FYS-STK 3155 Project 3 Erlend Abrahamsen

December 11, 2019

Abstract

Partial differential equations are very important and have wide applications, one example is the use of Navier-Stokes equations for weather forcasts. Much study and improvement of solvers for these are crucial. In this project we study deep neural networks (DNN) for solving a version of the heat equation as well as a rather ambitious ODE approach for finding eigenvalues. For comparison of the heat equation results we also solve it analytically and with the numerical finite differences method. Results showed that the DNN was efficient and had descent accuracy while finite differences had great accuracy with poor computational efficiency on large data. For eigenvalues the results were not that good and the predicted eigenvalues had chaotic behavior and oscillated between many of the true eigenvalues.

1 Introduction

For the source code (folder Source_code) and the results (folder Results) go to my Github: `https://github.com/ErlendAbrahamsen/FYS-STK3155/tree/master/Project3`

PDE's are widely used for explaining real physical problems and are generally approximated using numerical methods. An example of great interest is modeling of fluid dynamics. More computational efficiency is much needed in this area, and with a lot resent success of Neural networks, it is interesting to investigate a deep learning approach on PDE's.

In this project we explore the finite differences scheme and a feed forward

Neural network based on Tensorflow functionality for solving the 1d heat equation given some nice initial condition.

In the article Computers and Mathematics with Applications (reference 2, Authors Yi, Fu and Jin Tang: `https://www.sciencedirect.com/science/article/pii/S0898122104901101`) they have an interesting approach of predicting eigenvectors/eigenvalues by solving ODE system with recurrent neural network. Eigenvalues are essential in linear algebra, and we will also attempt their method with a feed forward network.

Outline [Introduction, Methods, Results and discussion, Summary]
(I will reference to source code/project part where relevant!)

1.1 The 1d heat equation (Part a)

The general heat equation is $\frac{\partial u}{\partial t} = \alpha[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}]$, where $u(x, y, z, t)$ is the temperature of a medium at position (x, y, z) in time t. The α constant is thermal diffusivity of the medium. We look at a 1d case with $\alpha = 1$.

Given a 1d rod of length $L = 1$, from position $x = 0$ til $x = 1$, we want to find the temperature $u(x, t)$ at any position in time (x, t).

We set the initial heat distribution for time $t = 0$ as $u(x, 0) = sin(\pi x)$ and constant zero temperatures at endpoints $u(0, t) = u(L, t) = 0$.

Heat equation with chosen conditions:

$u_t(x, t) = u_{xx}(x, t)$, $t > 0$, $x \in [0, 1]$

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

$u(x, 0) = sin(\pi x), 0 \leq x \leq 1$ (Initial conditions)

$u(0, t) = u(1, t) = 0$, $t \geq 0$ (Boundary conditions)

The solution is $u(x, t) = e^{-\pi^2 t} \sin(\pi x)$.

2 Methods

2.1 Analytic solution of 1d heat pde (Part a)

For the solution we use the pde technique separation of variables. Also note that for most initial conditions infinite Fourier series are computed. This is not needed for our simplified version and we avoid extra errors for lack of terms used.
Sep. of var.: Assume $u(x,t) = X(x)T(t)$ for some $X, T : R \to R$.

Plugging this into $u_t = u_{xx}$ yields $\frac{T'(t)}{T(t)} = \frac{X''(x)}{X(x)}$,

which has to be some constant $\lambda \in R$, since t and x are independent variables. Further we get two ODE's $T'(t) - \lambda T(t) = 0$ and $X''(x) - \lambda X(x) = 0$. These are easy to solve and have some different forms dependent on $\lambda = 0$, $\lambda > 0$ or $\lambda < 0$.

$\lambda = 0$: Using boundary conditions yield
$0 = u(0,t) = X(0)T(t) = u(1,t) = X(1)T(t)$ which implies $X(x) = 0$ since $X(x) = a + bx$, thus $\lambda = 0$ is excluded.

$\lambda > 0$: Again from boundary conditions we have $X(0) = X(1) = 0$ implying
$a = -b$ and $ae^{\sqrt{\lambda x}} - ae^{\sqrt{\lambda x}} = 0$, since $X(x) = ae^{\sqrt{\lambda}x} + be^{-\sqrt{\lambda}x}$, giving $X(x) = 0$. $\lambda > 0$ is also excluded.

$\lambda < 0$: $X(x) = a\cos(\sqrt{-\lambda}x) + b\sin(\sqrt{-\lambda}x)$ and $T(t) = ce^{\lambda t}$,
giving $a = X(0) = 0$, and $X(1) = b\sin(\sqrt{-\lambda}x)$ yielding $\sqrt{-\lambda} = \pi n$, $n \in N_1$. This we can solve for a none-zero solution.

Putting it all together we have now found the particular solutions
$u_n = b_n e^{\lambda t} \ sin(\sqrt{-\lambda}x) = b_n e^{-(\pi n)^2 t}\sin(\pi n x)$, giving the general solution $u = \sum_{n=1}^{\infty} u_n$ by super position principle.

$u(x,0) = \sin(\pi x) = \sum_{n=1}^{\infty} b_n \sin(\pi n x)$, so $b_1 = 1$ and $b_{n \neq 1} = 0$
giving our final general solution.

$u(x,t) = e^{-\pi^2 t}\sin(\pi x)$ for $t \geq 0$, $x \in [0,1]$.

