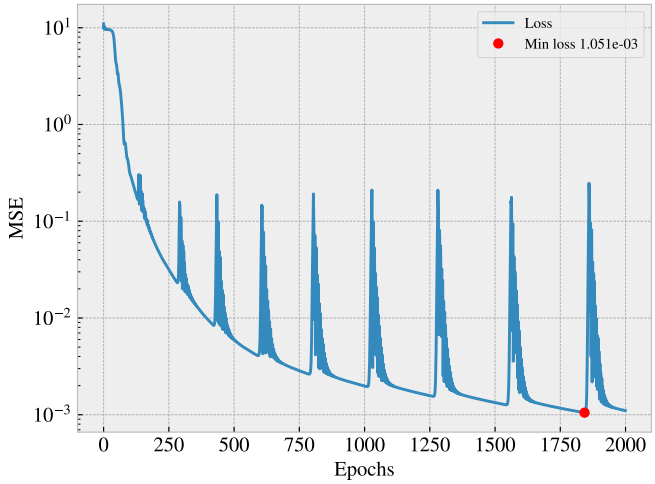
It is somewhat surprising that the equal step sizes (2b) produced a better result than the one with the Neumann criteria (figure 2a), given that the latter case has the finer discretization. A possible explanation is that the training of the network becomes somehow biased when we have an uneven number of spacial and temporal points. We did not look further into this, but it is an interesting topic for future investigation. Given this result we will use equal step sizes for the future neural network solutions.

When considering the result from the case of equal step sizes (2b), we see that the MSE is quite equally distributed across the chosen time frames ($t = 0.2, 0.4, 0.6, 0.8, 1$) with an MSE at roughly 10^{-7}. For the finite difference case using $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$ we saw a much higher difference in the error across the various time frames. This leads to the fact that the neural network performs slightly better for the first part of the time domain (where the solution changes more drastically) shown at $t = 0.2$, but worse for the remaining part of the time domain.

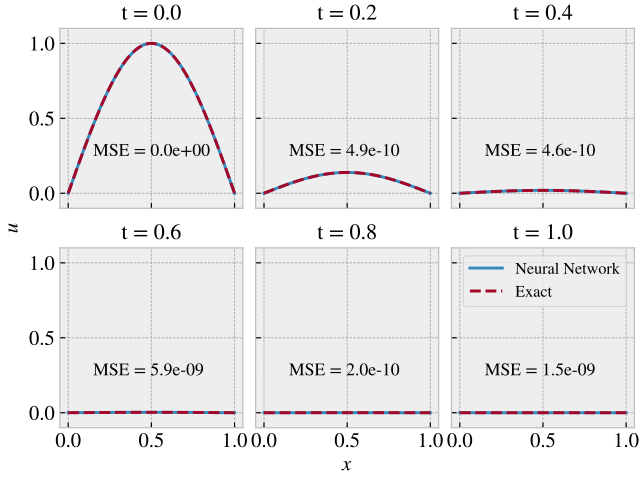
We also computed the numerical solution using $\Delta x = \Delta t = 0.01$. By continuing to use 2,000 epochs we did not see a great difference in the precision. Thus we ran the computation with 20,000 epochs in addition. Both results is shown in figure 3 together with the learning history, which show the mean squared residual as a function of epochs.



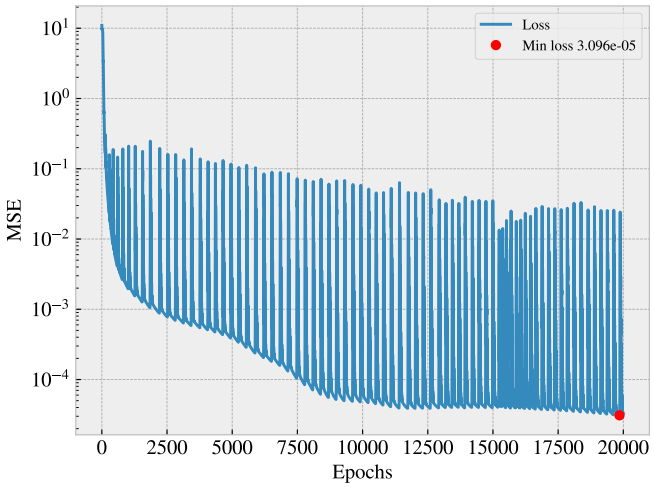
(a) Numerical solution, 2000 epochs



(b) Learning history, 2000 epochs



(c) Numerical solution, 20,000 epochs



(d) Learning history, 20,000 epochs

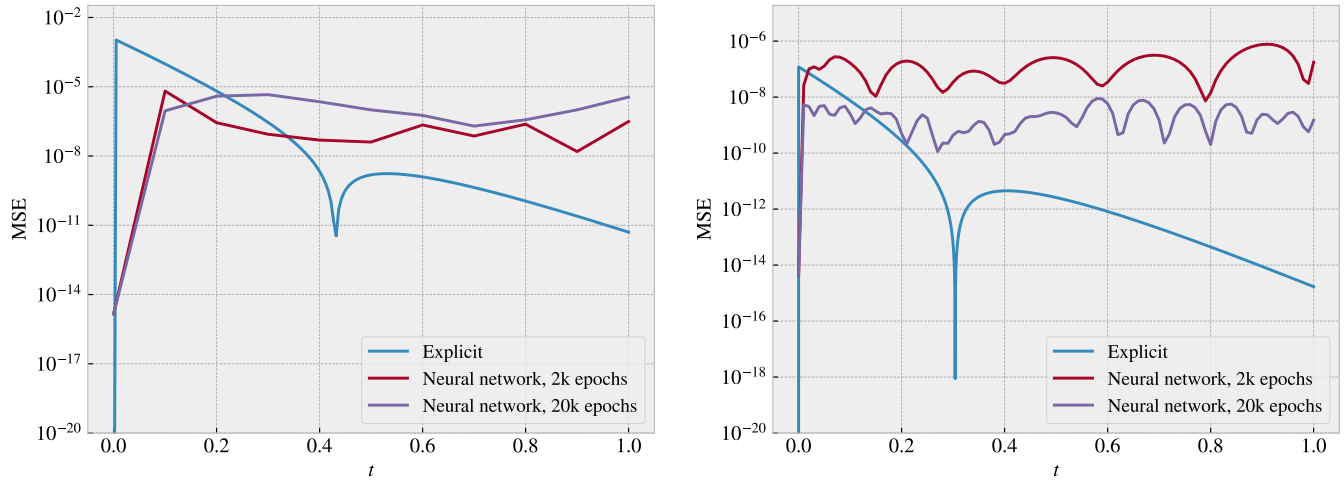
Figure 3: Neural network solution vs. exact solution for the diffusion eq. (1) with $L = 1$ and $T = 1$ using $\Delta x = \Delta t = 0.01$. We used 2,000 and 20,000 respectively epochs with an initial learning rate of 5×10^{-2}. The left column shows the solution at different timestamps and the right column shows the learning history (loss) given as the mean squared error (residual) for each epoch.

