Methods for Numerically Solving Differential Equations

Carl Fredrik Nordbø Knutsen, Halvor Tyseng, Benedict Leander Skålevik Parton

December 2022

In this report we use the Forward Euler algorithm and neural networks to solve differential equations.We first consider the one-dimensional diffusion equation, and solve it using both forward Euler and a neural network on a coarse discretization ($\Delta x = 0.1, \Delta t = 0.1$). We find that the neural network outperforms forward Euler when the stability criterion for forward Euler is not satisfied. By decreasing Δt to 0.001, forward Euler outperforms the neural network. For a more fine-grained discretization ($\Delta x = 0.01, \Delta t = 0.00001$), we find that both the neural network and forward Euler provide good approximations, with forward Euler having a lower mean squared error again. For the second part of this report, we examine a dynamical system whose equilibrium points are the eigenvectors of a real, symmetric matrix A. We then proceed to solve the differential equation by applying the same the forward Euler scheme and neural networks. Using the Forward Euler method we find that the solution converges to the eigenvector with the highest eigenvalue. To find more eigenvalues than just the highest, we use a coarse discretization and apply our neural network algorithm. We observe occasional convergence to high-cost solutions that are non-stable equilibrium points in the analytical solution. Using this method, we find 4 eigenvalues of a 6×6 symmetric, real matrix. Finally we reduce the eigenvalue problem to a minimization task that we solve with a neural network approach. Doing so, we manage to compute all 6 eigenvectors of a 6×6 real, symmetric matrix.

1 Introduction

Both ordinary and partial differential equations appear in mathematics, physics, engineering, and medicine, along with many other fields. They comprise a crucial part in applied science, where everything from building construction to epidemiological modelling - and thus much of modern society - depends on them. However, many differential equations prove difficult, or even impossible, to solve analytically. In these situations, numerical methods can be better suited for finding a solution. In this report, we will therefore employ numerical methods to tackle two problems that can be expressed in terms of differential equations.

We will first solve the one-dimensional diffusion equation using the forward Euler algorithm. This is a simple explicit differentiation scheme that can be used to approximate differential equations. Next, we will employ a neural network based approach to solve the same problem. This approach in essence entails modelling the diffusion equation in terms of a neural network N, that we optimize by minimizing a cost function. The cost function is given by the squared difference of the left hand side and right hand side of the diffusion equation. We will then compare these results with each other and the analytical solution.

Then we proceed to the problem of computing eigenvalues and eigenvectors. We will limit ourselves to considering the eigenvalues and eigenvectors of a symmetric, real matrix. By stating our problem in terms of a dynamical system where the eigenvectors correspond to the equilibrium points of the differential equation, we can derive a method of computing eigenvectors (Yi et al., 2004). Finding eigenvalues of a matrix is an important problem in many scientific fields, such as physics. For example would the eigenvalues of the Hamiltonian operator correspond to the energy of a specific state - the eigenstate. Eigenvalues also crop up in areas like technology, in both image compression and search engine ranking, and thus constitute an important part in the functioning of systems like the internet. We will use a random 6x6 matrix and compare the eigenvalues found with neural networks, with numerically calculated ones using the python library Numpy (Harris et. al., 2020).

2 Methods

2.1 Differential Equations

We will mainly consider two approaches for generating approximate solutions to differential equations.

2.2 Forward Euler Algorithm

The forward Euler algorithm is an explicit scheme for discretizing a differential equation and generating an approximate solution. The algorithm consists of portioning the space of the variables in the differential equation into steps, and iteratively finding the solution for the next time-step by utilizing the solution for the previous time-step. Given a differential equation of the form $f(t, u, u^{(2)}, ..., u^{(n-1)}) = y^{(n)}$, the forward Euler algorithm then finds the solution over a time-step Δt as:

$$u_{i+1} = u_i + \Delta t \cdot f(t_i, u_i, u_i^2, ..., u_i^{(n)}), \tag{1}$$

where the subscripts refer to which time-step they are evaluated at.

2.3 Solving Differential Equations Using Neural Networks

Neural networks are incredibly powerful tools to solve complex problems, such as differential equations. Neural networks are trained by feeding it with a training dataset, and then finding the derivative of a given loss function with respect to each weight and bias in the network, which is in turn adjusted to reduce said loss.

In the context of differential equations, on the other hand, we don't have a target value to compare with a predicted value to find the loss (indeed, this target value is what we are trying to uncover), but we have a criterion $f(t, u, u^{(2)}, ..., u^{(n)}) = 0$. Thus the loss can be evaluated as for instance the mean squared error between $f(t, u_{trial}, u_{trial}^{(2)}, ..., u_{trial}^{(n)})$ and 0, where u_{trial} is the function u modelled by the neural network.

2.4 The Diffusion Equation

We will limit ourselves to studying the one-dimensional version of the diffusion equation

2.4.1 The One-dimensional Case

The one-dimensional version of the diffusion equation is given by

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

In particular, we will study a system defined for $x \in [0,1]$ and $t \in [0,\infty)$. We use Dirichlet boundary conditions, that is

$$u(0,t) = u(1,t) = 0, \tag{3}$$

for all t. Furthermore, we will use the initial condition

$$u_0(x) = u(x,0) = sin(\pi x). \tag{4}$$

We now make the observation that the analytical solution is given by

$$u(x,t) = \exp\left(-\pi^2 t\right) \cdot \sin(\pi x). \tag{5}$$

This clearly upholds the boundary conditions along with aligning with the correct initial condition. We also see that

$$\frac{\partial^2 u(x,t)}{\partial x^2} = -\pi^2 \exp\left(-\pi^2 t\right) \cdot \sin(\pi x) = \frac{\partial u(x,t)}{\partial t}, \tag{6}$$

so our choice of u satisfies the diffusion equation. We will use the analytic solution to compare with our numerical estimates and evaluate their performances.

2.4.2 Solving the Diffusion Equation Using the Forward Euler Algorithm

We may use the forward Euler algorithm to solve the diffusion equation; first, we divide the length of the rod into I pieces x_i of length Δx, and likewise we divide the time into J parts t_j of length Δt. The derivative $\frac{\partial^2 u(x,t)}{\partial x^2}$ may be approximated in this partitioning by:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2} \tag{7}$$

We are looking to find the solution to equation 2; the diffusion equation in one dimension. Setting up the forward Euler algorithm from equation 1, we have:

$$u_{i+1} = u_i + \Delta t \cdot \frac{\partial u}{\partial t}, \tag{8}$$

and through the equality in the diffusion equation, as well as our previous approximation, this becomes:

$$u_{i+1} = u_i + \Delta t \cdot \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}. \tag{9}$$

This requires that we know the initial condition at time t_0, which we have defined for our problem with equation 4. We also need to make sure that the we find the solution stably. If the time-step is too large, the solution we find using equation 9 might grow uncontrollably large, whereas the real solution to the PDE converges. The stability criterion, which ensures that the solution we find does not diverge like this, requires that $\frac{\Delta t}{\Delta x^2} \leq 0.5$.

2.4.3 Solving the Diffusion Equation Using a Neural Network Approach

To solve the diffusion equation with a neural network, we model $u(x,t)$ in the following way:

$$u_{trial}(x,t) = (1-t)u_0(x) + tx(1-x)N(x,t), \tag{10}$$

where u_0 is the initial condition and $N(x,t)$ is the output of a neural network. This choice of trial function forces the predicted solution to agree with both the initial condition and the Dirichlet boundary conditions. Clearly, $u_{trial}(x,0) = u_0(x)$, and $u_{trial}(0,t) = u_{trial}(1,t) = 0$.