Verifying the solution:
$u_t(x,t) = -\pi^2 e^{-\pi^2 t}\sin(\pi x)$

$u_{xx}(x,t) = \frac{d}{dx}\pi e^{-\pi^2 t} cos(\pi x) = -\pi^2 e^{-\pi^2 t}\sin(\pi x) = u_t(x,t)$

$u(x,0) = \sin(\pi x)$, $u(0,t) = u(1,t) = 0$

2.2 Numerical approximation with finite differences (Part b)

Using 1. order taylor expantions we obtain $f'(x) = \frac{f(x+\Delta x)-f(x)}{\Delta x} + O(\Delta x)$ and

$f''(x) = \frac{f'(x)-f'(x-\Delta x)}{\Delta x^2} + O(\Delta x^2) = \frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2} + O(\Delta x^2)$.

We use this to set up the forward time, centered space, discrete approximation:

Using the derivative approximations on $\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}$ we let $u(x_i, t_j) = u_i^j$:

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

$u_i^{j+1} = u_i^j + \frac{\Delta t}{\Delta x^2}(u_{i+1}^j - 2u_i^j + u_{i-1}^j)$, let the ratio $\frac{\Delta t}{\Delta x^2} = d$

$u_i^{j+1} = du_{i+1}^j + (1-2d)u_i^j + du_{i-1}^j$, $i = 1, ..., n-1$, $j = 0, 1, ..., m-1$

With conditions:

$u_i^0 = \sin(\pi x_i)$, $i = 1, ..., n-1$

$u_0^j = u_n^j = 0$, $j = 0, 1, ..., m$

We choose $\Delta t/\Delta x^2 \leq 1/2$, as it ensures numerical stability and
set up the grids $x_i = \Delta x \cdot i \in [0,1]$, $t_j = \Delta t \cdot j \in [0, t_{stop}]$ and iterate explicitly with double loop.

We see that it runs in time $O(len(x) \cdot len(t)) = O(\frac{1}{\Delta x} \cdot \frac{t_{stop}}{R\Delta x^2}) = O(1/\Delta x^3)$.

(See finit_diff() in Part_b.py for Python implementation)

2.3 Neural Network learning PDE solution (Part c)

We will utilize the flexibility of Tensorflow to explore DNN's with an appropriate defined cost function. By the universal approximation theorem we know that a simple neural network can approximate any sufficient continuous function.

We let the approximation of PDE solution $u(x,t)$ be $\hat{u} = N(x,t)$, where $N = a^L$ is the DNN output of the final layer L.

The activation at each layer $l = 1, ..., L$ is $a^l = f(a^{l-1}W^l + b^l)$ for some activation function $f \in R$ and input $a^0 = [x' \quad t'] \in R^{(n \times 2)}$.

Dimensions:
W^l: $(2 \times m)$, $(m \times m)$, ..., $(m \times m)$, $(m \times 1)$ for $l = 1, 2, ..., L-1$
(m is #hidden neurons).

b^l: $(n \times 2)$, $(n \times m)$, ..., $(n \times m)$, $(n \times 1)$ for $l = 1, 2, ..., L-1$

a^l: $(n \times 2)$, $(n \times m)$, ..., $(n \times m)$, $(n \times 1)$ for $l = 0, 1, ..., L-1, L$
(n is length of x, t meshgrid)

Clearly we want $a^L \in [0,1]$ since $u(x,t) \in [0,1]$. A possibility for this is using $sigmoid(t) = \frac{1}{1+e^{-t}}$ as the output function, else one should think of good candidates and test them.
For optimizing the weights W^l and biases b^l we utilize Tensorflow's Adam optimizer, which uses variable learning rate and usually performs better than SGD (stochastic gradient descent).
For further explanations on minimizing the DNN loss see section 2.2 in Project 2 `https://github.com/ErlendAbrahamsen/FYS-STK3155/blob/master/Project2/Report.pdf`

Also for extra info on Adam optimizer see `https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c`)

(*HeatLearner class in Part_c.py for DNN implementation*)

2.3.1 Defining cost or loss for PDE problem

We need our cost to catch properties and conditions:

$u_t - u_{xx} = 0$ (i)

$u(x, 0) - sin(\pi x) = 0$ (ii)

$u(0, t) = 0$ (iii)

$u(1, t) = 0$ (iv).

A good suggestion would be

$L(x, t) =$

$MSE[\hat{u}_t, \quad \hat{u}_{xx}] + MSE[\hat{u}(x, 0), \quad sin(\pi x)] + MSE[\hat{u}(0, t), \quad 0] + MSE[\hat{u}(1, t), \quad 0]$

This punishes each property (i-iv) equally for being non-zero and the DNN learns them when $L \to 0$.
We use this loss function and implement it as loss() in HeatLearner class.

2.4 Eigenvalue learning with Neural network (Part d)

(reference 2, Authors Yi, Fu and Jin Tang: `https://www.sciencedirect.com/science/article/pii/S0898122104901101`)

From the paper on eigenvalue computations (ref. 2) they proposed solving a vector differential equation with an artificial recurrent neural network for finding the eigenvectors/eigenvalues of a symmetric $n \times n$ matrix.

Let A be a $n \times n$ real symmetric matrix and $x(t) = (x_1(t), ..., x_n(t))^T \in R^n$.
Proposed vector ODE:

(1) $x'(t) = f(x(t)) - x(t)$, where $f(x) = [x^T x A + (1 - x^T A x)I]x$.

Theorem 3 in the paper states that each solution of (1) starting from any non-zero point $x(0) \in R^n$ will converge to an eigenvector $v = \lim_{t \to \infty} x(t)$ of

A.

We find the corresponding eigenvalue $\lambda = \frac{v^T \lambda v}{v^T v} = \frac{v^T A v}{v^T v}$.

