UiO : Department of Physics
University of Oslo

FYS-STK4155: Applied Data Analysis and Machine Learning

Project 3

Solving Differential Equations with Feed-Forward Neural Network Models

Kristian Wold Nicolai Haug Tobias Netskar

December 19, 2019

Abstract

The aim of this project is to design feed-forward neural network (FFNN) models suitable for solving differential equations. Traditional numerical methods and closed form solutions, when present, have been used to assess the efficiency and accuracy of the FFNN models.

The first FFNN model is employed to learn the initial-boundary problem given by the heat equation given in one spatial dimension. The model managed to produce accurate results training on grid points consistent with the Forward Euler stability criterion. It also managed to produce accurate results on much sparser grids, where Forward Euler scheme would have failed. The mean error was 0.00351 and 0.00395, respectively. However, FFNN used 19 seconds to train, four magnitudes of order longer than Forward Euler.

A FFNN was also employed to learn the solution to the nonlinear, coupled ordinary differential equation (ODE), presented by Yi et. al in [1]. Given a real symmetric matrix A in the source term, the temporal dynamic described by this ODE has convergence properties to the largest eigenvalue. Simply replacing A with $-A$ yield the smallest eigenvalue. Our FFNN succeeded in computing both the largest and smallest eigenvalue to an accuracy of order 10^{-12} and 10^{-5} for some benchmark 3×3 and 6×6 real symmetric matrices, respectively. However, the FFNN model proved to not be as fast and accurate as Euler's method for solving this particular ODE. Also, the model relied on several hyperparameters, which proved hard to tune without knowing the solution beforehand and generalized poorly to bigger matrices.

Contents

1 Introduction

Differential equations have a wide range of applications in the natural sciences, and many methods have been developed for solving them. In numerical analysis, finite difference methods (FDMs) are widespread as they reduce the differential equation to a system of algebraic equations making the problem of finding the solution ideal to modern computers. The aim of this project is to design feed-forward neural network (FFNN) models suitable for solving ordinary, partial and coupled differential equations.

The rationale for proposing an approach with FFNN models is justified by the universal approximation theorem, which states that feed-forward neural networks, when given appropriate parameters, can approximate continuous functions on compact subsets of $\mathbb{R}^n$. This includes the solution of many differential equations. Neural networks are uniquely suited in optimisation problems, and can be trained by various methods such as the many flavors of gradient descent algorithms. By reformulating the differential equation to an equation where minimization of some parameters must be done, a neural net might be able to solve the problem. To that end, a possible method is to formulate a trial solution involving the result from a neural network, and construct a cost function which contains information about derivatives of the network.

The first FFNN model is employed within the TensorFlow framework to learn the initial-boundary problem given by the heat equation, a partial differential equation (PDE) with both temporal and spatial dependence. The problem formulation is scaled to the standard unity interval (0,1) in one spatial dimension. An explicit numerical scheme using Forward Euler will be used to assess the efficiency and accuracy of the FFNN model, alongside the exact solution of the problem.

The second FFNN model is employed, also within the TensorFlow framework, to learn the solution to the nonlinear, coupled ordinary differential equation (ODE) presented by Yi et. al in the article from Computers and Mathematics with Applications 47, 1155 (2004), which describes the state of a continuous-time recurrent neural network (CTRNN) model. Given a real symmetric matrix A in the source term, the temporal dynamic described by this ODE has convergence properties to an eigenvector corresponding to the largest eigenvalue λ, given that the initial vector $\mathbf{x}_0$ is not orthogonal to the eigenspace of λ. If $\mathbf{x}_0$ is not orthogonal to the eigenspace of the smallest eigenvalue σ, replacing A with $-A$ yields an eigenvector corresponding to σ. The eigenvalue itself is calculated with the Rayleigh quotient. The aim is to check if the constructed FFNN succeeds in computing both the largest and smallest eigenvalue for some benchmark 3×3 and 6×6 real symmetric matrices. We will also try choosing $\mathbf{x}_0$ orthogonal to the eigenspace corresponding to the largest eigenvalue, to make the network converge to an eigenvalue different from the largest. This starting point will itself be an eigenvector. In order to assess the FFNN model, we will compare the result with those from Euler's method for solving the same ODE and Numpy's linalg.eig which directly computes the eigenvalues of the matrix A.

This project is structured by first presenting a theoretical overview of the heat equation and the eigenvalue problem, as well as the aforementioned statistical learning methods in Section 2. This is followed by a presentation on the approach to study the various computations of interest in Section 3. Next, the results of the implementation are presented and discussed in Section 4, before subsequently they are concluded upon in Section 5. Lastly, an outline of possible continuations of the models, with respect to the implementation, are presented in Section 6.

2 Theory

2.1 The Heat Equation

In the natural sciences, problems with variables constrained by boundary conditions and initial values are often modelled by differential equations. The heat equation, which is a partial differential equation and a special case of the diffusion equation, describes the heat propagation in the evolution of time as it is transferred from regions of higher temperature to regions of lower temperature in a solid medium [2]. The general heat equation reads

$$\dot{u} = \alpha \nabla^2 u \tag{2.1}$$

where α is the diffusion coefficient which is a measure of the rate of transfer of heat with dimension length^2/time.

Typical diffusion problems may experience rapid change at the outset, but then the evolution of u converges to a stationary solution $\bar{u}$ as $t \to \infty$. In this limit $\dot{u} = 0$, and $\bar{u}$ is governed by $\nabla^2 \bar{u} = 0$, which is the Laplace equation. It is possible to solve for u using an explicit numerical scheme. Since the solution u of the heat equation is smooth and converges to a stationary state, small time steps are not convenient nor required by accuracy [2].