Then we want to solve the problem of training the neural network in a way that makes it output sensible values. We do this by considering the cost function

$$\mathcal{C}(x,t) = \left(\frac{\partial^2 u_{trial}(x,t)}{\partial x^2} - \frac{\partial u_{trial}(x,t)}{\partial t} \right)^2. \tag{11}$$

We define our loss $\mathcal{L}$ as mean costs over all the data points. This is a sensible choice of loss, since it coincides reducing the difference between the left hand side and right hand side of the diffusion equation.

2.5 Eigenvalues and Eigenvectors

Suppose we have a square matrix $A \in \mathbb{C}^{N \times N}$. Then the eigenvectors of A are the non-zero vectors $v \in \mathbb{C}^N$ that satisfy

$$Av = \lambda v \tag{12}$$

for some constant lambda. This constant λ is the eigenvalue corresponding to the eigenvector v. The set of eigenvectors that have a given eigenvalue is called the eigenspace of λ, which we will denote V_λ.

Given an eigenvector v, its eigenvalue can be computed by

$$\lambda = \frac{v^T A v}{v^T v}. \tag{13}$$

This follows from multiplying equation 12 with v^T from the left and solving for λ.

2.5.1 Real, Symmetric Matrices

In this report, we will limit ourselves to determining the eigenvalues of real, symmetric matrices. That is, matrices $A \in \mathbb{R}^{N \times N}$ that satisfy the relation

$$A = A^T. \tag{14}$$

Real, symmetric matrices possess some properties that make them especially interesting to study. Namely, the spectral theorem for real, symmetric matrices states that every real, symmetric matrix A can be written on the form

$$A = PDP^{-1} \tag{15}$$

where the columns of $P \in \mathbb{R}^N$ are a basis of eigenvectors for the column space of A, and $D \in \mathbb{R}^N$ is a diagonal matrix whose entries are the corresponding eigenvalues. Finding such a basis of eigenvectors can make it easier to analyze the properties of the matrix. Furthermore, using this basis can prove quite computationally beneficial.

2.5.2 Rephrasing the Problem in Terms of a Differential Equation

In a 2004 paper, Yi et. al. showed that the problem of computing the maximal eigenvalues of a real, symmetric matrix can be stated in terms of determining the equilibrium points of the following dynamics:

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)). \tag{16}$$

Here $x : \mathbb{R}_{\geq 0} \to \mathbb{R}^N$ describes the state of the system, and f is given by

$$f(x) = ((x^T x)A + (1 - x^T Ax)I)x. \tag{17}$$

Furthermore, the set E of equilibrium points of $x(t)$ corresponds to the eigenvectors of A along with the zero vector, that is

$$E = \bigcup_{\lambda} V_\lambda. \tag{18}$$

This observation enables for a reduction from the problem of computing eigenvectors to the problem of determining the equilibrium points of equation 21.

2.5.3 A Forward Euler Discretization for the Differential Equation

Discretizing $x(t)$ with the forward Euler scheme, we get that

$$x_{n+1} = x_n + \Delta t(-x_n + f(x_n)). \tag{19}$$

And since we know that the equilibrium points are the eigenvectors of A, one can do the following: compute a forward euler approximation of the differential equation, and check for convergence. If it converges, it will have converged to an equilibrium point, meaning that we have found an eigenvector.

In particular, Yi et. al. showed that given a nonzero $x_0 \in \mathbb{R}^N$, the approximated solution of equation 21 will converge to an eigenvector of A. And if λ_1 is the largest eigenvalue of A, then the solution will converge to an eigenvector in V_{λ_1} if x_0 is not orthogonal to V_{λ_1}. Since the smallest eigenvalue of A is the largest eigenvalue of $-A$, it can also be computed by applying the same algorithm on $-A$.

2.5.4 Solving the Differential Equation with a Neural Network

Equation 21 can also be solved using a neural network. To do this, we define the following trial value of x

$$x_{trial}(t) = x_0 e^{-t} + (1 - e^{-t})N(t), \tag{20}$$

where x_0 is the initial condition and $N(t)$ is the output of the neural network. This trial function satisfies $x_{trial}(0) = x_0$ as wanted.

By equation 21, we take

$$\mathcal{C}(t) = \left(\frac{dx_{trial}(t)}{dt} - (-x_{trial}(t) + f(x_{trial}(t)))\right)^2. \tag{21}$$

as our cost, and define our loss as the mean cost over time points considered. Lowering this loss means achieving a better approximation of the 21. With a sufficiently good approximation, equilibrium points can again be determined by taking whatever value x_{trial} approaches with increasing t.

2.5.5 Determining the Equilibrium Points Through a Minimization Problem

We will also consider another approach, where we entirely ignore the fact that 21 is a differential equation. Since we wish to find the equilibrium points, we can simply set the left hand side of the equation to be zero. Then we want to find a value of x that satisfies

$$0 = -x + f(x). \tag{22}$$

We take the approach of generating a trial value for x by

$$x_{trial} = \frac{x_0 + N(x_0; \theta)}{\|x_0 + N(x_0; \theta)\|} \tag{23}$$

where x_0 is our initial guess for the eigenvector and N is a neural network with parameters θ. Since we get eigenvectors at the equilibrium $x_{trial} - f(x_{trial}) = 0$, we choose

$$\mathcal{L}(x_{trial}) = \frac{1}{N}\|x_{trial} - f(x_{trial})\|^2 \tag{24}$$

as our loss function. Minimizing this loss function, we can approximate some eigenvector x with x_{trial}.

After computing the first k linearly independent eigenvectors, we choose an x_0 that is orthogonal to the k eigenvectors. We also initialize the weights of the neural network with low values. We do this to improve convergence towards not already computed eigenvectors.

3 Results and Discussion

The source code is linked in appendix A.

3.1 The Diffusion Equation

We will now examine how forward Euler and the neural network based model fare in approximating the diffusion equation, for three different discretizations. First, we will use $\Delta x = 0.1$ and $\Delta t = 0.1$. Then we keep using $\Delta x = 0.1$, but decrease Δt to 0.001. Finally, we use a discretization of $\Delta x = 0.01$ and $\Delta t = 10^{-5}$.

3.1.1 The First Discretization

The analytical solution to the diffusion equation as given by equation 5, can be seen in figure 1. We will use the analytical solution to benchmark our forward Euler and FFNN approximations.

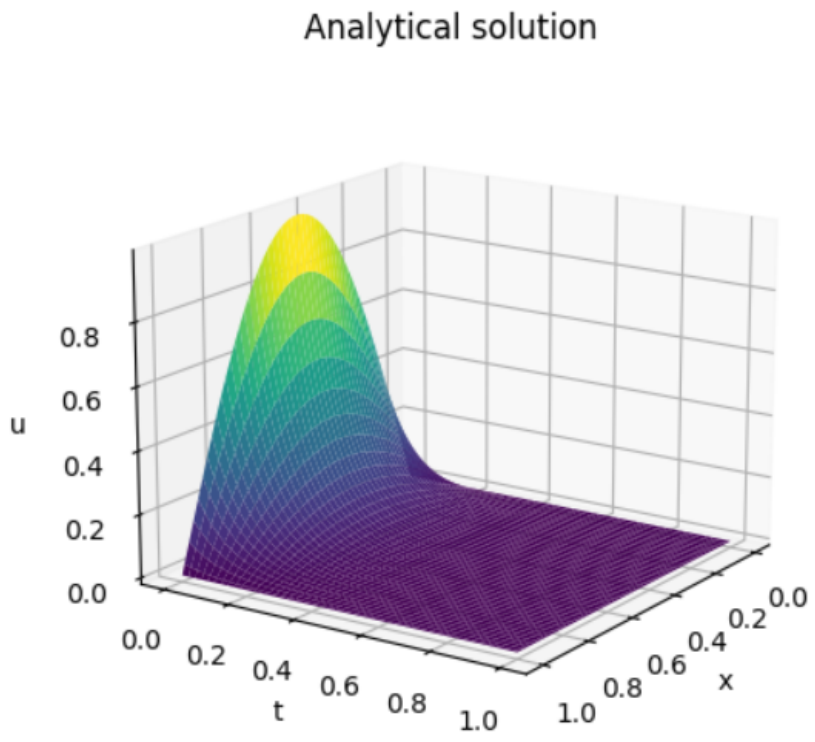

Figure 1: Here we see the analytic solution to the one-dimensional diffusion equation, given by equation 5.