From figure 3 we again observe that the error is quite evenly distributed across the various time frames. By increasing from 2000 to 20,000 epochs we get a solution that is competitive with the finite difference result at $t = 0.2$ but is drastically outperformed at the remaining time points. Note that some runs gave an unfortunate stopping point right after the perturbation from the ADAM optimizer. In that case we simply rerun as we did not bother to implement any automatic response to avoid this. We could possible improve the neural network performance by running even more epochs, but since the computation time already far exceeds the time used for the finite difference, we did not want to pursue this. Instead we will investigate the trade-off between error and computation time in more detail in section IV A 3. When considering the learning history for 2000 epochs (3b) we notice a descending trend, similar to an exponential decay, with a seemingly regular pattern of saw tooth spiked perturbations. We are not sure about the reason for these spikes, but we suspect that it is connected to the ADAM optimizer. When looking at the learning history for 20,000 epochs (3d) we see that initial exponential decay trend is interrupted (first time around 5000 epochs) as the MSE drops considerable more.

One possible reason for the oscillating trend could be the type casting of our data points. We used float32 as casting type, which has a minimum value of around $1.7 \cdot 10^{-7}$. By calculating the mean it could be possible that some of the value is below the computer precision, being put to zero, and then the gradient has to jump accordingly. This is however highly speculative.

3. *Extended comparison of explicit and neural network*

To compare the numerical methods in more detail, we compute the MSE as a function of time for the two methods respectively. We compute the results for both 2,000 and 20,000 epochs as shown in figure 4.



(a) $\Delta x = 0.1$.

(b) $\Delta x = 0.01$.

Figure 4: Comparison of MSE for explicit and neural network vs exact solution as a function of time. We use common settings of $\Delta x = 0.01$, $L = 1$ and $T = 1$, and use $dt = dx$ for the neural network and $dt = dx^2/2$ for the finite difference solution. For the neural network we used a initial learning rate of $\eta = 0.05$ and both 2×10^3 and 2×10^4 epochs. We fixed a lower limit on the y-axis at 10^{-20} to improve the readability of the plot.

From figure 4 we see that the finite difference solution is dominant in the majority of the time domain. The only exception is in the first interval $t \approx [0, 0.1]$. Considering the MSE for the finite difference method we see that the error decrease over time as observed previously in figure 1. In addition we see a notable downwards pointing spike. When analyzing the animation we see that the numerical solutions initially decays slower than the analytical solution, before passing it later on. This explains why the MSE dips temporally. For the neural network in the case of $\Delta x = 0.01$ (4b) the MSE exhibits an interesting oscillating trend. Thus it seems like the network favored certain time points. Given that the network is for all intents and purposes a "black box", the reasoning behind what it favours is difficult to decode. For the $dx = 0.1$ case (4a) we also notice that increasing the number of epochs actually turned out to give a mainly worse result. This could easily be due to an unfortunate stopping point combined with the effect from the ADAM optimizer, but we did not spend more time trying to unravel this.

As we computed the result for figure 4 it was apparent that the finite difference solution was significantly faster computational wise. Hence one could speculate whether this lead in computation time can be utilized in the lowering of the step sizes, such that the finite difference might be able to produce a better solution with respect to used computation time. In other words, we want to investigate which method yields the lowest error with respect to computation time used. Since we already know that the finite difference computation is faster than the neural network in the calculations for figure 4 the only possible domain for which the neural network might be better is at the early time points. Hence we pick $t = 0.05$ as our point of interest, and run the two solvers for different choices of Δx, still using $\Delta x = \Delta t$ for the neural network and $dt = dx^2/2$ for the finite difference. In addition we try different number of epochs for the neural network. We record the computation time for the various combinations and plot the MSE as a function of computation time. The result is shown in figure 5.

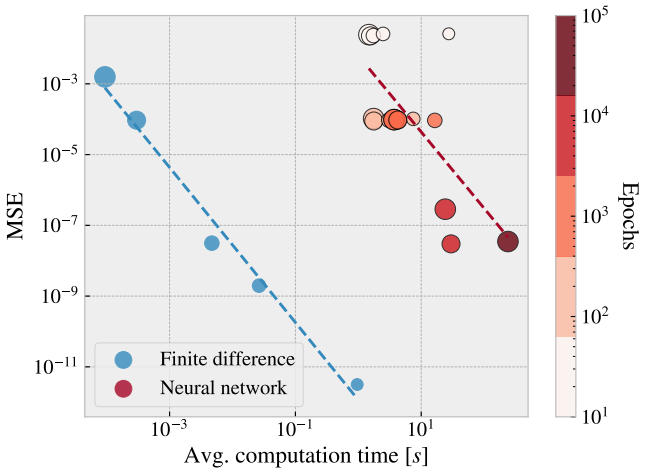


Figure 5: Mean squared error (MSE) between numerical and analytical function at $t = 0.05$ as a function of computation time for the diffusion eq. (1) with $L = 1$. The dot size represent the choice of Δx among the values: 0.1, 0.05, 0.01, 0.005 and 0.001. The shades of red colors differentiate between the choices of epochs among the values: 10, 100, 1000, 10^4, 10^5. To give a rough estimate of the trends we included a linear regression on the logarithmic scale.

From figure 5 we see that the data points corresponding to the two solvers gathers around two distinct areas. The closer the points are to origo (0,0), the better the performance, and since there is quite a severe gap between these two areas, the finite difference method appears to be the most efficient solver. By performing linear fits on the logarithmic plot we attempt to describe the relation between MSE and computation as a power law, on the form $f(x) \propto x^a$.

For the finite difference case we get a fairly good linear fit, in contrast to the neural network case where the data points seem to be more scattered. However, by considering these fits anyway it appears that the MSE decay with a similar rate for both methods. Thus it does not show any signs of the neural network efficiency being able to beat the finite difference method.

One might find a steeper fit for the neural network area by excluding certain outliers, corresponding to bad choices of number of epochs. However it still seems unreasonable to expect that the neural network could achieve the better efficiency in practice, since the point of overtaking the finite difference would be at an extremely low error and high computation time. Thus it is clear that the finite difference is superior in terms of optimizing the solution for the specific of the diffusion eq. (1). But this might not be the case for a more complex problem involving more dimensions, such as many-body problems. This is yet another topic of interest for future studies, for which we could repeat a similar analysis for more complex PDE's.

Another fact that should be mentioned is that the performance of the neural network regarding MSE vs computation time could be affected by optimizing the code. However, since we use TensorFlow, it is unrealistic to speculate in a code speed up that could close the gap between the finite difference solver and the neural network as seen in figure 5. This being said, there are still ways in which we could potentially improve the computational time. Examples of this is to explore different ways to divide the data into batches, using using different optimizers or even using GPU's.

