Poor performing machine learning methods demonstrating the power of machine learning

Gaute Holen, Paul Giraud & Fridtjof Ronge Gjengset
(Dated: December 16, 2021)

In this paper we explore solving partial differential equations using machine learning methods, and comparing it to the finite different scheme. Our differential equation is based on the application of the diffusion equation for a one-dimensional rod. We wish to explore how machine learning methods such as a neural network and linear regression models performs in comparison to the more traditional explicit scheme, with a look at the bias variance trade-off for the linear regression models OLS, Ridge and Lasso. In addition we will use a neural network to solve eigenvalue problems. Throughout this paper you will see properly working machine learning methods be outperformed by it's traditional numerical counterpart. This common thread demonstrates that we need to apply machine learning methods where we can take advantage of it's strengths, and where we have no reliable numerical methods, instead of pitting it against established numerical methods.

I. Introduction

We wish to explore the boundaries of using machine learning methods in physics problems. The best way to do this, we think, is to pit machine learning methods against other more traditional numerical methods. Differential equations such as the diffusion equation are a prominent part of physics, although we have some numerical approaches to them such as the explicit scheme, they often come at some numerical precision expense, like a stability requirement. Another example could be the Crank-Nicholson approach which has no stability requirements, but comes with a higher computational cost and error. This gives some motivation to make machine learning methods work.

In this paper we will first explore the analytical and numerical explicit scheme solution to the diffusion equation in a one-dimensional problem. We do this to establish a baseline for discussing the implementation of machine learning methods. We will then move onto discussing the implementation of neural network and linear regression methods to solve the diffusion equation. The bias, variance and MSE for the linear regression models are more closely studied to further assess the accuracy of the regression models.

We will also discuss the use of neural networks to solve eigenvalue problems, as discrete differential equations often can be written as an eigenvalue problem. Although we will not rewrite the diffusion equation as an eigenvalue problem in this case, we still wish to demonstrate the possible application of using machine learning to solve such a problem.

Secondly, we will present our results and compare the performance of the neural network, ridge and explicit scheme solution to each other, and use the analytical solution as the baseline for our discussion. We will highlight some of the challenges of implementing the machine learning methods, and share what we learned about restrictions we experience during this implementation. We will also demonstrate how to solve eigenvalue problems using a neural network, and discuss why this is useful.

Finally we will have a critical discussion about the different methods, and share some of our thoughts on the usefulness of machine learning methods when solving differential equations.

Note that throughout this paper we will refer to the learning rate as λ as opposed to the convention of using η as the learning rate and λ as a regularization parameter. We have also used λ or rather *lmd* as the learning rate in our code.

II. Method

A. analytical expression and solution

The partial differential equation we chose to solve for this paper is the diffusion equation on a one dimensional rod. The rod has a length $L = 1$ spanning from $x = 0$ to $x = 1$. The diffusion equation in one dimension can then be written as:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t},\ t > 0,\ x \in [0,1]. \tag{1}$$

Where $u(x,t)$ is the temperature gradient of the rod at a position x and time t.
We also need to impose some boundary conditions on the problem. We want heat to escape thought the endpoints of the rod, so we have the following boundary conditions:

$$u(0,t) = 0,\ t \geq 0, \tag{2}$$
$$u(L,t) = 0,\ t \geq 0. \tag{3}$$

In other words the temperature gradient u is zero at the two endpoints $x = 0$ and $x = L$ throughout the simulation.
We also need an initial state of the system at $t = 0$, which

fulfills the requirements for the two boundary conditions 2 and 3. Using the sine function allows us to do exactly this, and we can set the initial temperature gradient to:

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

As the length of the rod $L = 1$, the temperature gradient u will be zero at the two endpoints ,fulfilling the two boundary conditions, and have a maximum value at the centre of the rod of $u(x = 0.5, t = 0) = 1$.

For a rod of length $L = 1$, and with initial conditions as in Equation 2, 3 and 4, we can calculate the analytical solution:

$$u(x,t) = Qe^{-at} \sin(kx) \tag{5}$$

With initial shape of u along x

$$I(x) = Q \sin kx \tag{6}$$

Where

$$a = -\alpha k^2 \tag{7}$$

With $Q = 1$ and $k = \pi$ we get the exact solution

$$u(x,t) = e^{-\pi^2 t} \sin(\pi x) \tag{8}$$

We can plot the exact solution using equation 8, to get a visual representation shown in figure 1.

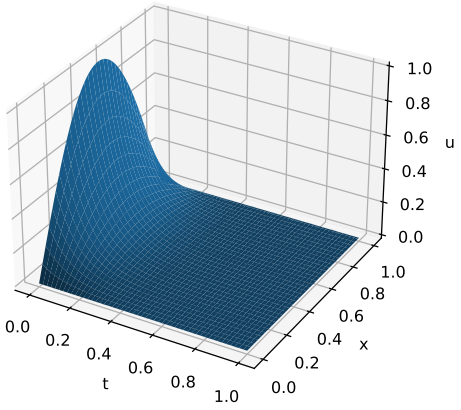


FIG. 1: Shows the exact solution for $u(x,t)$ using equation 8.

B. Numerical expression

For the numerical solution we simply use the the *explicit forward Euler algorithm* by discretizing time using the *forward* formula, and the *centre difference* for space. We can then write out the truncated approximations to be:

$$\begin{aligned} u_t &\approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \\ &= \frac{u(x_i,t_{j+1}) - u(x_i,t_j)}{\Delta t} \end{aligned} \tag{9}$$

for the time dimension and

$$\begin{aligned} u_{xx} &\approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} \\ &= \frac{u(x_{i+1},t_j) - 2u(x_i,t_j) + u(x_{i-1},t_j)}{\Delta x^2} \end{aligned} \tag{10}$$

for the space dimension. Note that Δx and Δt are the step-length in x and t. The subscripts i and j is the indices for x and t respectively.

We can now write the the diffusion equation 1 on a discretized form using equation 9 and 10 as:

$$u_{xx} = u_t. \tag{11}$$

It is however worth noting that the *explicit scheme* has a stability criterion, requiring that $\Delta t/\Delta x^2 \leq 1/2$. If this criterion is not met, the simulation will ultimately fail to approximate the analytical solution. We can rewrite this requirement as:

$$\begin{aligned} \frac{\Delta t}{\Delta x^2} &\leq \frac{1}{2} \\ \Delta t &\leq \frac{\Delta x^2}{2} \end{aligned} \tag{12}$$

C. Implementation of neural network

We mainly know neural networks as large hierarchical models that can learn patterns from data of complicated nature or distribution. This is why we see many successful applications for image, sound, video, and sequential action processing. But we usually don't remember that NNs are also universal function approximators, therefore, they can be applied to more "classical" mathematical problems as a tool for numerical analysis. In this section we apply simple NNs to solve the partial differential equation of diffusion 1.

To solve this equation we will define the trial function that we will need to define after our cost function. The main question is how to transform the equation integration problem into an optimization one, for example by minimizing the error of the numerical solution, taking into account the initial and boundary conditions.

$$g_t(x) = h_1(x) + h_2(x, N(x, P)) \tag{13}$$

According the initial and boundary conditions from 2 3:

$$g_t(x,t) = h_1(x,t) + x(L-x)tN(x,t,P) \tag{14}$$

with:

$$h_1(x,t) = (1-t)\sin(\pi x) \tag{15}$$

where N (x, P) is a neural network of arbitrary architecture, whose weights must be learned to approximate the solution. **P** being the collection of the weights and biases for each layer.

so the problem is transformed into this one:

$$\min_P \Big\{ \big(\frac{\partial^2 g_t(x,t,P)}{\partial x^2} - \frac{\partial g_t(x,t,P)}{\partial t} \big)^2 \Big\} \tag{16}$$

that leads to:

$$\min_P \Big\{ \frac{1}{N} \sum_{x_i,t_i \in D} \big(\frac{\partial^2 g_t(x_i,t_i,P)}{\partial x_i^2} - \frac{\partial g_t(x_i,t_i,P)}{\partial t_i} \big)^2 \Big\} \tag{17}$$

$$C(x,t,P) = \frac{1}{N} \sum_{x_i,t_i \in D} \big(\frac{\partial^2 g_t(x_i,t_i,P)}{\partial x_i^2} - \frac{\partial g_t(x_i,t_i,P)}{\partial t_i} \big)^2 \tag{18}$$

We want to minimize C(x, t, P) that can be written as C(x, t, $\{P_{hidden}, P_{output}\}$) (there are no weights and bias at the input layer).