We will first approximate the diffusion equation over a coarse discretization, where $\Delta x = 0.1$ and $\Delta t = 0.1$. Using this discretization, the stability criterion of forward Euler is not met. Simulating the differential equation using the forward Euler discretization, we get a mean squared error of $6.139 \cdot 10^{-3}$ and a maximum absolute error of 0.3516. The forward Euler approximation can be seen in figure 2. We see that such a large time step leads to a poor approximation.

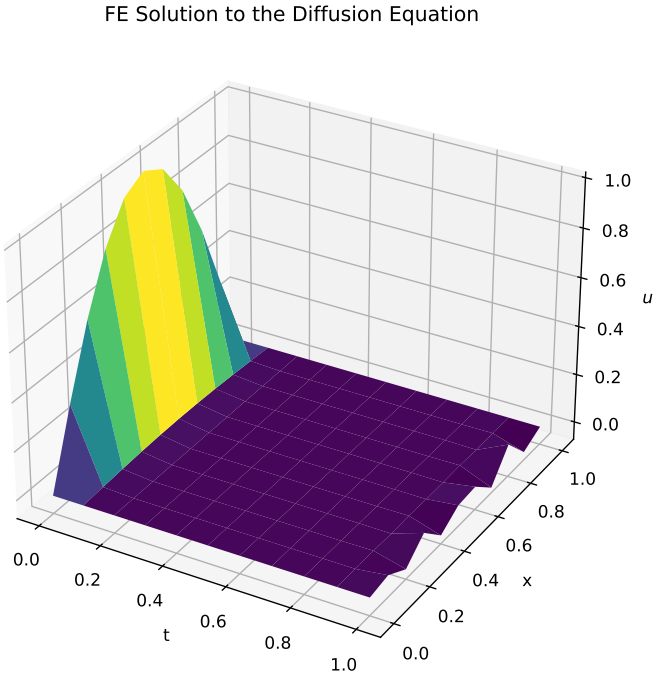

Figure 2: This figure shows the forward Euler solution to the diffusion equation for $\Delta x = 0.1$ and $\Delta t = 0.1$. It flattens out after just one timestep.

Next, we compare these results with those of an FFNN. We fix our neural network to have 3 hidden layers with 128 nodes each, and use

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{25}$$

as the hidden layer activation function, as it is well-documented to perform well (Karlik & Olgac, 2011). We then perform a hyperparameter grid search over the learning rate η and regularization parameter λ, over 1000 epochs of training. The hyperparameters tested in the search can be found in table 1. We tested each parameter configuration 20 times, and computed the average loss. We found that a learning rate $\lambda_{opt} = 10^{-3}$ and a regularization of $\eta_{opt} = 10^{-7}$ were optimal, yielding an average cost of $4.037 \cdot 10^{-3}$. 2. We then use the optimal parameters $\lambda_{opt} = 10^{-3}$ and $\eta_{opt} = 10^{-7}$ when training our model. The solution over the discretization can be seen in figure 3. Comparing with the analytical solution, we find that our neural network approximation has a mean squared error of $1.651 \cdot 10^{-4}$ and a maximum absolute error of $6.196 \cdot 10^{-2}$.

η	10^{-5}	10^{-4}	10^{-3}	10^{-2}
λ	0	10^{-7}	10^{-6}	10^{-5}

Table 1: Each combination of values of η and λ, as can be seen in this table, were tested in the grid search.

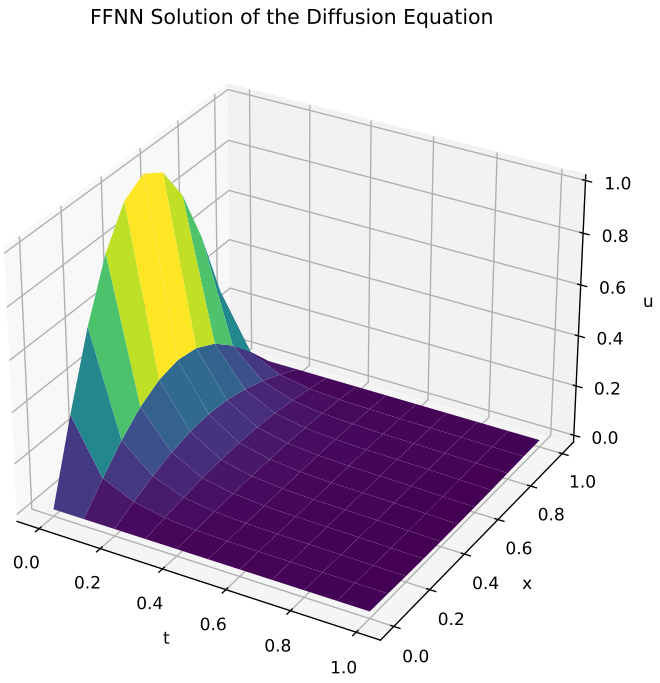

Figure 3: This figure shows the feedforward neural network approximation of the diffusion equation for $\Delta x = 0.1$ and $\Delta t = 0.1$.

.

Overall, we see that the neural network performed far better than forward Euler over a coarse discretization of $\Delta x = 0.1$ and $\Delta t = 0.1$. We note that this is a quite sensible result; since $\dfrac{\Delta t}{\Delta x^2} = 1 > 1/2$, the stability criterion is not met for forward Euler, and we expect a poor performance.

3.1.2 The Second Discretization

We repeat the same analysis for $\Delta x = 0.1$ and $\Delta t = 0.001$, such that $\dfrac{\Delta t}{\Delta x^2} = 1/10 \leq 1/2$. In other words, we require that the stability criterion is met. Using forward Euler, we get a mean squared error of $1.268 \cdot 10^{-7}$. The resulting approximation can be seen in figure 4

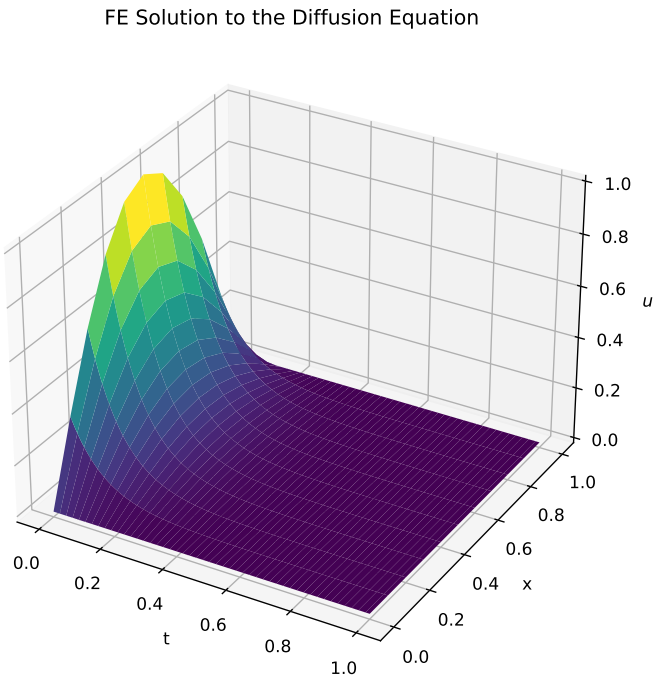

Figure 4: This figure shows the forward Euler approximation of the diffusion equation for $\Delta x = 0.1$ and $\Delta t = 0.001$.

.