B. Eigen value problem

In this section we study the eigenvalue problem discussed in section II C. The optimizer, number of hidden layers and nodes used in this section are the same as in the previous sections.

We generate a random symmetric 6×6-matrix and run the network over 2000 epochs with an initial learning rate of 5×10^{-4}. In figure 6 we plot the evolution of the eigenvalue found by the network over each epoch. The evolution is visualized together with the eigenvalues found by the standard eigenvalue solver.

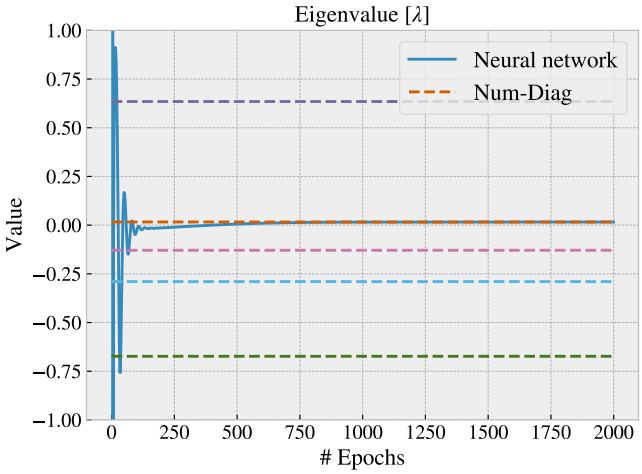


Figure 6: The figure shows the evolution of the eigenvalue produced by the neural network over 2000 epochs. The dashed lines visualizes all 6 eigenvalues of the randomly generated matrix as calculated by numpy's eigen-solver. The solid line shows the neural networks calculated eigenvalue over each epoch.

From figure 6 we see that the predicted eigenvalue oscillates drastically in the first 100 epochs. In the following epochs the network shows a clear convergence towards one of the eigenvalues of A. The final prediction from the network was $\lambda_{NN} = 0.01618$. Compared to the nearest eigenvalue from the standard eigenvalue solver, $\lambda_{np} = 0.01611$, the prediction error was 7×10^{-5}

$7 \cdot 10^{-5}$, corresponding to only $\approx 0.05\%$.

The corresponding eigenvector to the eigenvalue found in figure 6 is shown in figure 7. Here we display the evolution of the six eigenvector elements as a function of epochs.

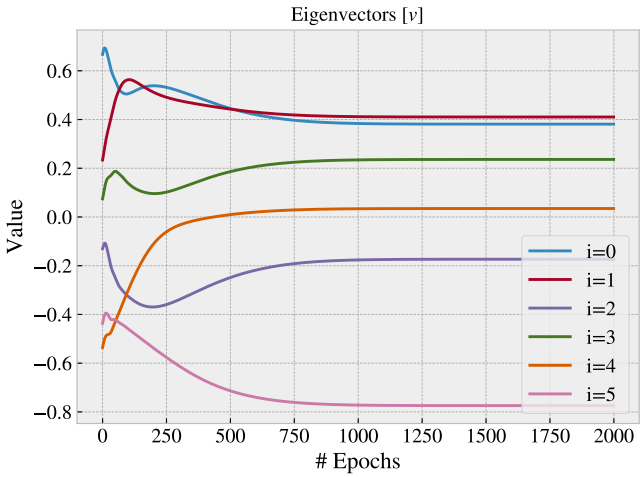


Figure 7: The figure shows the evolution of each element in the calculated eigenvector produced by the neural network, over 2000 epochs.

From figure 7 we see an initial small scramble of the eigenvector components before it converges towards a steady state. Contrary to the eigenvalues, the components of the eigenvector do not exhibit the same oscillations, but instead converges towards a solution in a much more stable fashion. The final eigenvector v_{NN} produced after 2000 epochs and the one produced from numpy's eigenvalue solver v_{np} is

$$v_{NN} = \begin{bmatrix} 0.381 \\ 0.410 \\ -0.173 \\ 0.236 \\ 0.035 \\ -0.774 \end{bmatrix}, \qquad \lambda_{np} = \begin{bmatrix} 0.381 \\ 0.410 \\ -0.173 \\ 0.236 \\ 0.034 \\ -0.774 \end{bmatrix}, \tag{19}$$

which is visualized in figure 8.

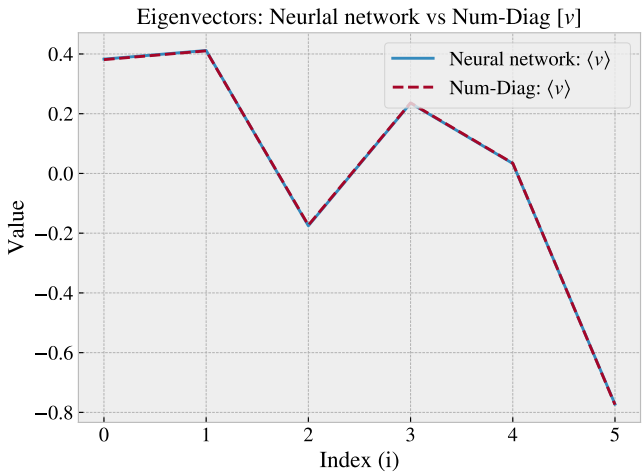


Figure 8: A comparison between the eigenvector produced by the network and the eigenvector produced by numerical diagonalization. Here index i is the i'th component of the vector.

The two vectors are identical up to an order of 10^{-3} for all but one of the elements, and the mean squared error with respect to the difference is 2.08×10^{-13}. From this we can conclude that the neural network was accurately able to calculate a eigenvector of the matrix A.

In addition we could have studied other eigenvalues by using experimenting with different initial parameter distributions (random seeds), but we leave that for further studies. An interesting note is the fact that We did not find the maximum eigenvalue as the theory predicted. One possible explanation could be that the smaller eigenvalues represent local minima in the cost function, for which we just happened to get stuck in one of those. If so, we could possibly avoid this problem by use of other optimizers or alternatively testing with a larger set of learning rates.

V. CONCLUSION

We successfully managed to produce numerical solutions for the 1D diffusion eq. (1) using both the finite difference and neural network approach. We mainly focused on the numerical solutions using a spacial step size of $\Delta x = 0.1$ and $\Delta x = 0.01$. For the finite difference method we where limited by the von Neumann criteria given as $\Delta t = \Delta x^2/2$ where the Neural network has no predefined limits. However, it turned out that the network performance was better for an equal number of spacial and temporal points, i.e. $\Delta x = \Delta t$ when $L = T$. We simply accepted this sub-result, but it would be interesting to investigate the details of this further. When comparing the numerical solutions against the analytical solution, mean squared error (MSE), for equal choices of Δx, we found the finite difference method to be generally more accurate. The only exception was for

small $t \lesssim 0.1$. However this was only achieved for a sufficient amount of epochs, 2000 epochs for $\Delta x = 0.1$ and 20,000 epochs $\Delta x = 0.01$, which required significantly more computational effort than the finite difference solutions. When computing the MSE as a function of computation time it was evident that the finite difference method is superior in terms of the error vs computation time ratio.