If there are N_{hidden} neurons in the hidden layer, then P_{hidden} is a $N_{\text{hidden}} \times (1 + N_{\text{input}})$ matrix, given that there are N_{input} neurons in the input layer, and P_{output} is a $N_{\text{output}} \times (1 + N_{\text{hidden}})$ matrix. (We add 1 to N_{input} and $N_{\text{hidden}}$to include the bias). The initialization is done by using *np.random.randn()*.

To train our neural network we will pass our inputs (in this case the position and time vectors x and t) into the network, i.e. if we note x the input of a layer we will have in output :

$$\boldsymbol{y}_i^{\text{hidden}} = f\big(\boldsymbol{b}_i^{\text{hidden}} + \boldsymbol{w}_i^{\text{hidden}} \boldsymbol{x}_1, ..., \boldsymbol{b}_i^{\text{hidden}} + \boldsymbol{w}_i^{\text{hidden}} \boldsymbol{x}_n\big)$$

which is just :

$$\boldsymbol{y}_i^{\text{hidden}} = f\big(\boldsymbol{P}_{i,\text{hidden}}^T X\big) \tag{19}$$

with an activation function f used for the neurons of the layer. This equation give us the output of the *Neural Network* by using $X = (\text{x,t})^T$ for the input and for each layer we use as input the output of the previous layer.

Now the *Neural Network* has to change to parameter $\boldsymbol{P}$ in order to minimize the cost function C(x, t, P). We introduce the **BackPropagation** function by using the gradient descent with a step size/learning rate λ. The value of λ decides how large steps the algorithm must take. We have to minimize the cost function $C(\text{x,t}, P)$ with respect to the two sets of weights and biases, P_{hidden} and P_{output} :

$$P_{\text{hidden,new}} = P_{\text{hidden}} - \lambda \nabla_{P_{\text{hidden}}} C(x,t,P) \tag{20}$$

$$P_{\text{output,new}} = P_{\text{output}} - \lambda \nabla_{P_{\text{output}}} C(x,t,P) \tag{21}$$

Now we have all the functions defined to implement our neural network.

To improve the convergence of the cost function per epoch, we introduced an adaptive learning rate λ. If the cost function at a certain epoch is larger then in the previous epoch, λ is reduced by 30%. This allows us to start with a relatively large initial λ value, so the cost function is reduced faster. This proved quite useful, as it also allowed us to run for a large number of epochs. We considered adding an additional criterion where λ is reduced if the cost function didn't converge fast enough. However, this proved tricky as it often resulted in a stalemate where the reduced λ hindered the cost function gradient to be large enough to fulfill the criterion. The λ was then further reduced ending in a cycle where the cost function basically remained unchanged.

D. Linear Regression

In previous projects, linear regression was used to approximate the Franke Function, which was a surface $f_{Franke}(x,y)$. Similarly, the diffusion equation $u(x,t)$ is a surface which can be approximated with the same method. As done previously, a design matrix X of some polynomial degree is set up with corresponding values z, where z is the analytical solutions to the diffusion equation. Further, X and z are split into train and test sets, before the mean square error is calculated between $z_{predict}$ and $z_{analytical}$.

1. *Ordinary least squares*

The first linear regression model used here is OLS, where we have the relationship

$$\text{X}\boldsymbol{\beta} = \mathbf{z} \tag{22}$$

Where $\boldsymbol{\beta}$ can be found with the training set $\mathbf{X}$

$$\hat{\boldsymbol{\beta}} = \left(\mathrm{X}^\top \mathrm{X}\right)^{-1} \mathrm{X}^\top \mathbf{z} \tag{23}$$

Finally, the predicted values are given as

$$\mathrm{X}_{test}\boldsymbol{\beta} = \mathbf{z}_{predict} \tag{24}$$

The following algorithm is used for OLS

```
def OLS_solver(X_train, X_test, z_train, z_test):

 # Calculating Beta Ordinary Least Square Equation with matrix pseudoinverse

 ols_beta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ z_train

 z_tilde = X_train @ ols_beta # z_prediction of the train data
 z_predict = X_test @ ols_beta # z_prediction of the test data

 return ols_beta, z_tilde, z_predict
```

2. *Ridge*

Ridge regression is very similar to OLS, but introduces the parameter λ such that

$$\widehat{\beta}_{\text{ridge}} = \left(X^T X + \lambda I_p\right)^{-1} X^T y \tag{25}$$

where I is the $p \times p$ identity matrix. Note that when $\lambda = 0$ we get the same expression as for OLS[1]. The λ parameter adds an L2 penalty, which is a secondary term which reduces the variance of $z_{predict}$, which tends to improve MSE. Here is a the code for the implementation

```
def ridge_reg(X_train, X_test, z_train, z_test, lmd):

    ridge_beta = np.linalg.pinv(X_train.T @ X_train + lmd*np.eye(len(X_train.T))) @ X_train.T @ z_train #psudoinverse
    z_model = X_train @ ridge_beta #calculates model
    z_predict = X_test @ ridge_beta

    return ridge_beta, z_model, z_predict
```

3. *Lasso*

The last linear model is Lasso, which

```
def lasso_reg(X_train, X_test, z_train, z_test, lmd):

    RegLasso = linear_model.Lasso(lmd)
    _ = RegLasso.fit(X_train,z_train)
    z_model = RegLasso.predict(X_train)
    z_predict = RegLasso.predict(X_test)

    return z_model, z_predict
```

[1] This λ is not the learning rate.

E. Eigenvalue with Recurrent Neural Network

In many engineering problems, it is useful and important to diagonalize a matrix, to inverse a matrix or simply to have its eigenvalues or eigenvectors. Thus, the search for methods to compute these eigenvalues and eigenvectors is very important nowadays. We are therefore interested in a new method using *Recurrente Neural Networks* to calculate these values on real symmetric square matrices. We know from the spectral theorem that these matrices are invertible so we will be able to look for these values.

For our study we will rely on the work [1] by Yi et al. We will look for the eigenvalues of a matrix A of size $n \times n$. For that let us consider the dynamic model of our neural network by the equation:

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)) \tag{26}$$

for $t \geq 0$, where:

$$f(x) = [x^T x A + (1 - x^T A x)I]x \tag{27}$$

where $x \in \mathbb{R}^n$ represents the state of the network, I is the n × n identity matrix. This equation is non-linear, so we will solve it as said before by a Recurrent Neural Networks. In the following, we will rely directly on the theorems presented in [1] without proving them, all the proofs are shown in the article.

Theorem 1: Given any non zero vector $x(0) \in \mathbb{R}^n$,if $x(0)$ is not orthogonal to the set of eigenvectors of A,then the solution of 26 starting from $x(0)$ converges to an eigenvector corresponding to the largest eigenvalue of A.

Theorem 2: Given any non zero vector $x(0) \in \mathbb{R}^n$,if $x(0)$ is not orthogonal to the set of eigenvectors of A, and if we replace the matrice A by -A in the *Recurrent Neural Network*, then the solution of 26 starting from $x(0)$ will converge to an eigenvector corresponding to the smallest eigenvalue of A.

Now we will look how to defined our loss function the *Recurrent Neural Network*. Cost function is defined as:

$$C(x, P) = h(x, \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2}, ..., \frac{d^n x(t)}{dt^n})^2 \tag{28}$$

Then here,

$$C(x, P) = h(x, \frac{dx(t)}{dt})^2 \tag{29}$$

from 26:

$$C(x,P) = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - x_i)^2 \tag{30}$$

$$C(x,P) = \frac{1}{N}\sum_{i=1}^{N}([x_i^T x_i A + (1 - x_i^T A x_i)I]x_i - x_i)^2 \tag{31}$$

Now we can use this cost function to run the *Recurrent Neural Network* with **TensorFlow**. For this we need to initialize our matrix A that will be generated randomly but we want A to be symetric so we generate randomly a matrix Q and use the fact that:

$$A = \frac{(Q + Q^T)}{2} \tag{32}$$

is symetric for any real matrix Q.