Let $\hat{x}(t) = N(x_0, t)$ for some initial guess $x_0 \in R^n$ where N is the NN output.

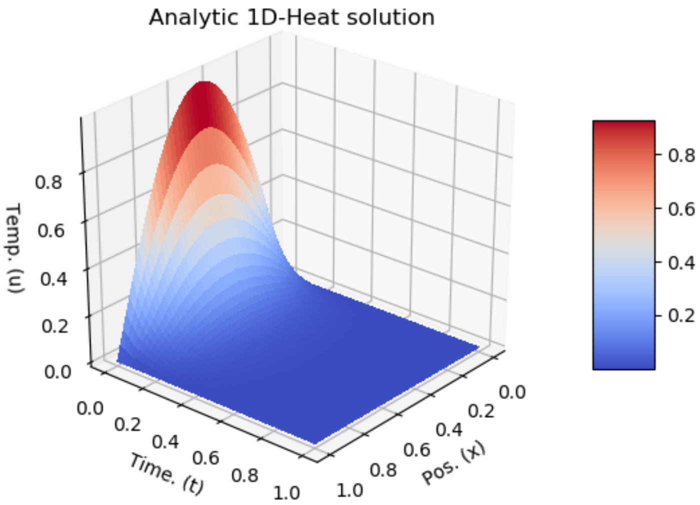
We set up the loss function $L(t) = mse[\hat{x}'(t), f(\hat{x}) - \hat{x}]$ since $\hat{x}'(t) \rightarrow f(\hat{x}) - \hat{x}$ when $L \rightarrow 0$.

In the implementation we reuse the PDE solver class HeatLearner. (*See Part_d for implementation*)

3 Results and discussion

3.1 Finite differences (Part b)

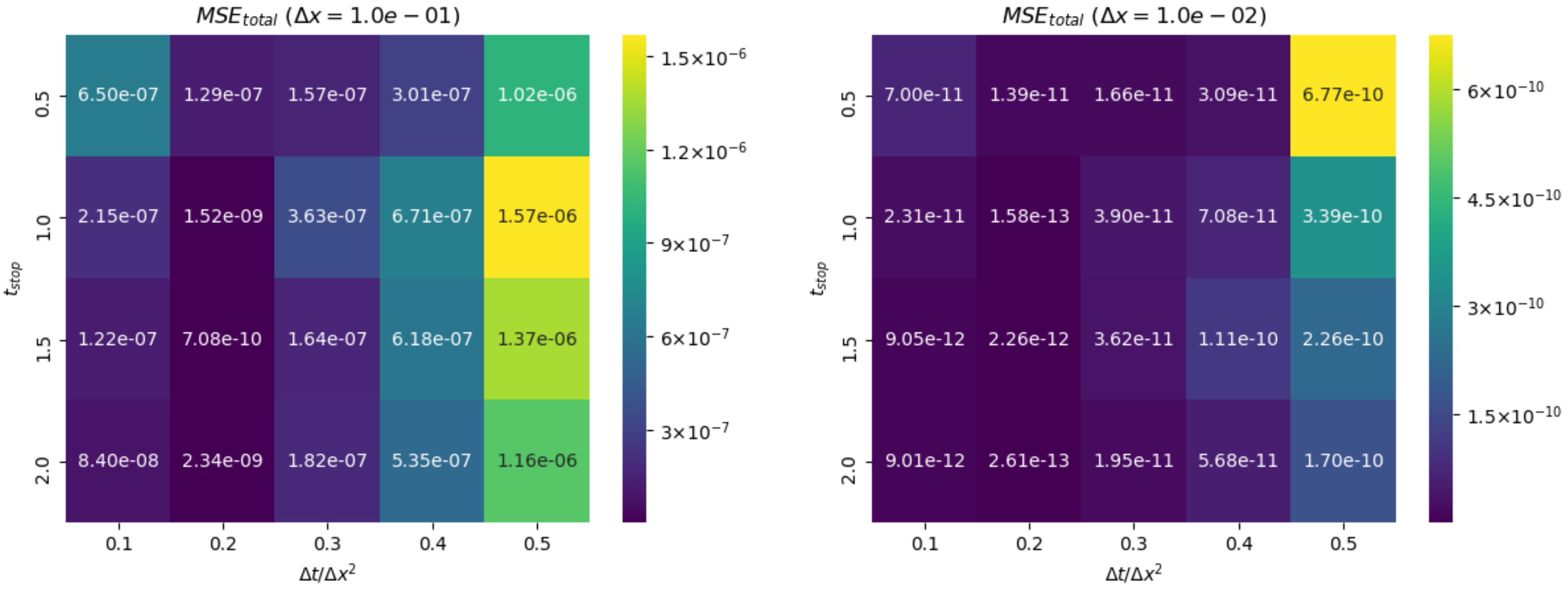
The analytic solution $u(x,t) = e^{-\pi^2 t} \sin(\pi x)$, $x, t \in [0,1]$:



Notice the quick temperature drop with respect to time caused by the exponential term.

For using the finit-diff. scheme we need the ratio $\Delta t/\Delta x^2 \leq 1/2$, else unstable results with massive erros occur.