It should be noted that this result might only be applicable to this specific type of problem given by the simple 1D diffusion equation. For more complex multidimensional problems this might not be the case, which serve as an interesting topic for future investigations. In addition we might be able to increase the efficiency of the neural network by changing optimizer or introducing batches. On the other hand, the neural network benefits for its ability to evaluate the numerical solution in any given point in contrast to the finite difference solution which only gives you discretized point. However we did not investigate the quality of the neural network solution for points in-between and outside the training grid points, but this is yet another interesting topic for further studies.

In our study of the eigenvalue problem we were successfully calculated one the eigenvalues of a randomly generated symmetric matrix A, to an accuracy of roughly 0.05%. The network calculated the given eigenvalues to $\lambda_{NN} = 0.01618$, while numpy the calculations from numpy's standard solver produced $\lambda_{np} = 0.01611$. The corresponding eigenvector yielded a MSE on the order 10^{-13} when compared to numpy's eigenvalue solver. We found that the network did not always gravitate towards the largest eigenvalue as the theory predicted, but we suspect that this is related to the fact that solver gets stuck in local minima corresponding to the other eigenvalues. In further analysis one try using different initial distirbutions for the parameter space (different seeds) or using different optimizers as an attempt to remedy this inconsistency.

REFERENCES

[1] David Deuvenad et al. “Neural Ordinary Differential Equations.” In: (2018). DOI: https://arxiv.org/pdf/1806.07366.pdf.

[2] J. G. Charney, R. FjÖrtoft, and J. Von Neumann. “Numerical Integration of the Barotropic Vorticity Equation”. In: *Tellus* 2.4 (1950), pp. 237–254. DOI: 10.3402/tellusa.v2i4.8607. eprint: https://doi.org/10.3402/tellusa.v2i4.8607. URL: https://doi.org/10.3402/tellusa.v2i4.8607.

[3] Hans Petter Langtangen. *Finite Difference Computing with PDEs*. Texts in Computational Science and Engineering. Springer, 2016. ISBN: 978-3-319-55455-6.

[4] Zhang Yi, Yan Fu, and Hua Jin Tang. “Neural Networks Based Approach Computing Eigenvectors and Eigenvalues of Symmetric Matrix .” In: (2003). URL: https://reader.elsevier.com/reader/sd/pii/S0898122104901101?token=285C446EECA78BB06330CC19351564FA763701EC1AED7F900F0B771C78DDCC414E17F80A24E583AB289A684A01EEDC0E&originRegion=eu-west-1&originCreation=20211216101049.

Bias variance analysis

Sakarias Frette, William Hirst, Mikkel Metzsch Jensen
(Dated: December 16, 2021)

I. INTRODUCTION

In this additional paper we are going to perform an analysis of the bias-variance trade-off on a regression type problem using decision trees, boosting (gradient-boosted-trees) and feed forward neural network respectively. We are going to use the so-called Ames housing data which concerns housing sale price in connection to relevant real estate features, such square area, number of rooms etc.

In the theory section we are going to introduce the bias-variance error decomposition and the bootstrap resampling method which constitutes the foundation of the bias variance analysis. In the implementation section we are going to introduce the data set in more detail, and explain hos this is handled to fit our scope. In addition we are going to briefly explain the regression methods in use and how these is implemented in our code. Finally in the result and discussion section we present the bias-variance analysis itself.

II. THEORY

A. Bias-Variance decomposition[1]

We consider a general dataset $\mathcal{L}$ which consists of the datapoints $\boldsymbol{X}_{\mathcal{L}} = \{(y_j, \boldsymbol{X}_j), j = 0, \ldots, n-1\}$. We assume that the true target data $\mathbf{y}$ originates from a model $f(x)$ with added normal distributed noise ϵ (with variance σ^2 and zero mean) given as

$$\mathbf{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}, \tag{1}$$

The Mean Square Error (MSE) between the target data $\mathbf{y}$ and a given prediction $\tilde{\mathbf{y}}$ can be written out as

$$C(\mathbf{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2]. \tag{2}$$

We are going to show that this can be decomposed into bias, variance and the noise variance σ^2. This is known as the Bias-Variance tradeoff. We have

$$\begin{aligned}
\mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] &= \mathbb{E}[(f + \epsilon - \tilde{y})^2] \\
&= \mathbb{E}[(f + \epsilon - \tilde{y} + \mathbb{E}[\tilde{y}] - \mathbb{E}[\tilde{y}])^2] \\
&= \mathbb{E}[f^2 + f\epsilon - \tilde{y}f + f\,\mathbb{E}[\tilde{y}] - f\,\mathbb{E}[\tilde{y}] \\
&\quad + f\epsilon + \epsilon^2 - \tilde{y}\epsilon + \epsilon\,\mathbb{E}[\tilde{y}] - \epsilon\,\mathbb{E}[\tilde{y}] \\
&\quad - \tilde{y}f - \tilde{y}\epsilon - \tilde{y}\,\mathbb{E}[\tilde{y}] + \tilde{y}\,\mathbb{E}[\tilde{y}] + \tilde{y}^2 \\
&\quad + f\,\mathbb{E}[\tilde{y}] + \epsilon\,\mathbb{E}[\tilde{y}] - \tilde{y}\,\mathbb{E}[\tilde{y}] - \mathbb{E}[\tilde{y}]^2 + \mathbb{E}[\tilde{y}]^2 \\
&\quad - f\,\mathbb{E}[\tilde{y}] - \epsilon\,\mathbb{E}[\tilde{y}] + \tilde{y}\,\mathbb{E}[\tilde{y}] - \mathbb{E}[\tilde{y}]^2 + \mathbb{E}[\tilde{y}]^2].
\end{aligned}$$

By taking the expectation value of the components we get

$$\begin{aligned}
\mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] &= \mathbb{E}[(f - \mathbb{E}[\tilde{y}])^2] + \mathbb{E}[\epsilon^2] + \mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})^2] \\
&\quad + 2\,\mathbb{E}[(f - \mathbb{E}[\tilde{y}])\epsilon] + 2\,\mathbb{E}[\epsilon(\mathbb{E}[\tilde{y}] - \tilde{y})] \\
&\quad + 2\,\mathbb{E}[\epsilon(\mathbb{E}[\tilde{y}] - \tilde{y})(f - \mathbb{E}[\tilde{y}])] \\
&= \mathbb{E}[(f - \mathbb{E}[\tilde{y}])^2] + \mathbb{E}[\epsilon^2] + \mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})^2] \\
&\quad + 2(f - \mathbb{E}[\tilde{y}])\,\mathbb{E}[\epsilon] + 2\,\mathbb{E}[\epsilon]\,\mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})] \\
&\quad + 2(f - \mathbb{E}[\tilde{y}])\,\mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})].
\end{aligned}$$