Next we initialize our starting solution be a random vector created by *np.random.random.sample* so each component of $x(0)$ is between 0 and 1. Then the *Recurrent Neural Network* is trained as for the **Implementation of neural network** with different hyperparameters:

- number of iteration.
- number of layers.
- number of neurons per layer.
- the learning rate.
- the activation functions.

F. Cross validation

In the case of this project, the data set used for the explicit scheme, with $dx \in 0.1, 0.01$, the dataset is not too big, which makes the regression prone to over-fitting and bad MSE estimates. To avoid this, cross-validation used to get a more realistic MSE value from the same dataset by effectively increasing the amount of test and train splits obtained from the same dataset. This is done by splitting the dataset into groups, where some groups are assigned to the "test" data and some to the "train" data. Each unique combination of groups in test and train sets is called a fold. Every fold has a unique set groups in the test and train sets, effectively meaning all the data is used as test and train data. Then, the mean of all the MSE from each fold's prediction versus test data is taken as the final MSE. In the results section you may see how this affects the MSE for different models. For more details of the specific implementation, please see project 1.[2]

[2] https://github.com/fridtjrg/FYS-STK4155/tree/main/projects/reports/project1

G. Bias variance tradeoff

In regression and machine learning, the increasing the bias can be used to to reduce the variance of the model, leading to better results. There's always a tradeoff, where the goal is to find the sweet spot where both the variance and bias is low, leading to a lower all around error.

The expected error of a learning algorithm can be analyzed as the combination of bias and variance such that

$$MSE = \mathbb{E}\left[(y - \tilde{y})^2\right] = \text{ bias }^2 + \text{ variance } + \sigma^2 \tag{33}$$

This was covered in detail in the report project 1.

To improve the accuracy of our bias variance analysis, cross-validation was used to calculate the bias, variance and MSE for each of the k folds, then a mean of each parameter for the fold was given as the final value. This is performed for a range of complexities, which can be seen in the results section.

H. MSE and difference

As we will see, the neural network does fit the data-set very well. We therefore decided to look at the absolute difference between the analytical solution and the NN prediction, instead of comparing their respective MSE scores. This was an attempt to gain a better insight into why the network performed poorly, as we can clearly see what would give the better MSE based on the absolute difference plots. We will present either the MSE score *or* the absolute difference depending on the nature of the problem.

III. Results and discussion

A. Using explicit scheme

As the explicit scheme method is the numerical method we are choosing to pit out machine learning algorithm against, we must first look at the performance using the *explicit scheme*, shown in figure 2

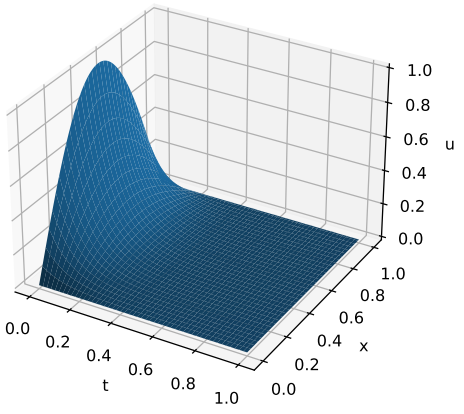


FIG. 2: Shows u as a function of x and t calculated using the explicit scheme.

Looking at figure 2, it is more or less indistinguishable from the analytical solution shown in figure 1. We therefore plot the absolute difference between the two in figure 3

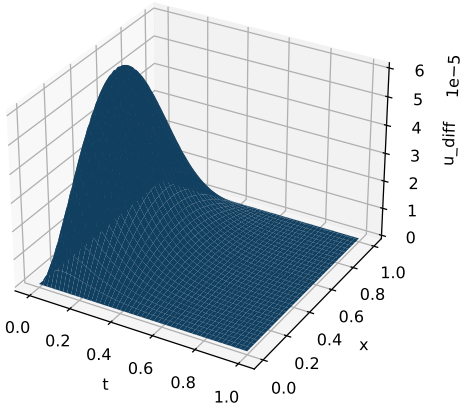


FIG. 3: Shows the absolute difference between the analytical result shown in figure 1, and the explicit scheme solution shown in figure 2.

By looking at the absolute difference between the analytical and numerical solution, shown in figure 3, we see that the difference is very similar to the solution itself. The main difference being that the absolute difference does not diminish as quickly with increasing t, as it does in the solution. This suggesting that the absolute difference is larger when u has a large gradient. We see the the peak absolute difference is $\sim 6 \cdot 10^{-5}$

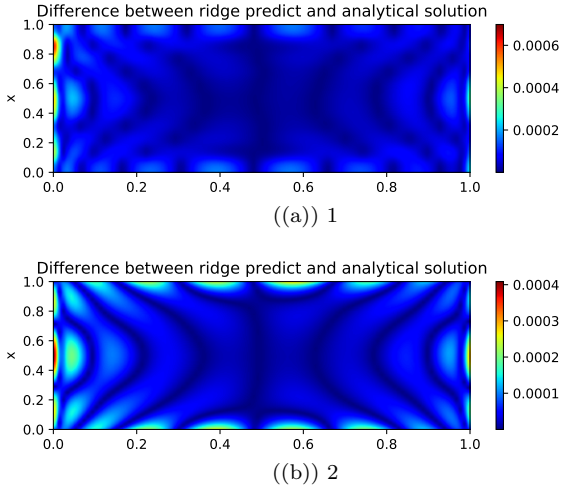
B. Ridge regression and optimal parameters

We will now look at the linear regression predictions on the diffusion equation, with the optimal parameters. The data is shuffled before the train/test split, leading to slightly different results for optimal parameters.

We study two cases: first having $\Delta x = 0.1$ and $\Delta t = 0.01$ and the second having $\Delta x = 0.01$ and $\Delta t = 0.01$.

Finding the optimal value for λ is done by investigating a range of lambdas in the range $[10^{-15}, 10]$. For both cases of Δx, depending on the specific train test split, the best λ values found were 10^{-3}, 10^{-10} and 10^{-12}. Moving on, 10^{-3} was picked as the favored value for λ.

In figure 4, the difference between the predicted $u(x,t)$ and the analytical solution is plotted. Note that the difference in resolution in x and t is very small here, much less than the order of magnitude difference in Δx.



((a)) 1

((b)) 2

FIG. 4: Shows the absolute difference between the analytical solution and the predictions from Ridge regression with $\lambda = 10^{-3}$ and $\Delta x = 0.1$ (a) and $\Delta x = 0.01$ (b) with a polynomial degree 10.

C. Cross validation to asses linear regression models

Cross validation is used to calculate the MSE for OLS, Lasso and Ridge, with $\lambda = 10^{-3}$ as before. In Figure 5 amd 6, the MSE is plotted with and without cross-validation with $k = 10$ number of equally sized folds. Compared to 4, the cross-validation MSE is in the same order of magnitude as the difference for Ridge. Note also that the difference in Δx here also does not seem to largely impact the MSE.

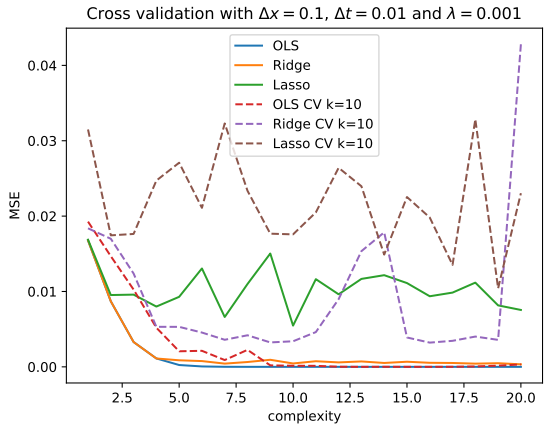


FIG. 5: The cross-validation MSE and MSE for OLS, Ridge and Lasso regression for $u(x,t)$ versus complexity with $\Delta x = 0.1$