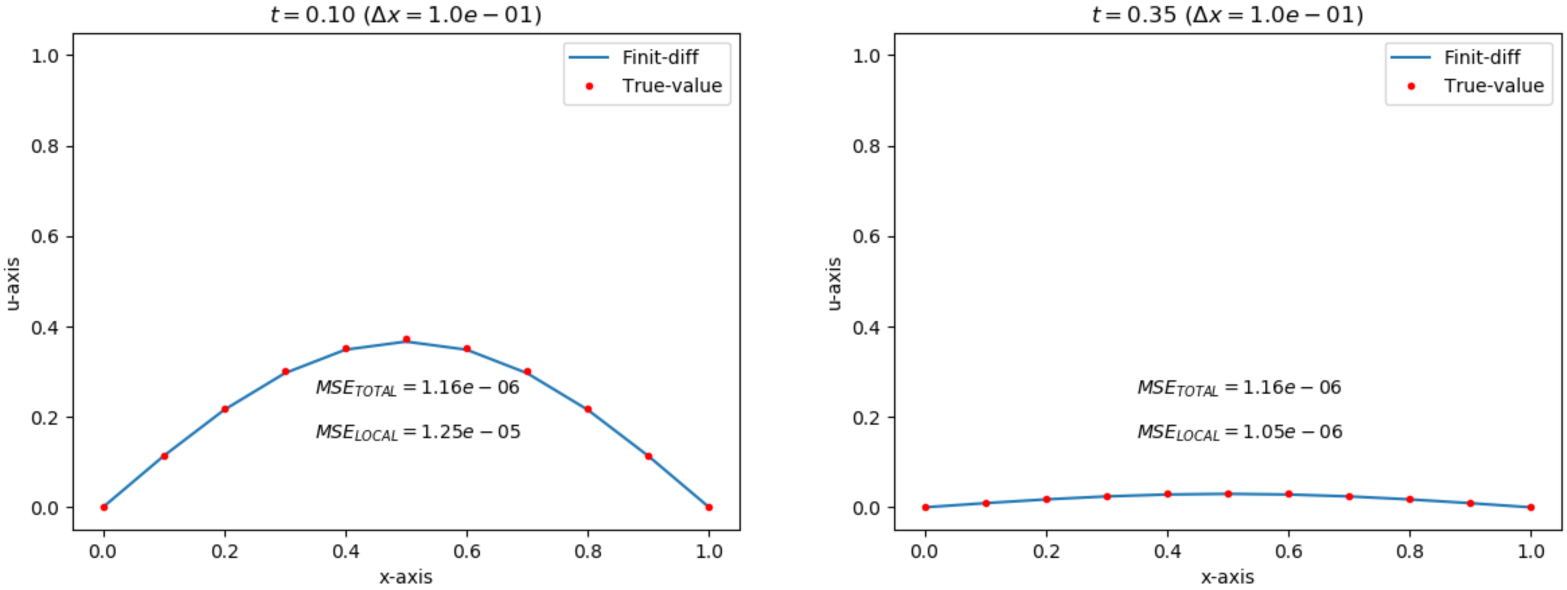
Tuning the model we create a heat-map, $MSE_{total} = MSE(u, \hat{u})$, for different time ranges $t \in [0, t_{stop}]$ and ratios $\Delta t/\Delta x^2$. We wont bother looking at $t > 2$ because of the exponential decay.



For fixed $(\Delta x)_1 = 10^{-1}$ and $(\Delta x)_2 = 10^{-2}$ we notice over all great results error-wise. The best ratio with respect to error is 0.2. Smaller ratios gives increased error.

Ratio at 0.5 is sufficient for both $(\Delta x)_1$ and $(\Delta x)_2$ at $t_{stop} = 2$, since MSE errors are respectively $1.16 \cdot 10^{-6}$ and $1.70 \cdot 10^{-10}$.
The model is not very flexible with respect to grid selection because of the stability criterion. Decreasing Δx is very computationally expensive and we might get a smaller Δt then necessary, which is a waste of time. The algorithm runs in $O(1/Deltax^3)$ time.

We look at the cheapest model ($(\Delta x)_1$ with $Ratio = 0.5$) at two fixed times $t_1 = 0.1$ and $t_2 = 0.3$:

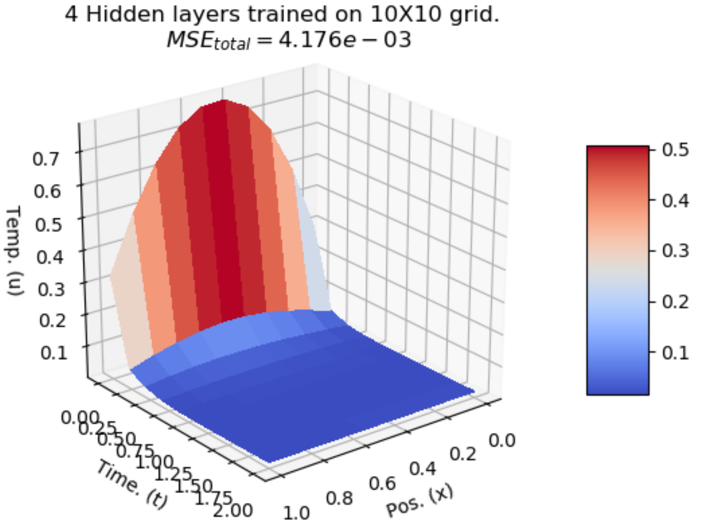


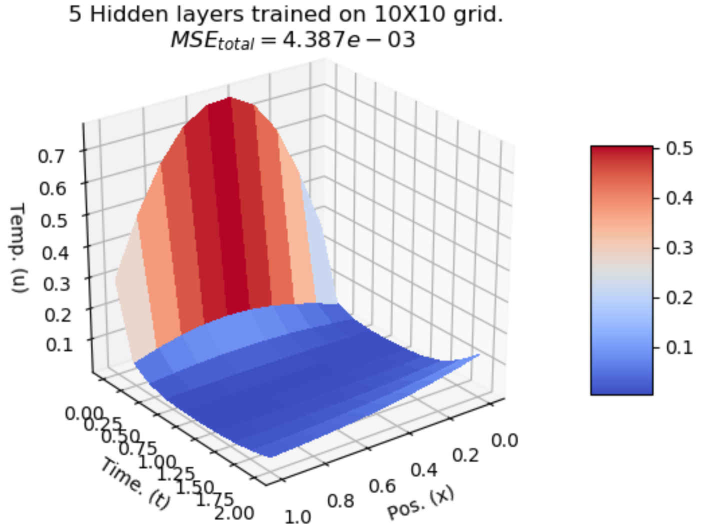
$(MSE_{local} = MSE[u(x, t = t_i), \quad \hat{u}(x, t = t_i)])$