We now examine how the forward Euler approximation behaves as a function of x, for times $t = 0.1$ and $t = 0.9$. Figure 5 shows us that the forward Euler solution is near-indistinguishable from the analytical solution for $t = 0.1$. We note that forward Euler appears to give slightly higher values than the analytical solution of u.

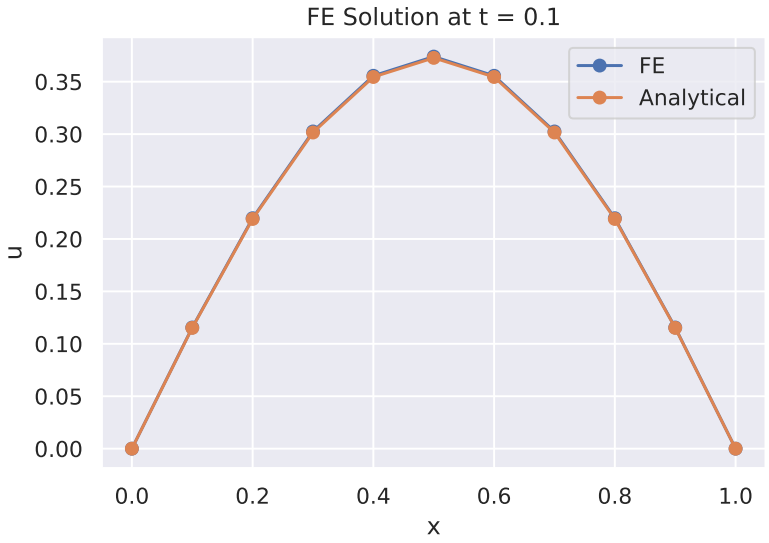

Figure 5: Here we see the forward Euler approximation of the diffusion equation at time $t = 0.1$, for discretization $\Delta x = 0.1$ and $\Delta t = 0.001$. It almost perfectly overlaps with the analytical solution.

.

On the other hand, we see a slight difference in forward Euler approximation, as compared to the analytical solution, when the time t increases and the value of u becomes smaller. Figure 6 shows precisely this. We interpret the fact that the error is larger as a consequence of the error propagating; the estimate of u at time $t = 0.9$ is based on the estimates of u at the previous time steps. However, since u is quite small, the absolute error remains low. We note that forward Euler still appears to give a higher value than the analytical solution of u.

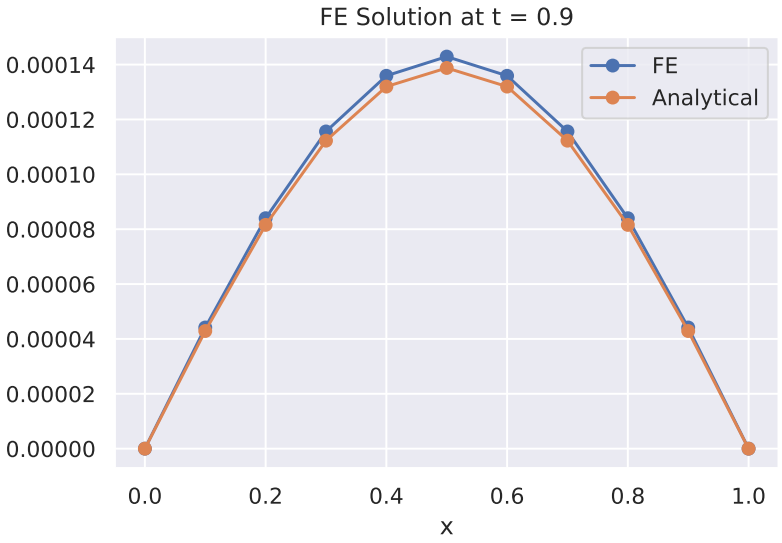

Figure 6: Here we see the forward Euler approximation of the diffusion equation at time $t = 0.9$, for discretization $\Delta x = 0.1$ and $\Delta t = 0.001$. Forward Euler predicts somewhat higher values than the analytical solution.

.

We perform another hyperparameter search for this discretization, over the values found in table 4, over 100 epochs and averaging over 5 trained models for each parameter configuration. This time we found that $\lambda_{opt} = 0$ and $\eta_{opt} = 10^{-4}$ were optimal. We then trained the model using these hyperparameters, and got an approximation with mean squared error $2.1356 \cdot 10^{-6}$. The resulting approximated solution can be seen in figure 7

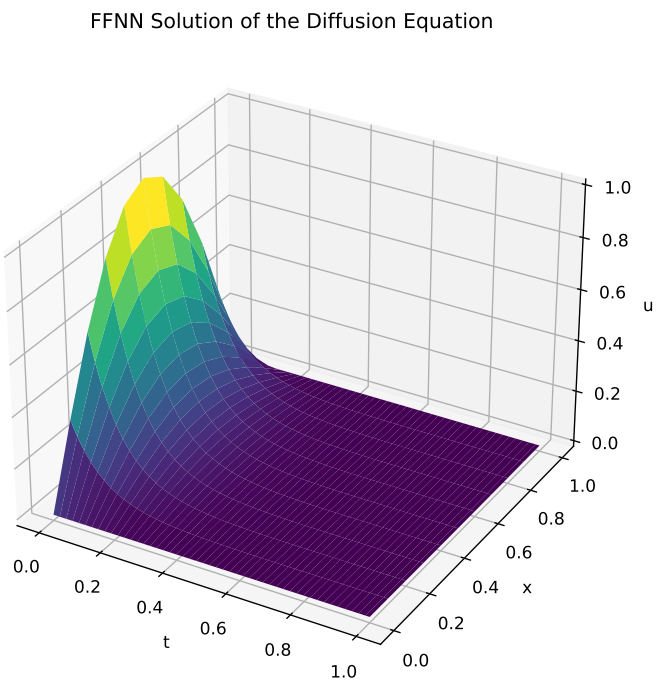

Figure 7: Here we see the feedforward neural network approximation of the diffusion equation for $\Delta x = 0.1$ and $\Delta t = 0.001$.

.

Furthermore, we plotted the solution for time $t = 0.1$, as can be seen in figure 8. Here, the FFNN appears to have learnt a quite good approximation, even between the discretization points.

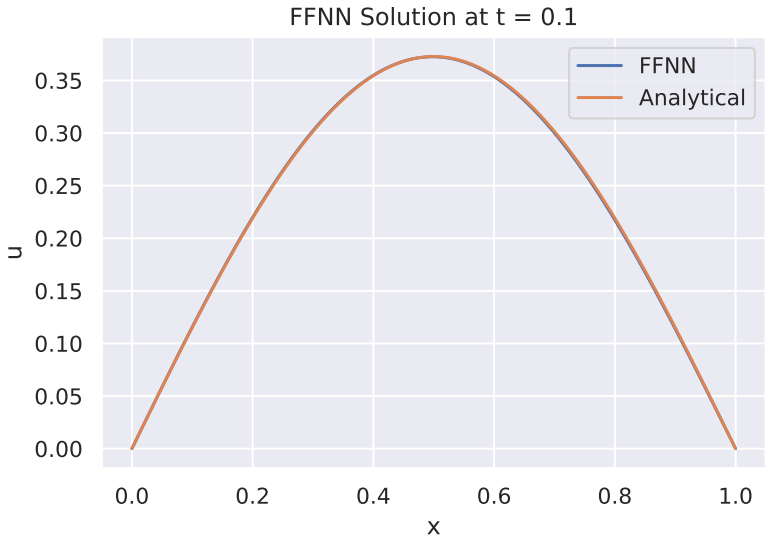

Figure 8: Here we see the feedforward neural network approximation of the diffusion equation for time $t = 0.1$, using discretization $\Delta x = 0.1$ and $\Delta t = 0.001$. It nearly perfectly aligns with the analytical solution.

.

For $t = 0.5$, we see that the neural network begins to predict a curve that is clearly different from that of the analytical solution. This can be seen in figure 9.