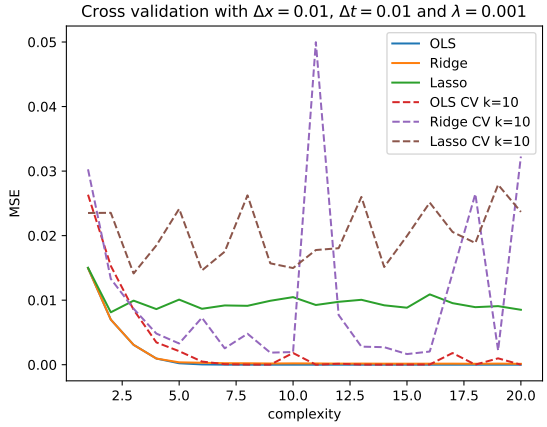


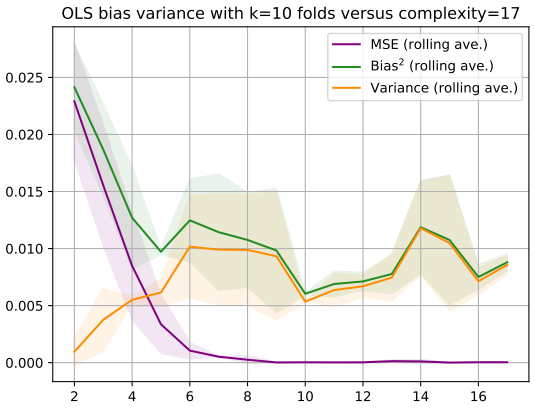
FIG. 6: The cross-validation MSE and MSE for OLS, Ridge and Lasso regression for $u(x,t)$ versus complexity with $\Delta x = 0.01$

Comparing the standard MSE to the cross-validation MSE, we see that as the complexity increases in Figure 5 and 5, the MSE converges towards zero. However, with cross-validation, the MSE seems to have less of a tendency to converge towards zero, except for with OLS. This may indicate overfittingh for higher order complexity. As such, the polynomial degree chosen for 4 was 10, as that seems to be the sweet spot before overfitting for Ridge for both values of Δx.

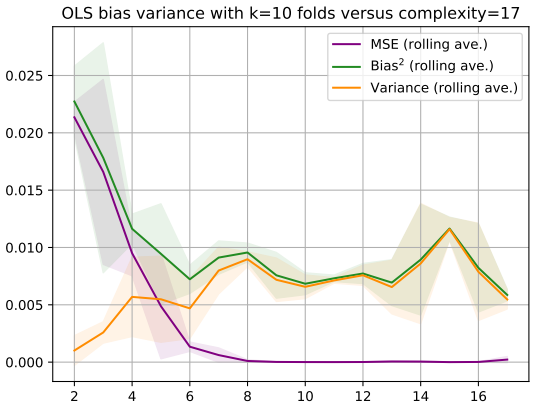
D. Bias Variance Analysis for linear regression models

To further asses the accuracy of OLS, Ridge and Lasso, the MSE, bias and variance is plotted as a function of complexity. The MSE, bias and variance is calculated from the resampled data from cross-validation with $k = 10$ folds.

In Figure 7, we observe initial bias and MSE to be high, with low variance. As expected, the MSE decreases, while the bias and variance hits its lowest values at complexity 10, as in Figure 5 and 5. Perhaps unexpectedly, the MSE does not increase for higher complexity, which can be a result of overfitting.



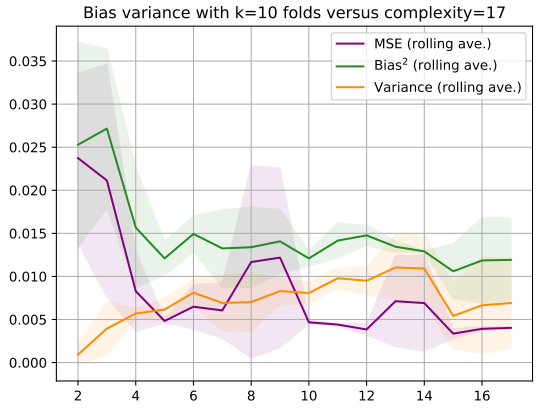
((a))



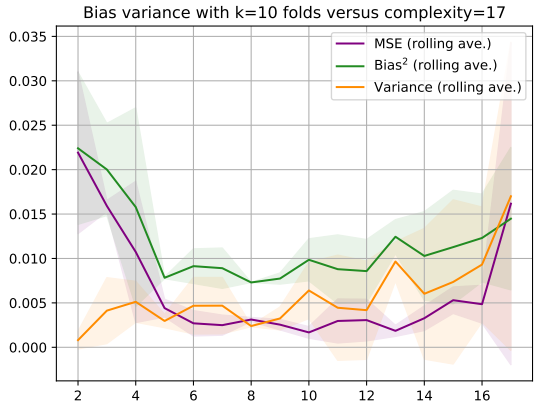
((b))

FIG. 7: The MSE, bias and variance as a function of complexity for OLS regression for the diffusion equation with $\Delta x = 0.1$ in a) and 0.01 in b).

In figure 8 (b), which has a penalty of $\lambda = 10^{-3}$, there is an increase in MSE for higher complexity compared to the OLS in Figure 7.

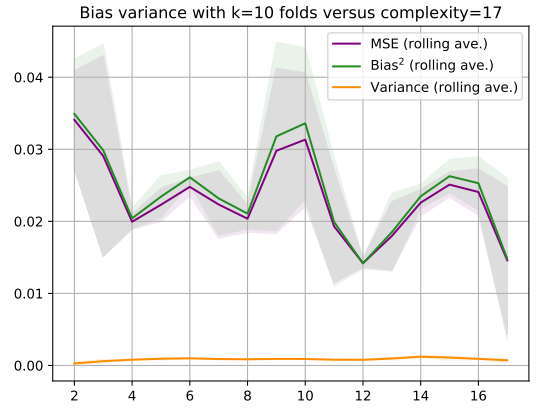


((a))

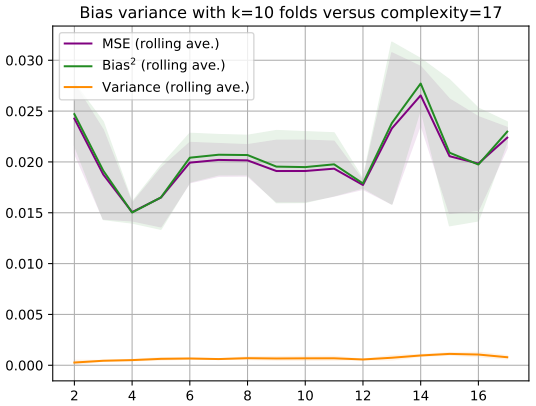


((b))

FIG. 8: The MSE, bias and variance as a function of complexity for Ridge regression for the diffusion equation with $\Delta x = 0.1$ in a) and 0.01 in b).



((a))



((b))

FIG. 9: The MSE, bias and variance as a function of complexity for Lasso regression for the diffusion equation with $\Delta x = 0.1$ in a) and 0.01 in b).

In figure 8 there's Lasso regression, which generally has an higher penalty than Ridge, where variance is very low compared to both OLS in Figure 7 and Ridge in 8. However, the MSE and bias is higher. Here, there is a clear trade-off between with low variance and high bias.

From the observations for bias, variance and MSE for OLS, Ridge and Lasso in Figure 7, 8 and 9, it is evident that OLS has the lowest MSE, probably due to over-fitting. Ridge has a somewhat more balanced trade-off between bias anb variance with a sweet spot for MSE at around complexity 10, whereas Lasso has higher overall MSE and bias but with very little variance.

E. Using neural network

One of the bigger challenges when training a neural network is choosing the appropriate parameters. As previously stated, we chose to use an adaptive learning rate. We use a three layered network, with 50, 25 and 25 neurons, after some brief testing, this seemed to yield the best results.

We still had do choose the number of data-points to include for x and t, as well as the number of epochs. As increasing both will cause very long run-time, we chose to plot them separately and reather discuss the differ-

ences. One simulation with 20×20 data-points and 500 epochs, and one with 50×50 data-points and 100 epochs.

We start by looking at the predictions for the two simulations compared to the analytical result, which is shown in figure 10 and 11.