Next we use that that the the noise ϵ is normally distributed with a mean equal to 0, and variance σ^2. Thus we have $\mathbb{E}[\epsilon] = 0$ and $\mathbb{E}[\epsilon^2] = \sigma^2$ which yields

$$\begin{aligned}
\mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] &= \mathbb{E}[(f - \mathbb{E}[\tilde{y}])^2] + \sigma^2 + \mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})^2] \\
&\quad + 2(f - \mathbb{E}[\tilde{y}])\,\mathbb{E}[(\mathbb{E}[\tilde{y}] - \tilde{y})].
\end{aligned}$$

Finally we have that the expectation value of an expectation value is the expectation value itself, $\mathbb{E}[\mathbb{E}(x)] = \mathbb{E}(x)$, which makes the last term in the above cancel out. Thus we get

$$\begin{aligned}
\mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] &= \mathbb{E}[(f - \mathbb{E}[\tilde{y}])^2] + \mathbb{E}[(\tilde{y} - \mathbb{E}[\tilde{y}])^2] + \sigma^2 \\
&= \frac{1}{n}\sum_i (f_i - \mathbb{E}[\tilde{y}])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}[\tilde{y}])^2 + \sigma^2 \\
&= \text{Bias}^2 + \text{Variance} + \text{Noise}.
\end{aligned} \tag{3}$$

Hence we arrived at the bias-variance relation. The decomposition shows that the MSE error can be explained in terms of

- Bias: A systematic error of the prediction.
- Variance: How much the predictions vary/fluctuates.
- Noise: The contribution from the noise.

[1] Parts of the theory discussed in the following section is heavily influenced or taken from project 1

However, in practice we don't know f_i (the un-noised data points), but in practice this is the only available quantity for which we can approximate the bias. By using our noised data points y_i as the best approximation we basically absorb the noise into the bias term. We see this by substituting f_i with y_i such that we get

$$\begin{aligned}\mathbb{E}[(y-\mathbb{E}[\tilde{y}])^2] &= \mathbb{E}[(f+\epsilon-\mathbb{E}[\tilde{y}])^2] \\ &= \mathbb{E}\left[\left(\epsilon+(f-\mathbb{E}[\tilde{y}]\right)^2\right] \\ &= \mathbb{E}[\epsilon^2]+\mathbb{E}[(f-\mathbb{E}[\tilde{y}])^2]+\mathbb{E}[-2\epsilon(f-\mathbb{E}[\tilde{y}])] \\ &= \sigma^2+\mathbb{E}[(f-\mathbb{E}[\tilde{y}])^2]+\mathbb{E}[-2\epsilon]\,\mathbb{E}[f-\mathbb{E}[\tilde{y}] \\ &\quad +\mathrm{Cov}(-2\epsilon, f-\mathbb{E}[\tilde{y}]) \\ &= \sigma^2+\mathbb{E}[(f-\mathbb{E}[\tilde{y}])^2] = \sigma^2+\mathrm{bias}^2.\end{aligned}$$

B. Bootstrap

In order to calculate the bias and variance from a single dataset we use bootstrapping. Bootstraping is a resampling method for which we resample the data by drawing new data points from with replacement from our available data set. Lets say we have a $N = 5$ dataset given as $\{1,2,3,4,5\}$. A possible bootstrap sample could then be $\{1,2,2,4,5\}$. For a sufficiently large data set there is practical no chance of getting the same sample twice, and thus this will generate slightly different data set for each resample. By drawing B bootstrap samples we can perform B predictions for which we can analyze the bias and variance in the prediction results according to equation 3.

III. IMPLEMENTATION

The numerical implementations can be found on our github.

A. Data

We are going to address the so-called Ames Housing dataset, which is an alternative to the well known Boston housing data. The data set contains 80 features specifying different aspect of real estates and the corresponding sale price (target value). It has a total of 1460 entries which we divide into train and test data with a 80-20% split. We are going to focus on regression methods and thus we trim away all features with a classification structure, such as categories of neighbourhoods and house styles. Notice that while binary classes could be quantized rather easily, the multi-class features would require a subjective ranking when transforming from categories to numbers. Thus we decide to exclude all category based features and we are left with 35 features. Among those we calculate the correlation value (Pearson's Correlation Coefficient) between each feature and the target value. In order to reduce the data set even more we pick only the top ten features, which is listed in the following, ordered from highest to lowest correlation number.

1. YearRemodAdd: Remodel date (same as construction date if no remodeling or additions)
2. YearBuilt: Original construction date.
3. TotRmsAbvGrd: Total rooms above grade (does not include bathrooms).
4. FullBath: Full bathrooms above grade.
5. 1stFlrSF: First Floor square feet.
6. TotalBsmtSF: Total square feet of basement area.
7. GarageArea: Size of garage in square feet.
8. GarageCars: Size of garage in car capacity.
9. GrLivArea: Above grade (ground) living area square feet.
10. OverallQual: Rates the overall material and finish of the house:
    - 10 Very Excellent
    - 9 Excellent
    - 8 Very Good
    - 7 Good
    - 6 Above Average
    - 5 Average
    - 4 Below Average
    - 3 Fair
    - 2 Poor
    - 1 Very Poor

We see that the first nine features essentially concerns age, area and number of rooms. Thus one could argue that the most effective data set reduction could be done by combining some of these into single categories like total area or total number of rooms. Another effective way to reduce the number of features properly is done by principal component analysis. However, for this paper we are not particular interested in maximizing efficiency or prediction quality but rather to compare performance and bias-variance trade-off across multiple methods. Thus we are satisfied with the top ten chosen features.

Since some of the features takes relatively big values numbers like year stamps and area, while other deals with small quantities like number of rooms, we perform a min-max scaling which scales all columns of the design matrix to the interval $[0,1]$. This is done by shifting the mean and scaling by a factor such that the minimum takes value 0 and maximum takes value 1.

B. Regressions method

We are going to utilize three different methods for the regression problem based on the housing data. We choose to use decision trees, boosting (gradient-boosted-trees), and feed forward neural network.