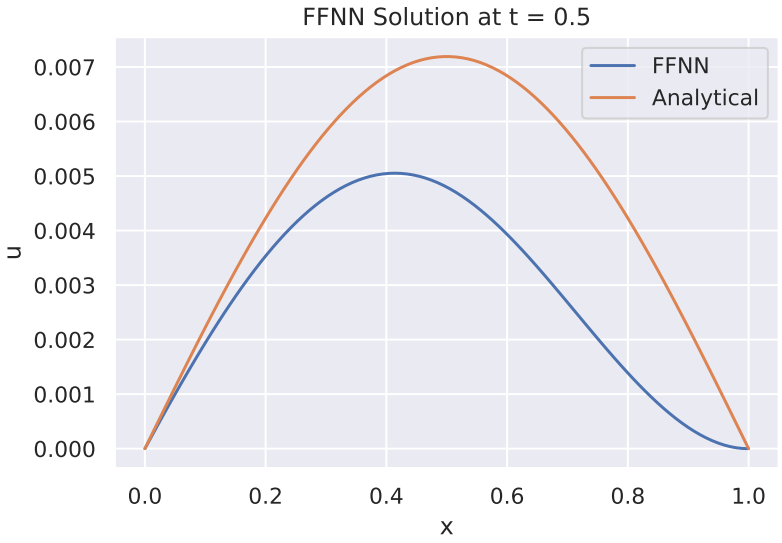

Figure 9: Here we see the feedforward neural network approximation of the diffusion equation for time $t = 0.5$, using discretization $\Delta x = 0.1$ and $\Delta t = 0.001$. It is starting to predict a different curve than the analytical solution.

.

And for $t = 0.9$, the FFNN predicts a curve that is in no way similar to that of the analytical solution. Given the small values of both the predicted and analytical solution, a higher relative error can occur without incurring too high of a loss. The predicted curve can be seen in figure 10.

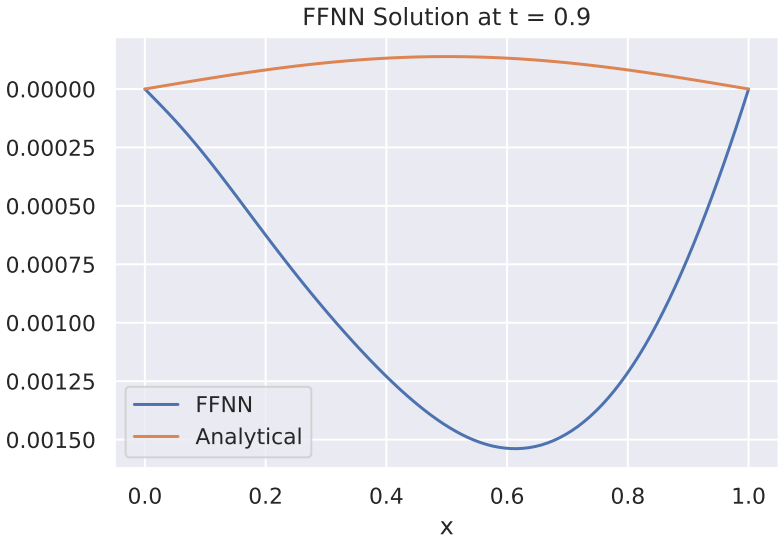

Figure 10: Here we see the feedforward neural network approximation of the diffusion equation for $t = 0.9$, using discretization $\Delta x = 0.1$ and $\Delta t = 0.001$. The neural network predicts an entirely different curve than the analytical solution curve.

.

Overall, we see that forward Euler outperforms the neural network for a discretization of $\Delta x = 0.1$ and $\Delta t = 0.001$. We do, however, note that the neural network can predict the value of $u(x, t)$ in between the discretization points, which can be a benefit. It does so pretty well for low values of t and close to the boundary points, where we have a model for how the system should behave. But for low values of u, far away from boundary points and the initial condition, the neural network can give predictions with a high relative error. Meanwhile, the forward Euler solution maintains a similar shape to that of the analytical solution.

3.1.3 The Third Discretization

As our third and final discretization, we choose $\Delta x = 0.01$ and $\Delta t = 0.001$ Again performing a forward Euler approximation, we get a mean squared error of $2.308 \cdot 10^{-11}$. As expected, this is even lower than the error of the previous discretization, since both Δx and Δt are smaller this time.

We then aim to train a neural network on the data. However, the data set contains on the order of 10^9 data points, making training a neural network on the entire data set quite computationally expensive. We therefore make the assumption that the hyperparameters found for the previous discretization are a good choice. We then train for 20 epochs. We then sample 10^5 random points from the data set to estimate the error. The model then gives a mean squared error of $1.394 \cdot 10^{-9}$. This error is remarkably low, albeit larger than that of the forward Euler discretization. Due to computational limitations, we were not able to train the network for more than 20 epochs, despite the loss not converging entirely. We therefore acknowledge the possibility that the network could

have performed better provided a proper parameter search had been done, or if it simply got more time to train. In any case we see that both the neural network and forward Euler can provide quite accurate estimates given a fine enough discretization (and enough training time in the case of the neural network).

3.2 Computing Eigenvalues

In this section, we will first explore how we can solve equation 21 to compute the eigenvalues of a real, symmetric matrix A. We first explore how we can do this with forward Euler. Next, we see how we can solve the differential equation, using the neural network described in section 2.5.4. Finally, we look at the approach for finding eigenvectors described in section 2.5.5.

3.2.1 Solving the Differential Equation with Forward Euler

We make the choice of discretizing with $\Delta t = 0.01$, and set the end of the interval we're considering to be $T = 3$. Then we generate a 6×6 matrix A. We generate a randomly selected unit vector x_0, and plot the evolution of $x(t)$ for the matrix A, as approximated by forward Euler. The components $x_i(t)$'s evolution over time can be seen in figure 11. Choosing the $x(t)$ estimate for the final time point and computing the corresponding eigenvalue, we compare with NumPy's estimate. The difference between the forward Euler and NumPy estimate is $6.2311 \cdot 10^{-6}$. We appear to have found a quite good approximation for the eigenvalue/eigenvector pair.

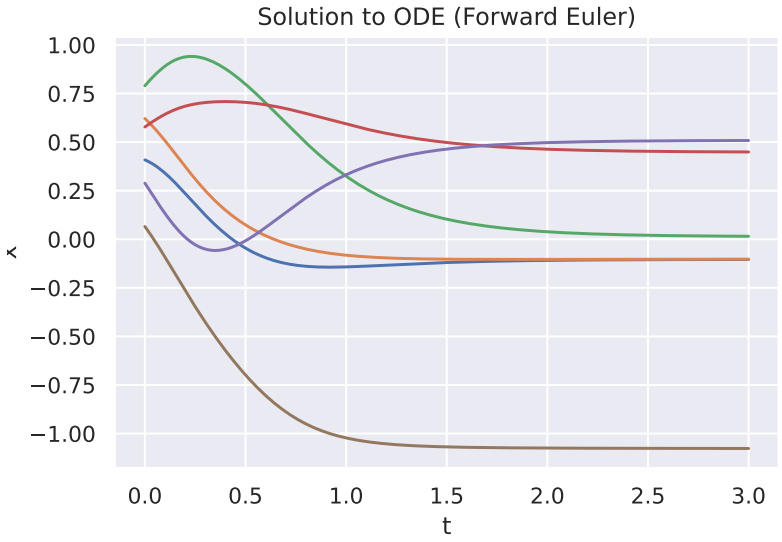

Figure 11: Each line denotes a component $x_i(t)$ of $x(t)$. This figure shows the evolution over time when solving for A. The components have clearly converged by $t = 3$.