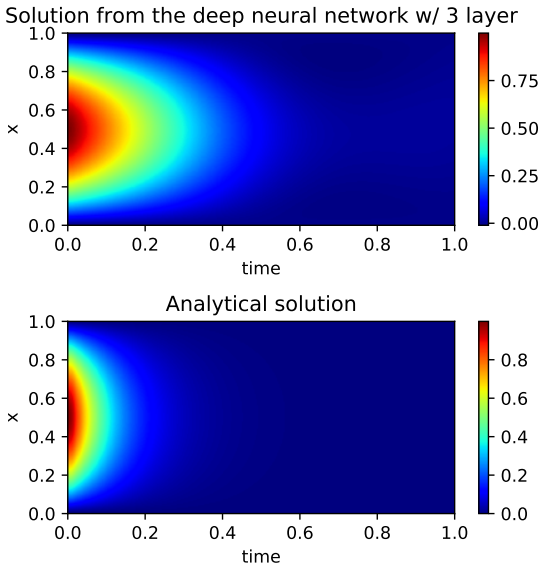


FIG. 10: This figure shows the analytical solution and the NN prediction, with 50×50 data-points and 100 epochs.

In figure 10 we see that the prediction offered by the neural network is somewhat similar to the analytical solution. However, in the solution by the NN, the rod takes way longer to cool down compared to the analytical solution.

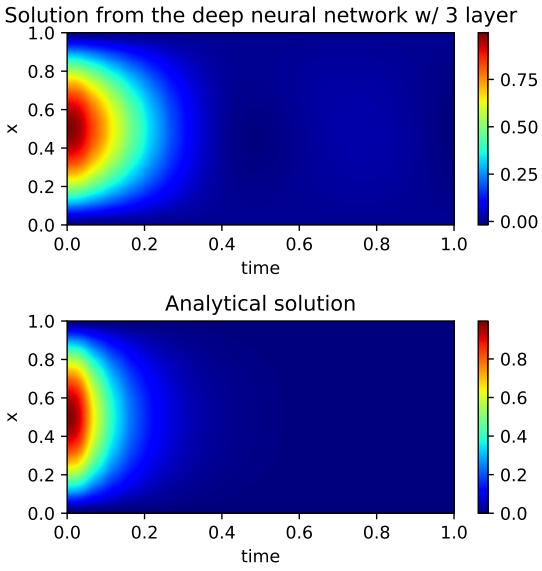


FIG. 11: This figure shows the analytical solution and the NN prediction, with 20×20 data-points and 500 epochs.

Compromising the number of data-points for more epochs, show in figure 11, we see the rod cool down slightly faster compared to figure 10. To further study what is happening in the two cases we must look at the difference between the analytical solution and the prediction made by the NN, which is shown in figure 12 and 13.

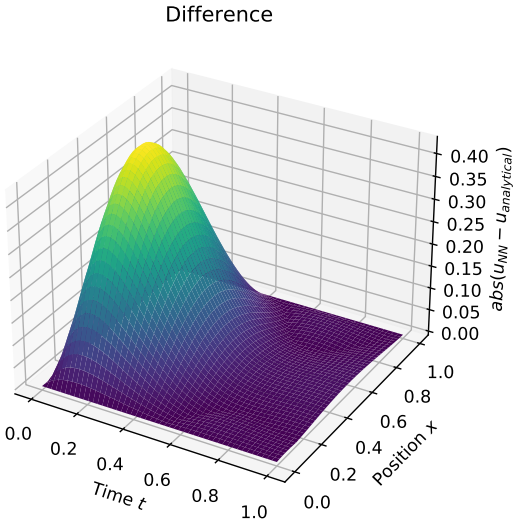


FIG. 12: This figure shows the absolute difference between the analytical solution and the NN prediction, with 50×50 data-points and 100 epochs.

Figure 12 shows that the absolute difference peaks around $t \sim 0.1$ and $x = 0.5$, which sounds reasonable if

we compare it to figure 10, as this is the region where the rod for the analytical solution has cooled down, while rod for the NN-solution has not. When $t \to 0$, we see steady decline in the difference as the rod in the NN-solution cools down. Looking at the difference close as $x \to 0$ and $x \to 1$, we see that the difference also declines. This is presumably because we move closer to the boundary condition described in equation 2 and 3.

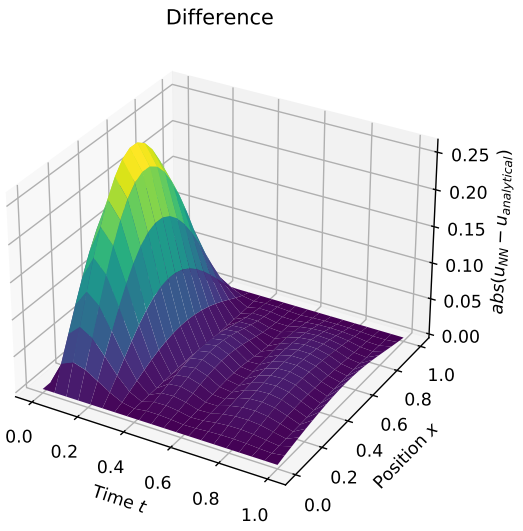


FIG. 13: This figure shows the absolute difference between the analytical solution and the NN prediction, with 20×20 data-points and 500 epochs.

In the case with 20×20 data-points and 500 epochs shown in 13, we see the same general pattern as we saw in figure 12. However the peak absolute difference has been reduced from ~ 0.4 to ~ 0.25, giving a better prediction.

To get a better sense of how the accuracy of the prediction is closer to the equilibrium state of the system, we also plotted the prediction of the final state of the system, which is shown in figure 14 and 15.

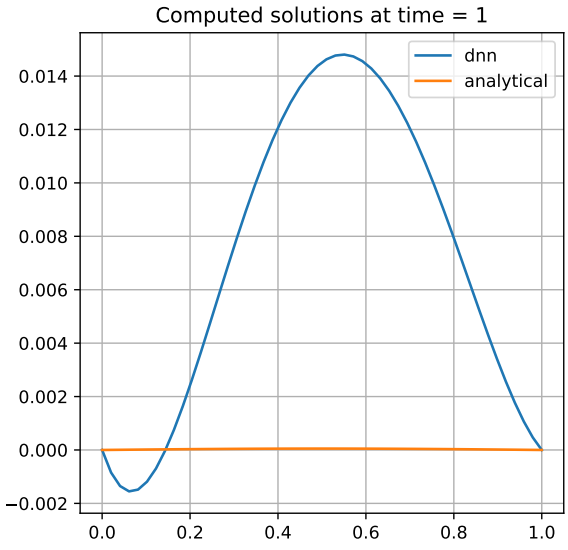


FIG. 14: This figure shows both the analytical solution and the NN prediction for the final time-step of the system, for 50×50 data-points and 100 epochs.

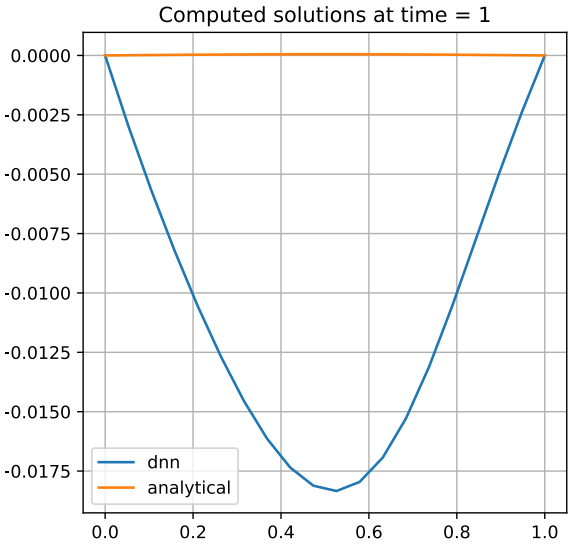


FIG. 15: This figure shows both the analytical solution and the NN prediction for the final time-step of the system, for 20×20 data-points and 500 epochs.

Contrary to what we saw in figure for the early states of the system such as $t \in [0,\ 0.5]$, the NN that uses more data-points has a lower absolute difference for the final state, than the NN that had more epochs. However, this difference is not as significant as the ones we saw for figure 12 and 13, as it is smaller, but also only represents a single time-state.

Further, we also ran the network using the *tanh* activation function. You can see the difference when using the tanh activation function from the analytical result in figure 16.