At t_1 when $\hat{u}$ is significantly curved the local 'sine term' error is greater than the total error. When $t > t_2 = 0.3$ $\hat{u}$ is closer to the stationary state and $MSE_{total} > MSE_{local} \to 0$.

3.2 DNN on heat equation (Part c)

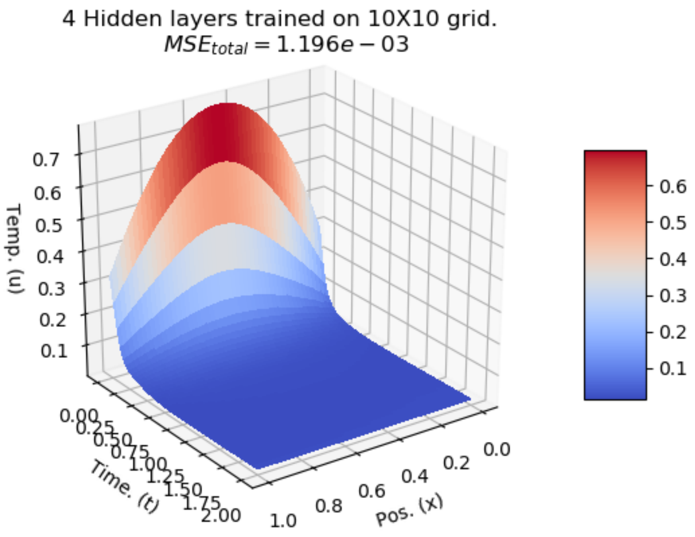
Trying Relu variations with some linear output function $f \in [0, 1]$ doesn't seem to do well with sine curves. It learns the exponential decay but along the x-axis we get very pointy linear parts. We end up using Sigmoid as hidden activation function and output function. The DNN output is then in $[0, 1]$ which we want.

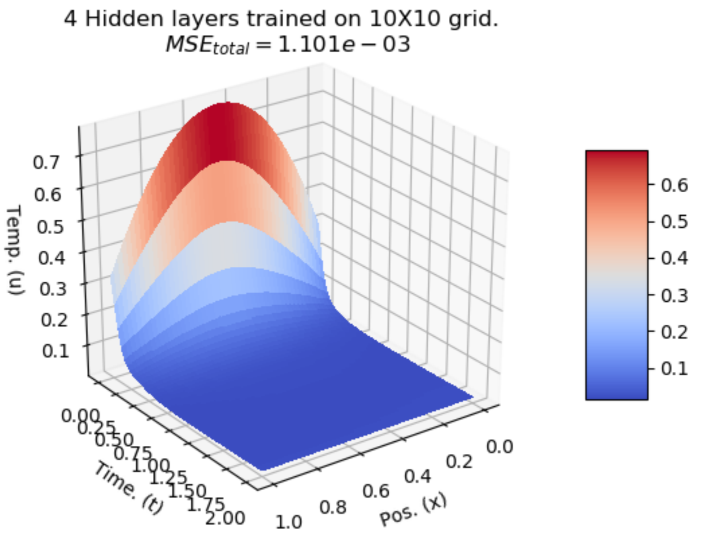




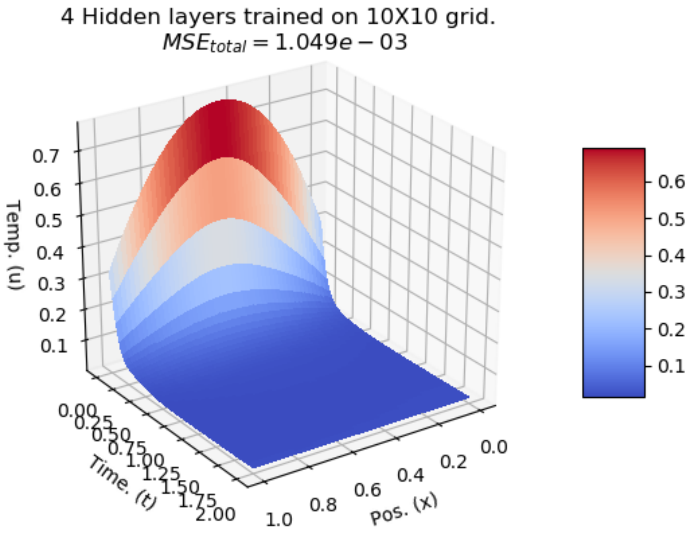
Adding more than 4 layers makes the model worse and at 5 hidden layers we see that $\hat{u}$ starts to curve again near $t = 2$ (Right figure). In the final model we end up with 4 hidden layers with 130 neurons each. The best we do are errors of magnitude 10^{-3} which is descent but not as good as 10^{-6} with finite-differences. The upside is that we can do fast training on a small grid and do relatively god prediction on much larger untrained grids, hence a big potential in computational efficiency gains. We train on small 10×10 grid (Training time 1.3 sec)