Keeping these parameters constant, we generate 1000 random initial conditions, and see what they

converge to. We find that they converge for 886 of the initial conditions. Furthermore, they all converged to the eigenvector with the largest eigenvalue. Repeating this simulation for the matrix $-A$, we find that forward Euler converges to its largest value (corresponding to A's smallest eigenvalue) 971 times out of 1000. For the other initial conditions, it did not converge to any eigenvector. This result is in agreement with the findings of Yi et. al. that $x(t)$ will converge to the eigenvector with the highest eigenvalue (2004).

To Illustrate this phenomenon even further, we initialized our vector orthogonally to the eigenvector of A with the highest eigenvalue, and simulated with $\Delta t = 0.1$ until $T = 3$. Over 1000 simulations, $x(t)$ converged to an eigenvector 289 times. In every single case, the estimated eigenvalue was within a tolerance of 10^{-3} from the largest eigenvalue. Even in this edge case, numerical simulations of 21 seem to give eigenvectors with maximum eigenvalue. We suspect this may have to do with floating point errors.

3.2.2 Solving the Differential Equation with a Neural Network

We first test the $N = 4$ case. We use a FFNN with 2 layers with 64 nodes each. For 20 random initializations, we trained for 10000 epochs and checked whether it converged to an eigenvector. In 12 of the cases, it converged to the eigenvector with the largest eigenvalue (0.809). In 5 cases, it converged to the eigenvector with the second largest eigenvalue (0.302). However, it did not converge to any of the two eigenvectors with lowest eigenvalue. We then repeated the same process, computing eigenvalues for $-A$. Out of the 12 times the vector converged to an eigenvector, it always converged to an eigenvector corresponding to the same eigenvalue, namely -0.302 (which corresponds to the eigenvalue 0.302 for A). Notably, it did not converge to the largest eigenvalue this time. We see that the neural network solutions do not necessarily have to correspond to the true solution of the differential equation. The cost function can have local minima that correspond to the tail of the $x(t)$ being some eigenvector that that does not have maximal eigenvalue; this is still an equilibrium point.

We make use of this observation in an attempt to determine all the eigenvalues/eigenvectors of a 6×6 symmetric matrix. We again use a neural network with 2 layers of 64 nodes. We initalize x_0 orthogonally to the subspace spanned by already found eigenvectors. We then train the network to solve the differential equation over 100 time steps. If the found vector is an eigenvector and linearly independent of the already found vectors, we add it to our solution. Repeating this for 50 steps, we found a total of 4 eigenvectors. The computed eigenvalues had a maximum absolute error of $1.798 \cdot 10^{-5}$, making the estimates quite accurate. By continuing the process for iterations, it could have been possible to determine the last 2 eigenvalues. We were however unable to do so due to computational limitations.

3.2.3 Solving the Minimization Problem

Now we will consider the minimization approach discussed in section 2.5.5. We generated a real, symmetric matrix A, given by

$$A = \begin{bmatrix} 0.4996943 & 0.2042662 & 0.1863525 & 0.4636774 & 0.685108 & 0.1817188 \\ 0.2042662 & 0.4802477 & 0.4062406 & 0.2700818 & 0.556021 & 0.281753 \\ 0.1863525 & 0.4062406 & 0.1018862 & 0.496746 & 0.5053825 & 0.1579067 \\ 0.4636774 & 0.2700818 & 0.496746 & 0.0630985 & 0.5159196 & 0.5162116 \\ 0.685108 & 0.556021 & 0.5053825 & 0.5159196 & 0.9785835 & 0.34525 \\ 0.1817188 & 0.281753 & 0.1579067 & 0.5162116 & 0.34525 & 0.0403137 \end{bmatrix}. \tag{26}$$

We then compute its eigenvectors training the neural network, minimizing the loss. We perform this computation using a learning rate $\eta = 0.005$ over 20000 epochs. The network consists of 2 hidden layers with 20 nodes each. We use the ReLU function as hidden layer activation function, and we initialize the weights of the neural network by sampling from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1/100$. For analysis of the effect of weight initialization, see appendix B We reject all solutions with a loss greater than 10^{-5}, amounting to a total of 10 rejected trial eigenvectors. The final computed eigenvectors can be found in table 2.

v_1	v_2	v_3	v_4	v_5	v_6
0.0317484	0.2017753	0.2209159	0.3984849	0.5663826	0.6556576
-0.2912782	-0.1169072	-0.0418974	0.369079	0.5756471	-0.6573731
0.1299427	0.4397176	0.7005952	0.3249078	-0.3317797	-0.2885428
0.5575924	-0.722427	0.1330279	0.3753501	-0.0927238	0.002473
-0.4967156	-0.0550812	-0.3146664	0.6239837	-0.4785682	0.1812024
0.5827742	0.4767944	-0.5846687	0.2630248	0.0116578	-0.1478676

Table 2: Each column is one eigenvector, as computed by the neural network based algorithm.

We then compare these eigenvectors with eigenvectors computed using the NumPy library. The resulting eigenvectors can be found in table 3. We observe that the vectors are reasonably similar to the ones computed using the neural network. Quantitatively, the average angle between our eigenvector estimate and the corresponding NumPy estimate is $1.733 \cdot 10^{-3}$, and the maximum angle is $3.516 \cdot 10^{-3}$.

v_1	v_2	v_3	v_4	v_5	v_6
0.0320288	0.2019331	0.2188162	0.3989841	0.566669	0.655755
-0.2908101	-0.1162748	-0.0436805	0.3700923	0.5750581	-0.6575268
0.1301141	0.4399518	0.7015166	0.3245981	-0.3302243	-0.2880542
0.5574791	-0.7222949	0.1330175	0.375724	-0.0929437	0.0025222
-0.4971552	-0.0548242	-0.313272	0.6232021	-0.4800288	0.1813373
0.5826901	0.4768962	-0.5850239	0.2625455	0.0089502	-0.1475774

Table 3: Each column is one eigenvector, as computed using the NumPy package's function numpy.eig.

Next, we computed eigenvalue estimates using equation 13, with our neural network estimates for the eigenvectors. These eigenvalues, along with the ones we computed using NumPy can be found in table 4. We find that our algorithm has a mean absolute error of $2.073 \cdot 10^{-6}$ and a maximum absolute error of $6.116 \cdot 10^{-6}$, using the NumPy implementation as our ground truth. Similarly it has a mean relative error of $1.341 \cdot 10^{-5}$ and a maximum relative error of $6.523 \cdot 10^{-5}$.

Neural Network Estimates	0.1442520	-0.6272925	-0.1284611	2.4671149	-0.0551485	0.3633591
NumPy Estimates	0.1442521	-0.6272938	-0.1284624	2.4671210	-0.0551521	0.3633590

Table 4: Eigenvalue estimates using both the neural network approach and NumPy's function numpy.eig.

Overall we see that this approach more reliably produces eigenvectors than the approach discussed in section 2.5.4. Similar results could likely be found by simply applying gradient descent to the cost function

$$g(x) = (-x + f(x))^2 \tag{27}$$

given some initial guess x_0, without use of any neural network.

4 Conclusion

For the diffusion equation, we see that forward Euler tends to outperform the neural network given a fine enough discretization. Additionally, computing a forward Euler approximation is far less computationally expensive than training a neural network. We would like to point out, however, that a neural network based algorithm can have some benefits. In particular, it can allow for prediction of the function between the discretization points, which can be beneficial in some situations.

For the eigenvalue problem, forward Euler can accurately and consistently be used to discretize and solve the differential equation. In turn, we see that at a time $t = 3$, the equation has generally had the time to converge to an equilibrium. This method can reliably be used to determine the highest eigenvalue of A. We see that solving the differential equation using a neural network can find more eigenvectors, as local minima in the cost function can correspond to finding other eigenvectors. We were only able to retrieve 4 out of the 6 eigenvalues this way, however. Using the neural network minimization approach, we were able to more reliably find all 6 eigenvalues. The neural network based methods serve as interesting mathematical problems, but they are not suited for replacing existing eigenvalue solvers at this time. In practice, it is too time-consuming to train the model.