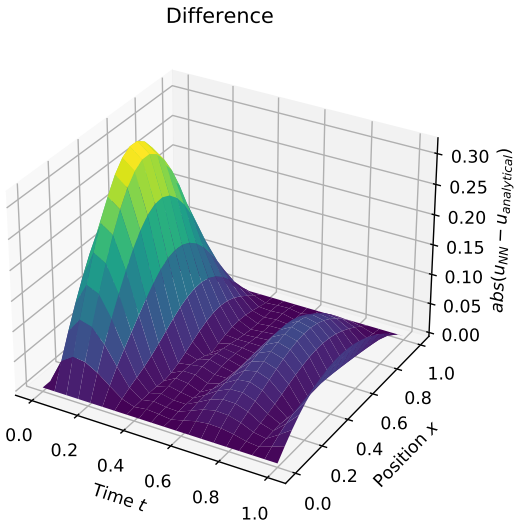


FIG. 16: This figure shows the absolute difference between the analytical solution and the NN prediction, with 20×20 data-points and 500 epochs, when using the tanh activation function

If we compare figure 13 and 16, we see that using the tanh activation function actually gave a slightly worse result, which is why we are focusing on the results where we use the sigmoid activation function.

As a control we also ran the network using the sigmoid activation function with 20×20 data-points and 100 epochs, its deviation from the analytical solution is shown in figure 17

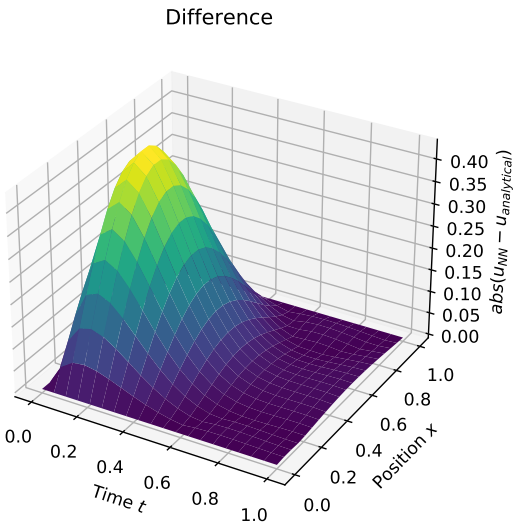


FIG. 17: This figure shows the absolute difference between the analytical solution and the NN prediction, with 20×20 data-points and 100 epochs.

Looking at the neural network solution for 20×20 data-points and 100 epochs shown in figure 17 we can observe two important behaviors. Comparing it to the simulation with 500 epochs, shown in figure 13, we see that increasing the number of epochs improves the data set's fit to the analytical solution. Comparing it to the simulation with 50×50 data-points, we see that an increase in data-points causes a *very* slight improvement in the fit to the data set.

Overall, the network prioritising number of epochs over the number of data-points generally performed better, as it had a lower difference to the analytical solution. This is not to say that the network made a good prediction, the prediction is still very bad. As previously stated, the network would have to produce a smaller peak absolute difference than $\sim 5e^{-5}$ to outperform Euler's explicit scheme. It is nowhere close to this limit with a peak absolute difference of $\sim 2.5e^{-1}$.

It is however worth noting the striking similarities between the absolute difference of the Euler explicit scheme, shown in figure 3, and the absolute difference for the neural network shown in figure 13. Both show a peak difference around $t \sim 0.1$ and $x = 0.5$, and then a steady decline from there as t increases. Both also show a decline in the difference as we approach the boundary conditions for $x = 0$ and $x = L$.

For us, it seems that the prediction offered by both the NN and the Euler explicit scheme, seems to struggle to adapt to the rapidly decreasing u leading to a large difference where the gradient of u is large. Increasing the

number of epochs in the NN, gives it more time to refine the gradient, and thus decrease the error/difference from u. This also explains why the network with 500 epochs generally performed better. In the regions where the gradient of u is small, the difference seems to be more stable and have about the same magnitude when having 50×50 data-points and 100 epochs, compared to the NN where we have 20×20 data-points and 500 epochs.

Training with a large amount of epochs, the network is able to calculate steeper gradients, however, this will also lead to more volatile gradients in the region where the they are supposed to be low. This is what we see in figure 13, where the peak difference is lower as the network can predict the steep slope, but after $t = 0.5$ the difference keeps fluctuating. Comparing figure 14 and 15, we can see that these fluctuations cause a larger difference for the final state for the network trained with 500 epochs, than that which was trained with 100.

We saw that increasing the number of data-points gave a slight improvement to the fit. Having more points will also cause the difference between each point to be smaller, this allows the network more time to adapt to a steep slope, and therefore also improve the fit, without causing big fluctuations in the difference in the low gradient region.

IV. Solving eigenvalue problems[3]

In this part we will calculate simulations by applying the theory part. We will therefore compute the eigenvectors corresponding to the max and min eigenvalues of a square symmetric real matrix and compare with the eigenvalues obtained by linalg. We will apply this to a matrix of size (6x6). To do this let's generate a matrix of size 6 x 6 using the equation 32:

$$A = \begin{pmatrix} 0.1508 & 0.4852 & 0.5617 & 0.2370 & 0.4601 & 0.6325 \\ 0.4852 & 0.4503 & 0.4126 & 0.7386 & 0.0683 & 0.4512 \\ 0.5617 & 0.4126 & 0.5879 & 0.2562 & 0.5119 & 0.4279 \\ 0.2370 & 0.7386 & 0.2562 & 0.1275 & 0.4829 & 0.5652 \\ 0.4601 & 0.0683 & 0.5119 & 0.4829 & 0.7885 & 0.6730 \\ 0.6325 & 0.4512 & 0.4279 & 0.5652 & 0.6730 & 0.9201 \end{pmatrix} \tag{34}$$

We will use our neural network to obtain the eigenvector associated to the maximum eigenvalue of A.

[4] In this part, we are no longer using an adaptive learning rate.

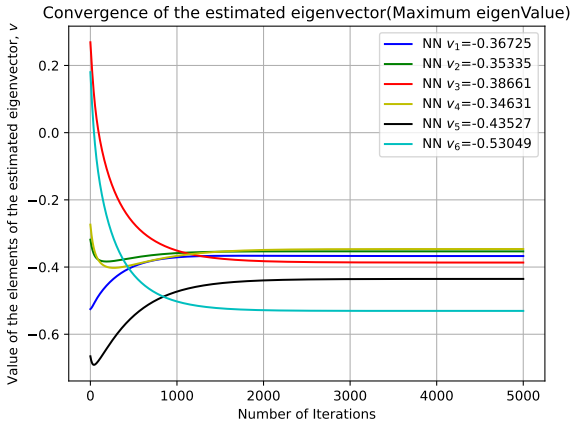


FIG. 18: Eigenvector component of the maximum eigenvalue for $\lambda = 0.001$, 5000 iterations, two layers of 50 and 25 neurons and sigmoid function as activation function

and there the loss of the *Neural Network* during the training :

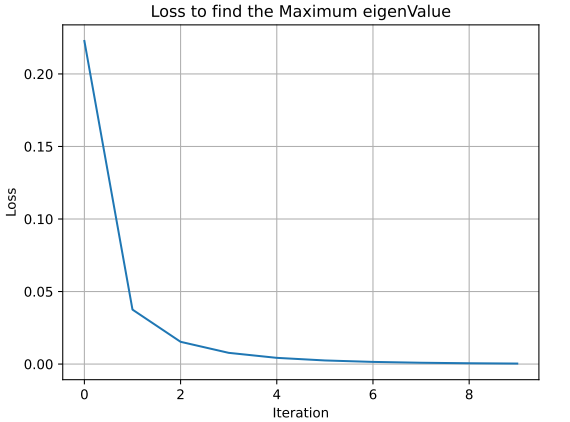


FIG. 19: Loss of the *Neural Network* during the training

From figure 18 we can see that our model converges and reaches its final value between 2000 and 3000 iterations. We can observe from figure 19 that in few iterations we have our loss goes throught zero then we obtain as maximum eigenvector and eigenvalue :

$$V_{max} = \begin{pmatrix} -0.3672 \\ -0.3533 \\ -0.3866 \\ -0.3463 \\ -0.4352 \\ -0.5304 \end{pmatrix} \text{ and } v_{max} = 2.89171616 \tag{35}$$

We do the same method to find the minimum eigenvalue, according to *Theorem 2* by using -A in the neural network.