1. Decision Tree

In the regression case a decision tree divides the predictor space (the set of possible values $x_1, x_2, \ldots, x_p$) into J distinct and non-overlapping regions $R_1, R_2, \ldots, R_J$. For every new observation that falls into the region R_j we simply predict the mean value $\bar{y}_{Rj}$ of the training values in R_j. Since it becomes computational challenging to consider all possible partitions we are going to use a top-down approach. That is, we will consider the best split at each step rather than looking ahead and picking the splits that will lead to a better tree in the future. For a given feature j we consider a cut-point s which splits the predictor space into two regions

$$R_1 : \{X | x_j < s\},$$
$$R_2 : \{X | x_j \geq s\},$$

where X denotes all the features in the form of the design matrix. We want to choose the feature j and cut point s such that we minimize the MSE given as

$$\text{MSE}(j, s) = \sum_{i:x_i \in R_1} (y_i - \bar{y}_{R1})^2 + \sum_{i:x_i \in R_2} (y_i - \bar{y}_{R2})^2$$

For N data points there is a maximum of $N - 1$ cut points which will make an unique contribution to the error. Thus for p features the total combination of splits is proportional to $p \times N$. Thus we can actually effort to go through all of them and pick the combination of j and s which minimize the MSE. After the first cut the tree branches into two regions for which we can repeat the splitting process again. We continue to split the branches until the max depth is reached, or when a certain regions contains less point than some predefinied limit. In all cases we have to stop splitting a certain branch when there is only one training point left in the region. We denote the end of a branch as a leaf.

We use the DecisionTreeRegressor class from sklearn.tree to perform the training. Our implementation can be found on our github in bias_variance_analysis.py which is executed through decision_tree.py.

2. Boosting

The basic idea of boosting or gradient boosting is to combine weak classifiers in order to create a combined strong classifier. We are going to use a shallow decision tree as our weak classifier and thus we might denote this method as gradient-boosted-trees. We decide on a number of weak estimators M for which we want to create a model on the form $\tilde{y} = F(x)$. We build up the model in stages $m \in 1, 2, \ldots, M$ by adding a new estimator h_m such that

$$F_{m+1} = F_m(x) + h_m(x) = y \quad \Longleftrightarrow \quad h_m(x) = y - F_m(x)$$

For each step we want to choose $h_m(x)$ such that we minimize the MSE, by fitting h to the residual $y - F_m(x)$. Thus each added estimator tries to correct the errors from the previous model.

For the implementation of this method we use AdaBoostRegressor from sklearn.ensemble. Our implementaiton can be found on our github in bias_variance_analysis.py which is executed through boosting.py.

3. Feed Forward Neural Network

For details about the Feed Forward Neural Network we refer to a previous paper that we made about the subject. For this implementation we used Sequential from tensorflow.keras which can be found on our github in bias_variance_analysis.py which is executed through FFNN.py.

IV. RESULTS AND DISCUSSION

A. Decision Tree

We begin by analyzing the bias-variance trade-off for the decision tree. Here we identify the tree depth to be the complexity variable. For this we could effort to use 5000 bootstrap samples. The result is shown in figure 1.

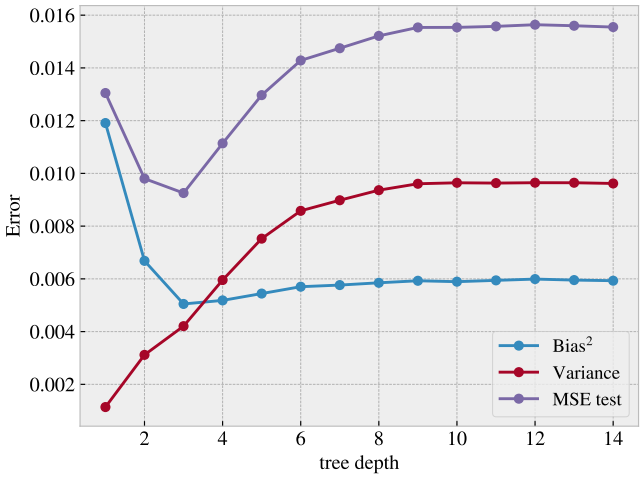
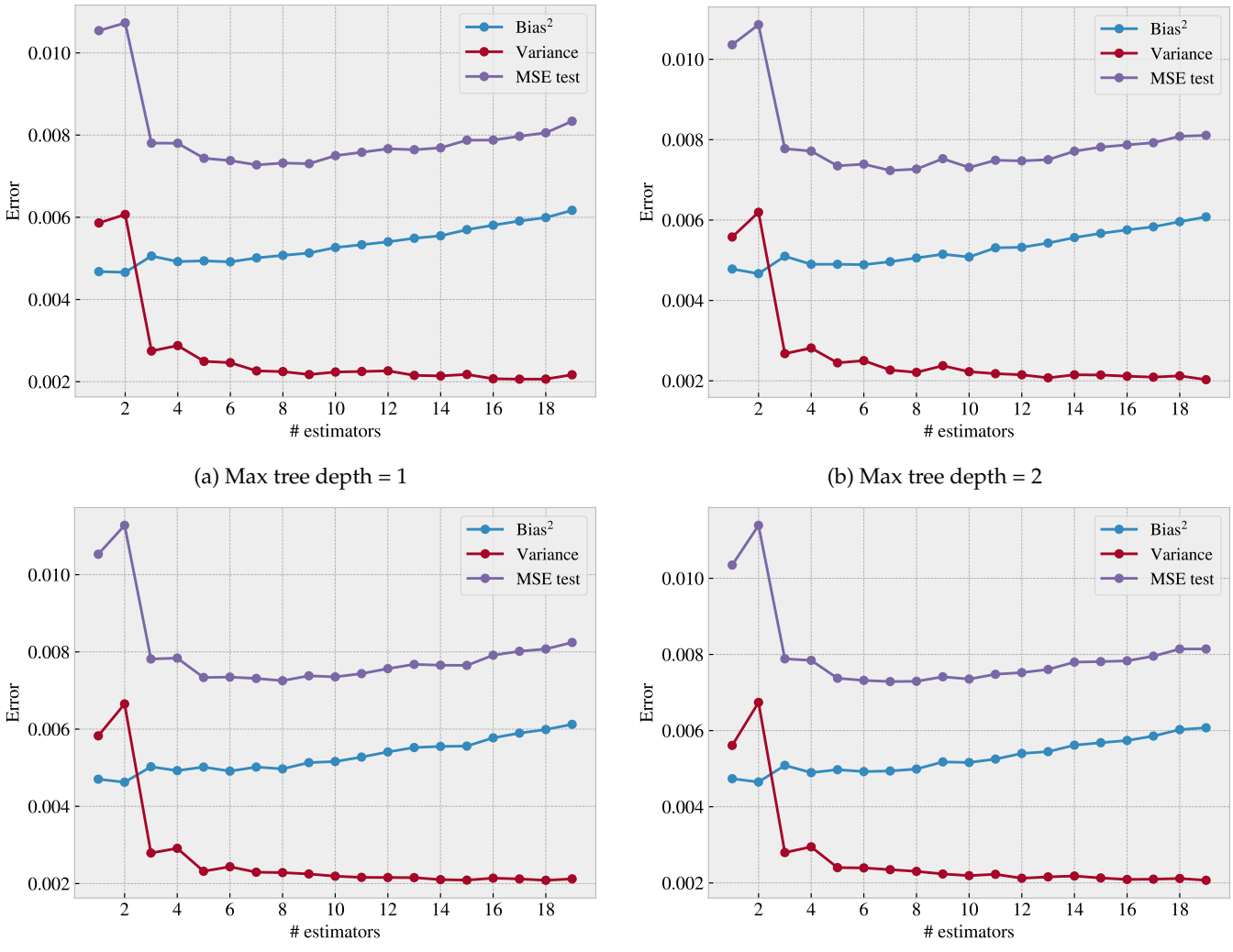


Figure 1: Bias-variance trade-off for a decision tree with a varying max tree depth. For this analysis we used 5000 bootstrap samples.