Overall, we see that the forward Euler scheme is an easy, computationally cheap, and reasonably good method for approximating differential equations. It is a reliable scheme that performs quite well given a fine enough discretization.

We also see that neural networks show promise in solving differential equations. In this report we see that they can generate approximations that are quite good. The neural network-based algorithms generally did not outperform forward Euler in the tests we ran, however. We observe some

primary weaknesses and strengths with using a neural network based model. A strength of neural network based approaches is that they allow for using domain knowledge in modelling tasks. In constructing trial functions, one can require that the solution obeys initial conditions, boundary conditions, and so on. One major drawback is the computational load required to train a good model. Comparatively speaking, forward Euler is far more time efficient for solving tasks like these.

We suggest two main areas for future work regarding solving differential equations with neural networks. Neural network solvers are especially powerful when one can combine them with domain knowledge. We therefore suggest testing and comparing the performance of different trial functions that use different assumptions and observations about the solution of the diffusion equation. For example, one could impose a condition that the solution is positive for the diffusion equation. We also suggest looking into the effect of varying neural network structure, activation functions, and other hyperparameters of the neural network. While we were not able to consistently train networks that outperformed forward Euler, this may be possible with a different network design.

References

1 Harris, C.R., Millman, K.J., van der Walt, S.J. et al (2020). Array programming with NumPy. Nature 585, 357–362.

2 Karlik, B., & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. International Journal of Artificial Intelligence and Expert Systems, 1(4), 111-122.

3 Yi, Z., Fu, Y. & Tang, H. J. (2004). "Neural Networks Based Approach for Computing Eigenvectors and Eigenvalues of Symmetric Matrix". `https://www.sciencedirect.com/science/article/pii/S0898122104901101` Accessed: 18.12.2022.

A Code

The code used to compute estimates and generate plots can be found at:
`https://github.com/Rednael18/FYS-STK3155-project-3`

B Weight Initialization

In this appendix, we will briefly demonstrate the importance of good weight selection for the eigenvector minization problem. Assuming that we are limited to training over 1000 epochs, we train using a learning rate $\eta = 0.01$ and scaling the standard deviation of the weights of the neural network by powers of 2. We want to check for convergence towards a specific eigenvector, so we initialize our vector $x_0 = [0.5, 0.5, 0.5, 0.5, 0.5, 0.5]^T$ and compute the difference between the predicted eigenvalue after training, and the eigenvalue of the closest eigenvector to x_0. The resulting graph can be seen in figure 12. We observe that we get poor convergence for little scaling. This is due to the neural network function having a large initial value. In turn, this means that the initial prediction made does not actually have to be close to x_0, so that one can get convergence to

other eigenvectors (with other corresponding eigenvalues). We see that a weight initialization with $\sigma \in [1/16, 1/8]$ appears to be a good range. However, lower values also appear to be fine.

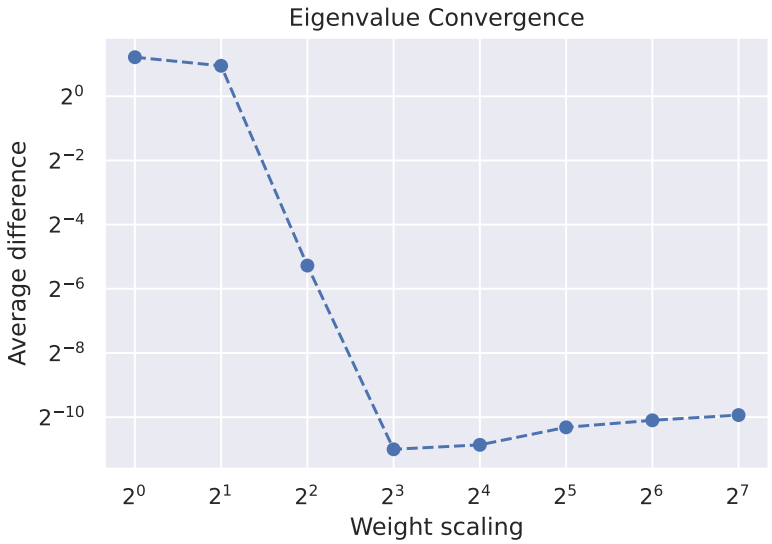

Figure 12: This figure shows the convergence towards the eigenvalue of the closest eigenvector, as a function of the scaling factor $w_{scaling}$. The standard deviation of the initialized weights of the neural network are given by $\sigma = 1/w_{scaling}$. We averaged the absolute difference in eigenvalue over 100 simulations.

Analyzing the bias-variance tradeoff for three different algorithms

Halvor Tyseng, Benedict Leander Skålevik Parton, Carl Fredrik Nordbø Knudsen

December 2022

1 Introduction and method

When using different models to approximate a function, there is a certain tradeoff between the bias of the model - that is, the extent to which the average estimated prediction is skewed compared to the true value - and the variance of the model; how sensitive the model is to changes in the training dataset, indicative of overfitting. When decreasing one, the other must increase.

We have analyzed the bias-variance-tradeoff for three different models; a simple decision tree regressor, a feed forward neural network, and polynomial fit. We used Scikit-learn's implementation of decision tree regressors, as well as FFNN, whilst the linear regression code was our own.

For each of the models found the estimated variance, bias and error as a function of model complexity. The model complexity refers to the number of features of parameters in a model, which is represented in different ways for different models. In our analysis we have chosen to analyse over maximum polynomial degree for the polynomial fit, depth of the decision tree for the decision tree model, and number of nodes in the hidden layer for the feed forward neural network.

We chose to use data generated from the Franke function on our models, where in all cases we generated 500 points in $x, y \in [0, 1]$, with an added noise $\epsilon_i \overset{\text{iid}}{\sim} \mathrm{N}(0, 0.01)$. The dataset was then split into training and testing data,

We used independent bootstrap to estimate the variance and bias that the predictions of each model inherit. The bootstrap was done in all cases with 100 n-samples, and the variance, bias and error were calculated in line with chapter 5 in the jupyter-book "Applied Data Analysis and Machine Learning" by Morten Hjorth-Jensen (2021).

2 Results and discussion

The source code used in this additional exercise is found in the folder "additional exercise" in our GitHub repository, `https://github.com/Rednael18/FYS-STK3155-project-3/tree/main/Additional%20Exercise`.

We first found and plotted the bias, variance and error (MSE) of OLS on the Franke function dataset. Using polynomial order as the model complexity, we got the plot as seen in figure 1. We clearly see that the bias starts out high, and falls with increasing model complexity, while the variance instead increases with model complexity. This corresponds with what we would expect, as

a model with low complexity would not be able to sufficiently adapt to the dataset to consistently hit the target value, therefore causing a significant bias to its predictions. The variance, meanwhile, stays low, as the model is very unsensitive to changes in the training dataset, since it already adapts to the data quite poorly.

On the other hand, with high model complexity, the model easily manages to approximate the training dataset, and tends towards overfitting - this means that it has no certain bias in its predictions, but that these predictions can be all over the place depending on what training dataset was used (since this would cause it to adapt very differently), thus causing a high variance.