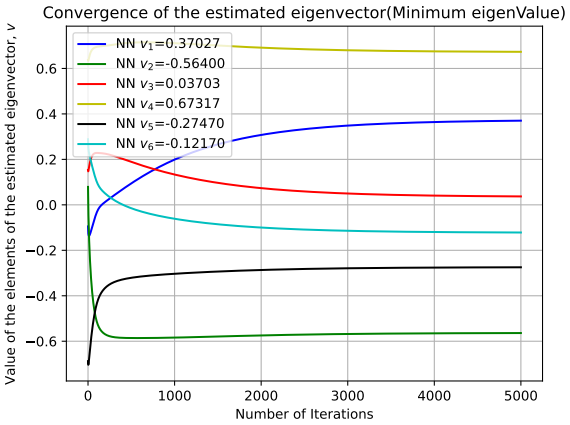


FIG. 20: Eigenvector component of the minimum eigenvalue for $\lambda = 0.001$, 5000 iterations, two layers of 50 and 25 neurons and sigmoid function as activation function

and there the loss of the *Neural Network* during the training :

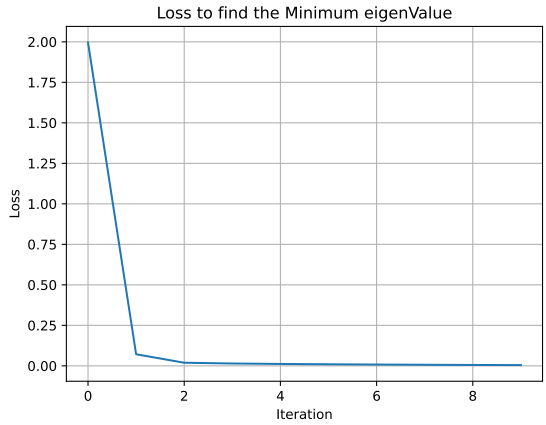


FIG. 21: Loss of the *Neural Network* during the training

From figure 20 we can see that our model converges but slower than for the maximum. We can also observe from figure 21 that in few iterations we have our loss goes through zero as for the maximum eigenvalue and :

$$V_{min} = \begin{pmatrix} 0.3702 \\ -0.5639 \\ 0.0370 \\ 0.6731 \\ -0.2747 \\ -0.1217 \end{pmatrix} \; and \; v_{min} = -0.64683469 \qquad (36)$$

We will now compare with the linalg method which gives us the exact eigenvalues:

```
Eigenvector analytic =
 [[-0.36725105 -0.37402628 -0.77387398  0.29187583 -0.20027224  0.03239748]
 [-0.35335958  0.56324556 -0.15700646 -0.23393012 -0.02635227  0.69124718]
 [-0.38661928 -0.03480392  0.4574133   0.14815268 -0.78491903 -0.04516858]
 [-0.34630328 -0.67192545  0.2588227  -0.47271958  0.24593708  0.27866143]
 [-0.43525453  0.27392142 -0.16279875 -0.52334115  0.0464615  -0.65800843]
 [-0.53048727  0.12300699  0.27157152  0.58376898  0.52958801 -0.09197959]]


Eigenvalues analytic =
 [ 2.89171616 -0.64684314 -0.28711168  0.1030072   0.34578947  0.61875794]
```

FIG. 22: eigenvalues and eigenvectors from linalg for the matrix A

Comparing out NN solution shown in expression 35 and 36 to the analytical solutin shown in figure 22, we can validate that our neural network gives us converges well to the eigenvectors associated with the maximum and minimum eigenvalues of A with a precision of lambda = 0.001.

Now we will study how the different hyperparameters influence the convergence of the eigenvectors.

First we will study the influence of the activation function by running our model with the *tanh* function:

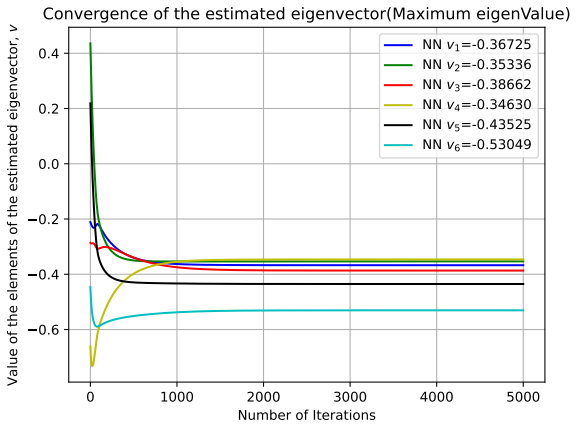


FIG. 23: Eigenvector component of the minimum eigenvalue for $\lambda = 0.001$, 5000 iterations, two layers of 50 and 25 neurons and tanh function as activation function

We can see in figure 23 that for the tanh activation function the model still converges, indeed the theorem 1 does not depend on the activation function. We can nevertheless observe that the convergence is faster than for the sigmoid function.

How does the number of iterations affect the convergence of the model? Normally when we decrease the number of iterations until we reach a point before convergence, the accuracy of our eigenvalues decreases. However, we could observe in some cases using the sigmoid activation function which converges less quickly that if we reduce the number of iterations too much, our eigenvector solution converges but to an eigenvector of a value which is not the maximum or minimum value. We can't explain it and we can't find any study on this subject. It seems that we can determine other eigenvalues of a matrix by playing with the convergence. However we need to know the exact eigenvalues to know if we converge to another eigenvalue. So it becomes a case by case approach, so less useful.

How does the learning rate affect the convergence of the model? The learning rate represents the speed at which we will try to converge our gradient. However, if it becomes too high our model may diverge or not stabilize. What we can see here :

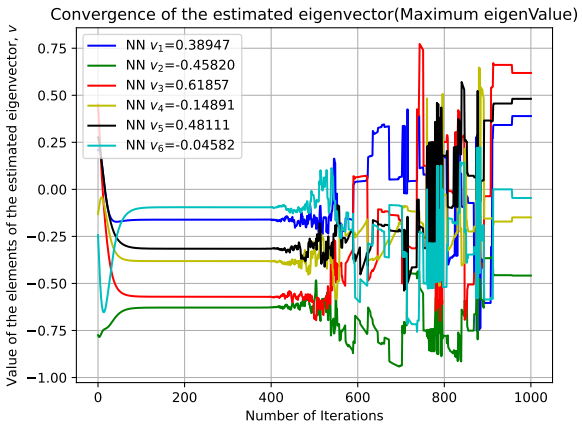


FIG. 24: Eigenvector component of the minimum eigenvalue for $\lambda = 0.01$, 1000 iterations, two layers of 50 and 25 neurons and tanh function as activation function

Moreover we can see that our model converges quickly and that the cost in computation time is very low so it is not useful to try to have the highest possible learning rate, a low learning rate ensures us a convergence and a higher precision.

Can we reduce the number of layers to one? Indeed, we can see that only two layers are needed to make our model converge, but let's study if we can use only one:

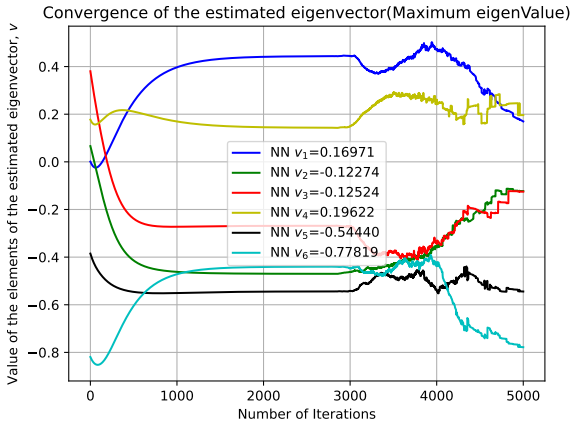


FIG. 25: Eigenvector component of the minimum eigenvalue for $\lambda = 0.001$, 5000 iterations, one layers of 50 neurons and tanh function as activation function

We can tell from figure 25 that our model cannot converge for a single layer. If we reduce the number of iterations to avoid divergence, we obtain a vector which is not an eigenvector of the matrix A, so the optimal number of layers for reasonable size matrices is 2. What about for high dimensional matrices ? For this part we will run our program for a matrix of size (50x50) :

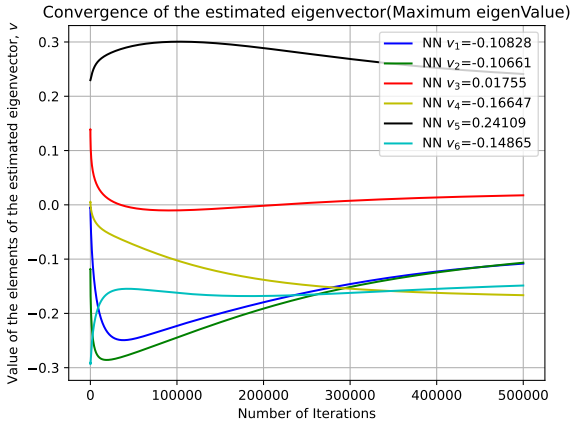


FIG. 26: Eigenvector component of the minimum eigenvalue for $\lambda = 0.0001$, 500 000 iterations, two layers of 50 and 25 neurons and sigmoid function as activation function for 50x50 matrix size

During the simulation we see that for a matrix of this size we diverge very quickly so we use the sigmoid function which converges less quickly. Then we have to reduce the learning rate to limit the divergence but this reduces the speed of convergence so we have to increase considerably the number of iterations. However, we can see that this is not enough and we do not converge to the eigenvector associated to the maximum eigenvalue.