From figure 1 we see a quite typical example of the bias-variance trade-off where the initial increase in complexity results in a reduce in bias and a simultaneously increase in the variance. The sweet spot in our case is found to be at max tree depth of 3 for which the total test MSE is at its lowest. While the variance increase somewhat monotonically towards a plateau, the bias hits a minimum at tree depth = 3 and then slowly increases to another plateau. Since both the bias and variance plateaus it does not show any promise of better predictions for a further increased complexity.

B. Boosting

Next we expand on the single decision tree model by introducing an ensemble of trees. From the previous result (see figure 1) we found the best classifier with tree depth = 3 thus we try to train the model with (weak) classifiers of depth 1, 2, 3 and 4. We identify the number of estimators in the ensemble as the complexity in this case. This computation was more expensive and we had to settle on 100 bootstrap samples. The results is shown in figure 2, for which we see little to no difference regarding the choice of the classifier max depth. For all cases we see that the variance decrease with complexity while the bias increases. This is an opposite relation compared to our observations regarding a single decision tree for increasing depth. Thus when expecting the values further we see that bias is approximately on the same size as observed for the single decision tree, while the variance drops to about 20% of the variance plateau for the single decision tree case. Overall the boosting seems to decrease the variance which results in the lowest MSE given at approximately 0.075 compared to about 0.095 for the single decision tree.



(a) Max tree depth = 1

(b) Max tree depth = 2

(c) Max tree depth = 3

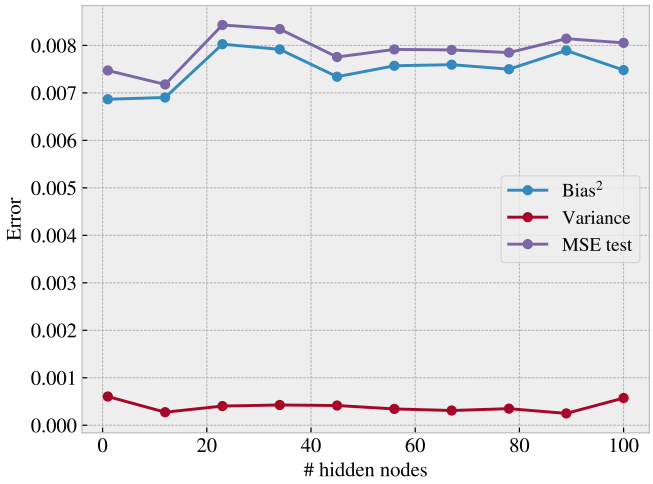
(d) Max tree depth = 4

Figure 2: Bias-variance trade-off for boosting (gradient-boosted-trees) with a varying number of estimators for a (weak) classifier of max depth 1,2,3 and 4 respectively.. For this analysis we used 100 bootstrap samples.

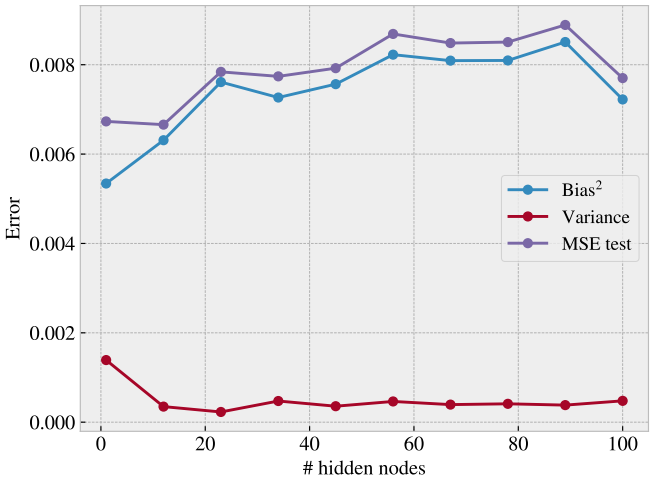
C. Feed Forward Neural Network

Finally we perform a similar bias-variance analysis for the feed forward neural network. Here we have two main candidates for the complexity: Number of hidden nodes, and number of hidden layers. Thus we perform a multiple of analysis, first with fixed number of layers and varying number of hidden nodes per layer (same for every layer) and then vice versa. We use again 100 bootstrap samples. The results are shown in figure 3 and 4. Watching the case of fixed number of hidden layers and increasing number of nodes (figure 3) we see that the bias carries most of the error. There seems to be a common trend of the bias dropping slightly while the bias increases slightly over the first few increments of the number of nodes. Apart from that the individual fluctuations does not seem to follow any straight forward trend. Going on to the case of fixed number of nodes and variable number of layers (figure 4) we still observe that most of the error lies in the bias. It is difficult to state anything about the trend as the complexity increases. Considering both result we found a few combinations where the total test MSE just dips below 0.006. While the bias seems to be about the same level as seen for the two previous method the difference appears to be the ability to reduce the variance.

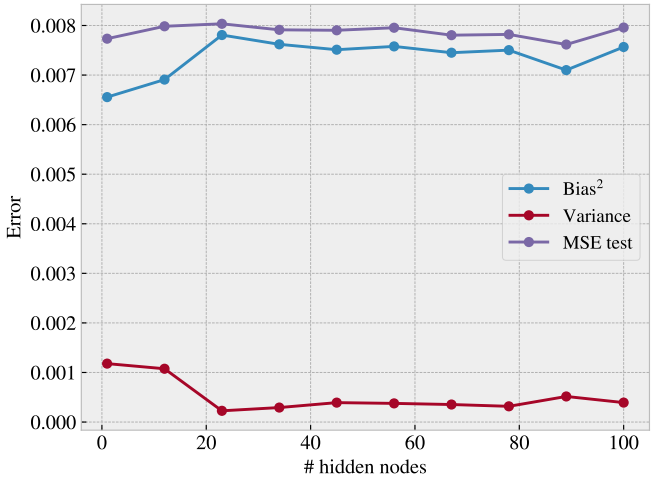
We would expect these results to be heavily influenced by the specifics of the data set, and thus it could be interesting to perform a similar investigation for a completely different data set.