(100×100 grid prediction) (200×200 grid prediction)



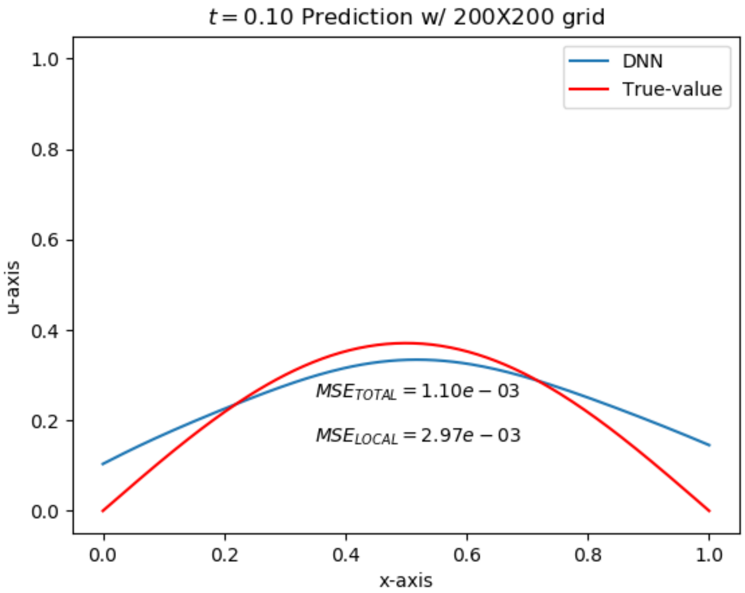


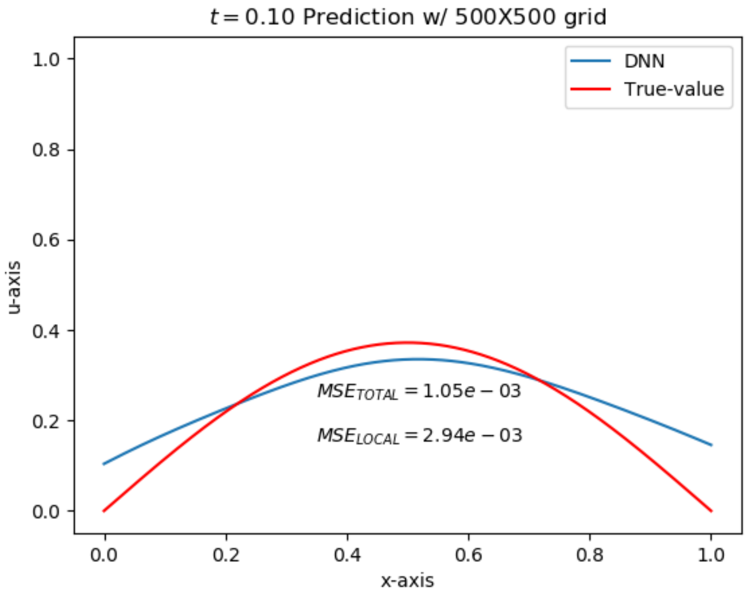
(500 × 500 grid prediction)



For 100 × 100, 200 × 200, 500 × 500 grids the errors are $MSE_{total} = \{1.20 \cdot$

$10^{-3}, 1.10 \cdot 10^{-3}, 1.05 \cdot 10^{-3}\}$ respectively. We should also calculate the local sine errors for fixed times and see if they get worse for bigger grids.