With NN: $v_{max} = 1.75922481$, with linalg: $v_{max} = 25.2674682$. We can therefore conclude that for high

dimensional matrices, neural networks are not efficient.

We can also look at what happens for a non-symmetric matrix even though the proofs of theorems 1 and 2 have been done for symmetric matrices.

$$A = \begin{pmatrix} 0.7463 & 0.6801 & 0.0897 & 0.9761 & 0.2048 & 0.5579 \\ 0.8790 & 0.3903 & 0.2568 & 0.3698 & 0.2239 & 0.9547 \\ 0.9254 & 0.7366 & 0.0824 & 0.4081 & 0.0296 & 0.3281 \\ 0.4686 & 0.5237 & 0.4252 & 0.4336 & 0.6119 & 0.5263 \\ 0.0528 & 0.6259 & 0.0862 & 0.1766 & 0.4247 & 0.1458 \\ 0.9400 & 0.7903 & 0.1673 & 0.2539 & 0.0693 & 0.8808 \end{pmatrix} \tag{37}$$

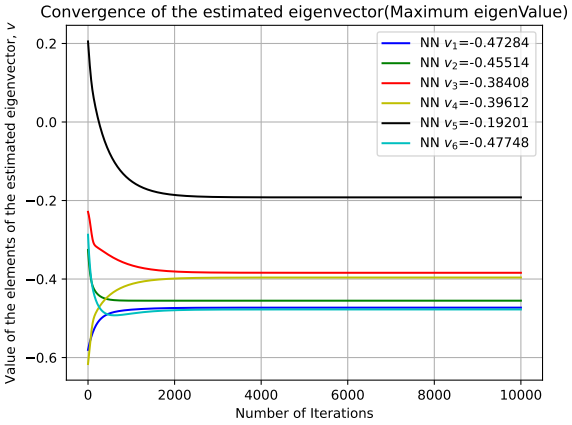


FIG. 27: Eigenvector component of the minimum eigenvalue for $\lambda = 0.001$, 10 000 iterations, two layers of 50 and 25 neurons and sigmoid function as activation function for 6x6 matrix size

We can observe from figure 27 that we have a convergence of the model to a vector. So we will study if this vector is an eigenvector of A. According to our program this vector is an eigenvector of A associated with the maximum eigenvalue: With NN : $v_{max} = 2.9383385$ with linalg we obtain the list of eigenvalue : [2.93833849+0.j, 0.28837559+0.25221242j, 0.28837559-0.25221242j, -0.0674939 +0.25550732j, -0.0674939 -0.25550732j, -0.42187051+0.j].

Thus, our model gives us the maximum real eigenvector of A. We don't know if the neural network would be able to work on all non-symmetric matrices but it seems that in some cases it is possible.

V. Conclusion

A. Numeric and linear regression models for solving diffusion equation

The first method to be addressed, the explicit scheme, is a method specifically developed to solve the diffusion equation with the specific initial conditions given. Of all the methods used, the explicit scheme produces the smallest difference to the actual analytical values. However, as mentioned already, the explicit scheme is only applicable to this specific type of problem, while the linear regression models and neural network models can be applied to many types of problems.

As for the linear regression models, they have comparable but with the optimal parameters and choice of algorithm better approximation of the solution than the neural network. However, when the error is very low, as with high complexity OLS, the results are prone to overfitting, which could be problematic in cases with much noise.

The bias variance analysis is an attempt at investigating which linear regression model to use for what scenarios. To help compare these methods with the rest, the test data was the analytical solution of the hear equation without noise, so that tends to favour the OLS. However, it appears that the Ridge regression at complexity 10 has a good balance of bias and variance, with low MSE, which inclines us to favour this algorithm. If the amount of noise is unknown, then this is even more critical, and Lasso might even be favoured for its low variance if the noise is expected to be high. The more complex and uncertain the noise, the differential equation and the problem, the more sophisticated and general the approach should be.

B. Neural network for diffusion equation

The neural network solution for the diffusion equation is simply underwhelming with an error much larger then it's numerical *Euler explicit scheme* counterpart. The network suffers from a somewhat paradoxical shortcoming. If not not given enough epochs, it struggles to predict the steep initial gradient, causing a large difference between the prediction and the analytical solution. When the network runs with more epochs, it handles the initial slope better, but is then too sensitive small fluctuations after the rod has cooled, again causing a large difference.

With both the number of epochs and data-points drastically increasing run-time, we are unable to avoid this error. An increased number of hidden layers and neurons might allow the network to better adapt to a wider set of gradients, it will also come at a computational cost.

Although the use of neural networks to solve differential equations is very useful for high dimensional problems, it cannot compete with it's 'traditional' numerical counterpart, in the case of a one-dimensional diffusion equation.

C. Solving eigenvalue problems with a neural network

We could also study another use of neural networks, that of being able to solve eigenvalue problems and provide, for a certain type of matrix, the maximum and minimum eigenvalues by obtaining their eigenvectors.

For this we use our neural networks to solve a nonlinear differential equation. We also used Autograd to take derivatives which is very easy to operate.

From our simulations, we can conclude that the neural network method for solving the eigenvalue problem works very well and converges very fast for reasonable size matrices. However, it becomes less advantageous for large matrices as it will require more iterations to converge. Moreover, the matrices were supposed to be symmetric which reduces the number of problems for which we can apply this method. We have seen however that this method can work for non-symmetric matrices, which could be an interesting topic for future work. Nowadays neural networks are indispensable tools in the solution of many problems and we have just seen a new use case. It is not as powerful as linear algebra so there is little interest in using it but it shows us the power of neural networks.

D. Final comments

It's quite remarkable how many problems we are able so solve using machine learning methods. With the same underlying principles we can do everything from solving differential equations and eigenvalue problems to classification of breast cancer data[5]. Although the machine learning methods did not perform up to par compared to their numerical counterpart in this paper, it is important to remember that these numerical methods are highly specialized and only intended for this purpose. It is also worth noting that there is no constraint stopping us from improving the performance of the machine learning methods except computational power, which is often not the case for numerical methods. All this demonstrates the usefulness, adaptability and power of machine learning methods.

In conclusion, although machine learning methods are very powerful, we still need to play to it's strengths. Neural networks are generally good at reducing the dimensionality of problems, and dealing with problems where we have no reliable numerical methods. Similar to using an advanced regression algorithm on a straight line will be more costly than simply calculating the inclination and intercept of the line, using a neural network two solve the one dimensional diffusion equation is more costly than using Euler's explicit scheme. It does not undermine the usefulness of machine learning methods, but simply highlights that we need to apply such methods to a suitable problem.

E. Future work

As previously mentioned, exploring how the solutions of non-symmetric matrices could be solved is an exiting application for further study. We also wish to examine how we could better build a neural network that can handle huge variations in the gradient of the solutions, as this is the main shortcoming of our network.

VI. References

[1] Yi et al. in the article from Computers and Mathematics with Applications 47, 1155 (2004)

[2] Stephan Dreiseitl, Lucila Ohno-Machado,"Logistic regression and artificial neural network classification models: a methodology review," Journal of Biomedical Informatics, 2002

[3] Micheal A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

[4] Friedman, Jerome H. The elements of statistical learning: Data mining, inference, and prediction. springer open, 2017.

[5] Mehta et al, arXiv 1803.08823, A high-bias, low-variance introduction to Machine Learning for physicists, ArXiv:1803.08823.

[5] as done in project 2