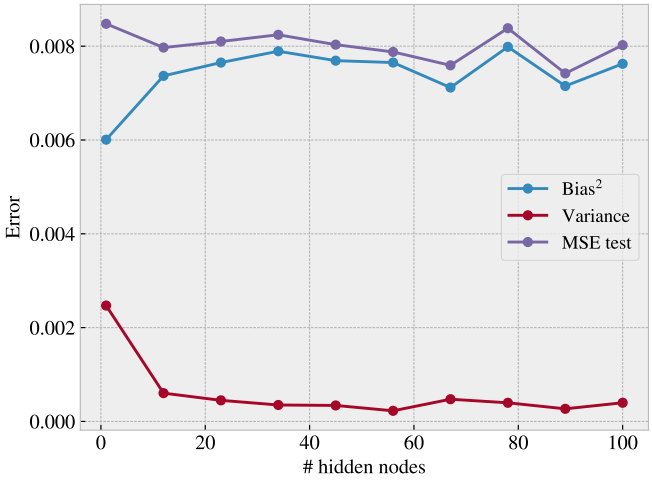
(a) Number of hidden layers = 1



(b) Number of hidden layers = 3

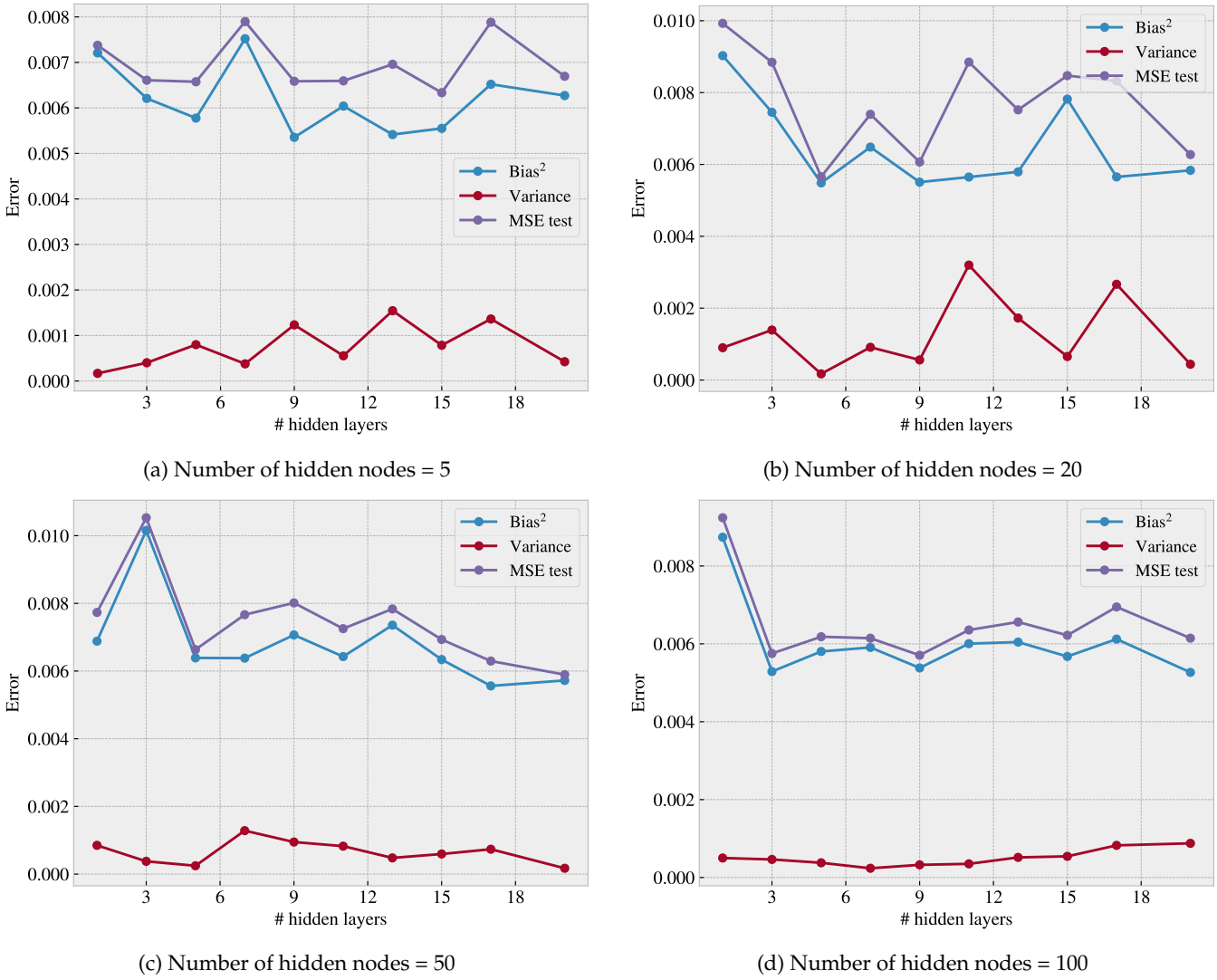


(c) Number of hidden layers = 5



(d) Number of hidden layers = 10

Figure 3: Bias-variance trade-off for a feed forward neural network with a varying number of hidden nodes for a network of 1, 3, 5 and 10 hidden layers respectively. For this analysis we used 100 bootstrap samples.



(a) Number of hidden nodes = 5

(b) Number of hidden nodes = 20

(c) Number of hidden nodes = 50

(d) Number of hidden nodes = 100

Figure 4: Bias-variance trade-off for a feed forward neural network with a varying number of hidden layers for a network with 5, 20, 50 and 100 hidden nodes on each hidden layer respectively. For this analysis we used 100 bootstrap samples.

V. CONCLUSION

When comparing the various method we found that the single decision tree followed a somewhat typical bias-variance trade-off where increasing the complexity would decrease the bias with a cost of the variance going up. For the boosting (boosted-trees) we more or less found an opposite relation where an increase in complexity would decrease the variance with the cost of the bias going up. The bias of these two methods was about the same size, however the variance was clearly reduced by boosting. When comparing the lowest MSE among the tested values the boosting method gave 0.075 while a single decision tree gave 0.095. When investigating the feed forward neural network we found an even greater reduction of the variance such that most of the error was carried by the bias. While we did not see any clear connections between the complexity, and the bias and variance respectively, we see about the same bias level as the other two methods which resulted in a slightly better lowest MSE being just below 0.006.

We suspect that our results is quite heavily influenced by the specifics of the data set, and thus it would be interesting to perform a similar investigation on a different data set.

In a future study of the problem it would be interesting to repeat the analysis for a more complex problem. Given that we found an ideal tree-depth of 3, one might be interested in studying if our findings are specific to this relatively "simple" problem or if they are in did general.