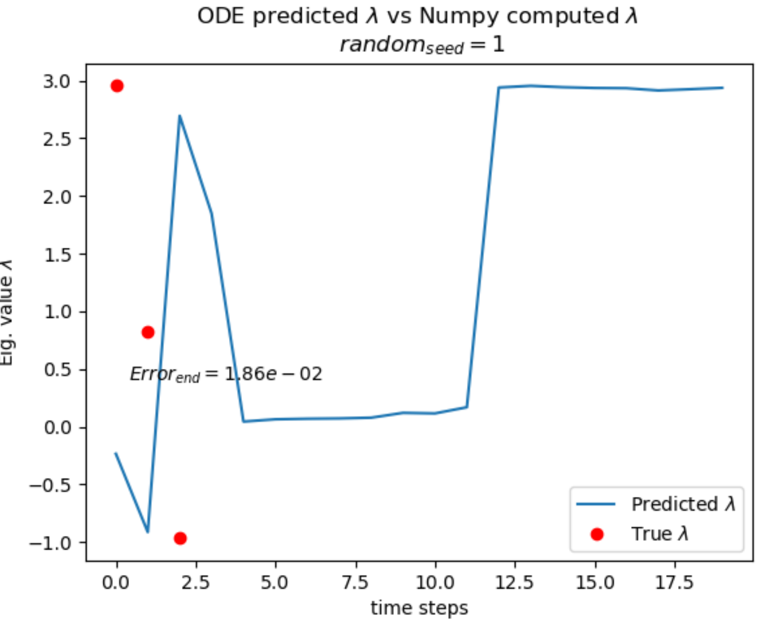


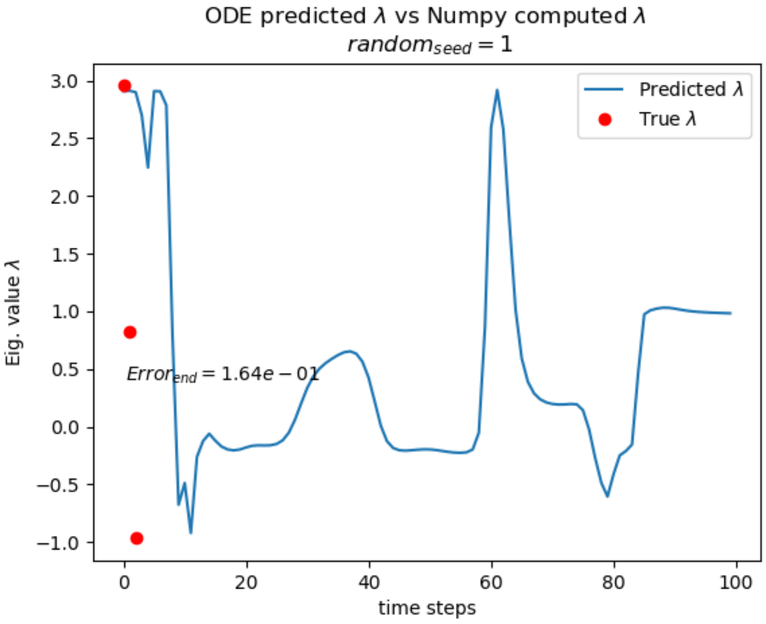
At $t = 0.1$ the cheapest finit-diff. model had MSE_{local} of magnitude 10^{-5}, here we have MSE_{local} of magnitude 10^{-3}. The DNN x end points are a bit bad and it looks like the model is averaging initial condition ($u(x, 0) = sin(\pi x)$, $x \neq \{0, 1\}$) and the boundary conditions ($u(0, t) = u(1, t) = 0$).

3.3 DNN for eigenvalues (Part d)

We look at different pseudo random symmetrical matrices using numpy seeds. Because of very unreliable convergence results we change to SGD optimizer attempting to get some control by setting a learning rate. We ended up with $tanh$ as activation and output function using a architecture of 4 hidden layers with 50 neurons each. The model is extremely unreliable and not precise enough as we sometimes get absolute error $|\lambda_{predicted} - \lambda_{numpy}|$ of magnitude 10^{-2}. The overall results are still rather interesting. As you will see in the plots the predictions jumps chaotically between multiple eigenvalues.
We test with matrices up to 6×6, $Error_{end}$ is absolute error between nearest eigenvalue and the last predicted one.

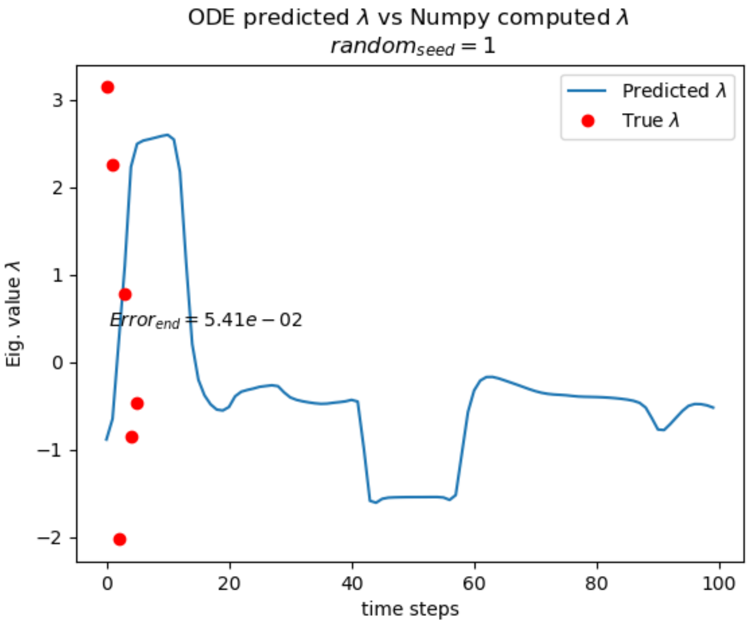
We seem to get the best results with 3×3 (or less) matrices:

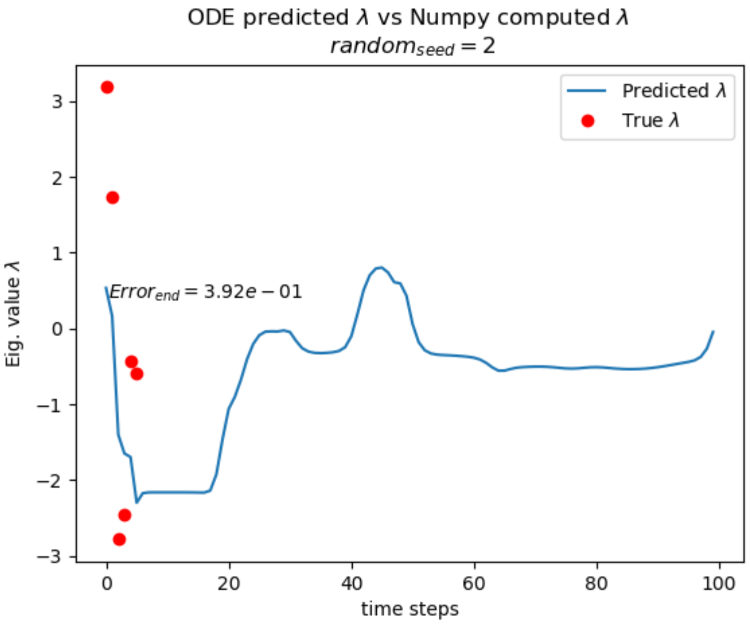


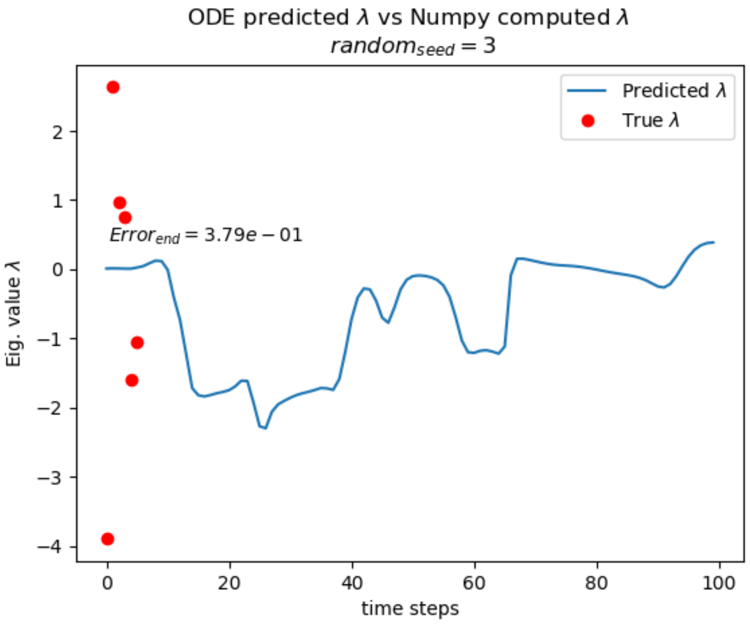


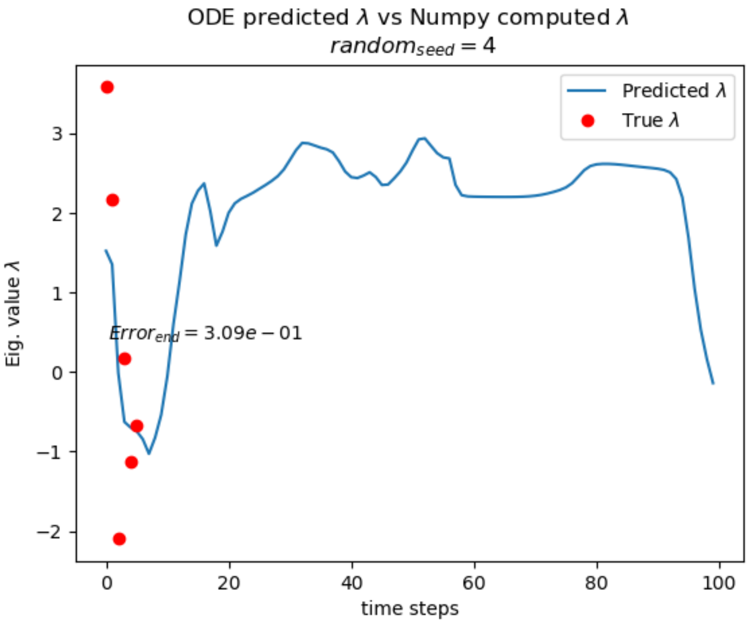
In the first plot with fewer time steps we see an okay end error and the predictions seem to visit every λ_{true} atleast once. For more time steps in the second plot it gets a little more chaotic and most of the peeks seem to be somewhat close to the true eigenvalues.

Looking at 6×6 matrices:









Not really any convergence, but again the peaks goes towards different eigenvalues. Could be interesting spending some more time on this and improve. This approach on eigenvalues was motivated by the paper by Yi et al (Reference 2) where they made a very successful model for finding the largest/smallest eigenvalue with a efficient recurrent neural network.

3.4 Unit test of HeatLearner class

A simple test was set up with $u_x = cos(x)$ and $u(0,0) = 0$.
We test for $MSE[u(x,0), \quad cos(x)] < tol$ for some tolerance. (*See unit_test.py*)

4 Summary (Part e)

The findings for the heat equation were overall that DNN's are more flexible and can be much faster on larger grids which can make it a very important part of solving large scale real world problems. For the finite-differences it was very precise on the 1d heat eq. but it is not flexible and scales badly for larger problems.

Finite-difference worked very well error wise. Cheapest model had $MSE_{total} = 1.16 \cdot 10^{-6}$. The run time is $O(\Delta x^2)$.

With the DNN model we trained it on a small 10×10 grid and managed errors MSE_{total} of magnitude 10^{-3} for grids 100×100 and up to much larger ones. The efficiency gains lays in minimizing the sub-sets for training as well as having good enough precision.

Our implementation of the eigenvalue solver had interesting but poorly results. We managed sometimes absolute errors of magnitude 10^{-2} to 10^{-4}. In Yi et al's paper (Ref. 2) they had pretty good results with symmetrical matrices with ability to achieve high computing performance since their recurrent model could parallel process asynchronously. In section 5 they did simulations on different random symmetrical matrices and found the biggest/smallest eigenvalues efficiently with precision 10^{-4} and stable convergence.

References

1. Lecture note on ODE in Tensorflow: https://github.com/krisbhei/DEnet/blob/master/DNN_Diffeq/example_ode_poisson.ipynb

2. Paper on Eigenvalues with Neural Networks (Yi, Fu and Jin Tang): https://www.sciencedirect.com/science/article/pii/S0898122104901101

3. Deep Neural Networks Motivated by Partial Differential Equations by Lars Ruthotto and Eldad Haber https://www.researchgate.net/publication/324492734_Deep_Neural_Networks_Motivated_by_Partial_Differential_Equations

4. Adam optimizer https://towardsdatascience.com/adam-latest-trends-in-deep-learning