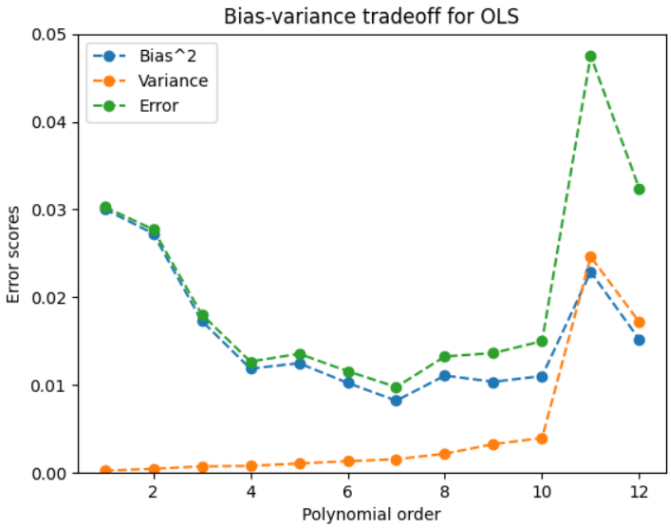

Figure 1: Bias, in blue, appears to taper off with increasing complexity (here by polynomial order), whilst the variance, in orange, increases.

For the feed forward neural network we used a model with three layers; one input layer, one hidden layer, and one output layer. The model complexity in this instance was the number of nodes in the hidden layer; however, one might well also choose number of hidden layers, or a combination of these two, to be indicative of model complexity. This might be something to investigate further, to build on our work. From this we got the plot as seen in figure 2.

In this case, we also observe that the bias decreases and the variance increases as model complexity increases. However, the change doesn't appear to be very dramatic compared to what we saw with OLS. This may be due to our choice of complexity being how many nodes there are in the (single) hidden layer. This might be restrictive, compared to the complexities a neural network might achieve with multiple hidden layers, and so the change in bias and variance is rather subdued.

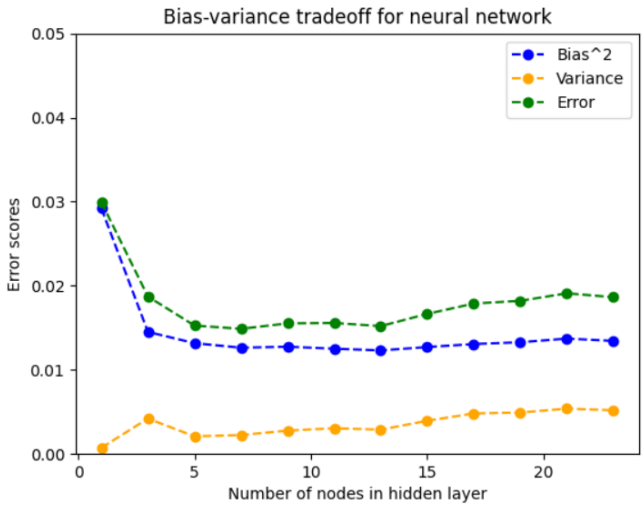

Figure 2: Bias, in blue, appears to taper off with increasing complexity (here by number of nodes in the hidden layer), whilst the variance, in orange, increases somewhat.

An interesting path of further investigation might thus be to see the change in bias and variance with more hidden layers, and nodes within said layers.

Lastly we preformed the independent bootstrap with a decision tree regressor of different depth. We let the depth of the tree represent the model complexity. Figure 3 shows a plot of the estimated bias, variance and error for a decision tree regressor, with depth up to fourteen.

We see the same underlying trend here, as we did for the two other models as well; bias decreases and variance increases as we increase model complexity. One could argue that the results in the plot shows signs of a variance that stabilizes.

To investigate this further we also did the same analysis with an increased depth range; this is seen in figure 4.

Figure 4 supports our theory as we see a near stabilization of all the measures. Once the structure and art of decision trees is understood, one can easily see how with enough depth, any training data (up to a size constrained by the computing power) could be perfectly overfitted, aligning a branch to each data point in the training set. This is also true for polynomial fit to the part where we could, with high enough polynomial order, perfectly fit a polynomial to the training data. However, the outcomes of these two overfittings are completely different. The overfitting of a polynomial could result in a function whose values can be arbitrarily big for a data point not included in the train data. This is not the case for decision trees, as it will only return values close to what it is trained on - the output being decided solely by the individual branches of the tree. Further investigations are surely warranted into how the variance of a decision tree model behaves.

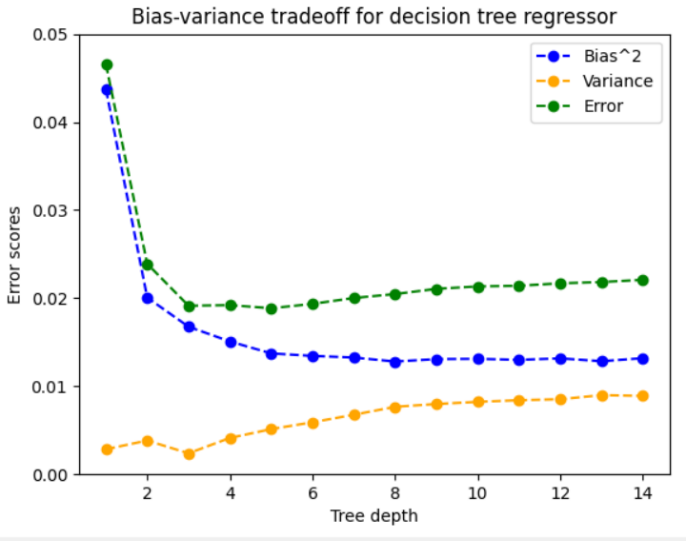

Figure 3: Bias, in blue, appears to taper off with increasing complexity (here by the depth of the decision tree), whilst the variance, in orange, increases.

The links between the behaviour of the decision tree regressor and that of the feed forward neural network, which seems quite matching, may also be of interest to look into.

This analysis is done solely on one regression problem, with a fixed number of data points. We are not quite sure how this generalizes to other cases but our findings suggest that due to the fundamental structure of the different models, the overfitting problem, and thus the bias-variance tradeoff unfolds quite differently for the three models under investigation. The polynomial fit seems to be much more prone to high variance with high model complexity than a decision tree, at least in fitting problem. We suggest that further analysis could be done on other data sets in order to check if our findings generalise.

3 Conclusion

We have performed an analysis on the bias-variance tradeoff for three different models. With OLS, we found that the bias and variance behaved predictably, respectively decreasing and increasing for higher complexities. With both decision tree regressors and neural networks, we found the same tendencies, though somewhat subdued, and with decision tree regressors we found that they plateaued instead of continually falling/rising. This hints to some of the major differences between the models from a bias-variance perspective; that increasing the complexities for decision trees and seemingly neural networks after a certain point does not significantly impact this tradeoff, while it makes a huge difference for OLS. From this we conclude that the bias-variance tradeoff is an important factor to consider no matter the model one chooses, though it may behave quite

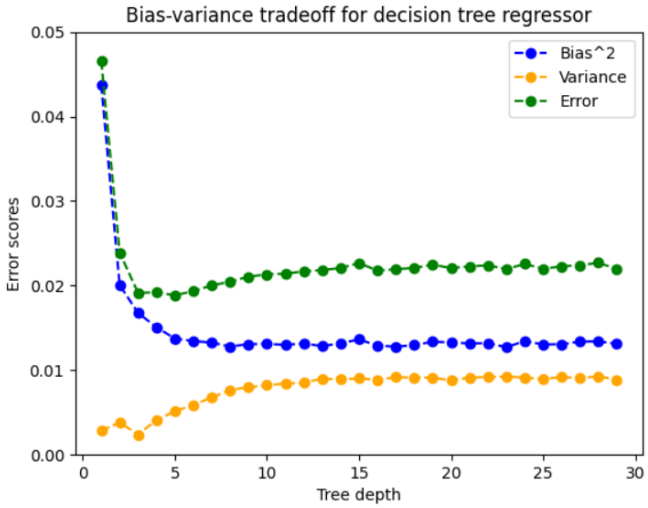

Figure 4: Bias, variance and error all appear to stabilize beyond a certain model complexity.

differently with different models.

References

1 Hjort-Jensen. M, (2021). Applied Data Analysis and Machine Learning. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html#applied-data-analysis-and-ma` Accessed: 18.12.2022.