In this project, the physical problem is that of the temperature gradient in a rod of length L in one spatial dimension. The temperature distribution in the rod at $t = 0$ is given by the initial condition

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

By imposing Dirichlet boundary conditions, the temperatures at the endpoints of the rod, $u(0, t)$ and $u(L, t)$, are prescribed at all time. Physically, this corresponds to a system where a heat source keep the temperature constant at the endpoints. We shall use the simplest boundary condition: $u = 0$. The complete initial-boundary value heat diffusion problem in one spatial dimension can then be specified as

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \qquad x \in (0, L), t \in (0, T], \tag{2.2}$$

$$u(x, 0) = \sin(\pi x), \qquad x \in (0, L), \tag{2.3}$$

$$u(0, t) = 0, \qquad t \geq 0, \tag{2.4}$$

$$u(L, t) = 0, \qquad t \geq 0 \tag{2.5}$$

In this project, we will consider a version of the heat equation where any varying parameters are scaled away and the spatial interval is transformed to the standard unity interval $[0, 1]$. Assuming that the scaled parameters are denoted by a hat, we will for the sake of notational simplicity replace $\hat{x} \to x$ and $\hat{t} \to t$. The heat equation problem then reads

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \qquad x \in (0, 1), t \in (0, T], \tag{2.6}$$

$$u(x, 0) = \sin(\pi x), \qquad x \in (0, 1), \tag{2.7}$$

$$u(0, t) = 0, \qquad t \geq 0, \tag{2.8}$$

$$u(1, t) = 0, \qquad t \geq 0 \tag{2.9}$$

2.2 Closed Form Solution of the Heat Equation

Assume that the solution has the form

$$u(x,t) = f(x)g(t).$$

By the initial conditions, we get that

$$f(x) = k^{-1}\sin(\pi x),$$

where $k = g(0)$. Then, from Equation (2.6),

$$f''(x)g(t) = f(x)g'(t) \quad \Leftrightarrow \quad \frac{f''(x)}{f(x)} = \frac{g'(t)}{g(t)}.$$

Plugging in for $f(x)$ and $f''(x) = -\pi^2 a^{-1}\sin(\pi x)$, we find that

$$-\pi^2 = \frac{g'(t)}{g(t)}.$$

We integrate both sides with respect to t to obtain

$$-\pi^2 t + C = \int \frac{dg}{dt}\frac{1}{g}dt = \int \frac{1}{g}dg = \log|g|.$$

Hence

$$g(t) = D\mathrm{e}^{-\pi^2 t},$$

where D is some constant. Since $g(0) = D$, we have $D = k$. We get that

$$u(x,t) = g(t)f(x) = \mathrm{e}^{-\pi^2 t}\sin(\pi x),$$

which can be checked to be a solution of Equation (2.6) satisfying the initial and boundary conditions. Next we show that this solution is unique.

Suppose that that u_1 and u_2 are solutions. Define $v(x,t) = u_1(x,t) - u_2(x,t)$ and let

$$w(t) = \frac{1}{2}\int_0^1 v(x,t)^2\,\mathrm{d}x, \quad t \geq 0.$$

Note that by Equation (2.6),

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

by the initial condition $w(0) = 0$, and by the boundary condition $v(0,t) = v(1,t) = 0$ for $t > 0$.

By Leibniz' rule [3, 8.11.2], we can differentiate under the integral sign, so that

$$w'(t) = \int_0^1 v(x,t)\frac{\partial v(x,t)}{\partial t}\,\mathrm{d}x = \int_0^1 v(x,t)\frac{\partial^2 v(x,t)}{\partial x^2}\,\mathrm{d}x.$$

Then integration by parts yields

$$w'(t) = v(x,t)\frac{\partial v(x,t)}{\partial x}\bigg|_{x=0}^{1} - \int_0^1 \left(\frac{\partial v(x,t)}{\partial x}\right)^2 \mathrm{d}x = -\int_0^1 \left(\frac{\partial v(x,t)}{\partial x}\right)^2 \mathrm{d}x,$$

implying that $w'(t) \leq 0$ for all $t > 0$. On the other hand, since $w(t) \geq 0$ for all $t > 0$ and $w(0) = 0$, we must have $w'(t) \geq 0$. It follows that $w'(t) = 0, t > 0$, and thus $w(t) = 0$ for all $t \geq 0$. We conclude that $u_1 = u_2$.

2.3 Explicit Numerical Scheme Using Forward Euler

This section is based on the procedure described in [2]. We have here opted for deriving a scheme for the non-scaled PDE to obtain more flexibility in the numerical model.

In order to derive the discrete set of equations to be solved, the domain $[0, L] \times [0, T]$ is discretized by replacing both the spatial and temporal domains by a set of uniform mesh points

$$x_i = i\Delta x, \quad i = 0, ..., N_x$$

and

$$t_n = n\Delta t, \quad n = 0, ..., N_t$$

The mesh function, denoted u_i^n, then approximates the exact solution $u(x_i, t_n)$ for $i = 0, ..., N_x$ and $n = 0, ..., N_t$. Requiring the heat PDE in Equation 2.2 to be satisfied at each mesh point (x_i, t_n), $i = 0, ..., N_x$ and $n = 0, ..., N_t$, leads to the equation

$$\frac{\partial}{\partial t}u(x_i, t_n) = \alpha\frac{\partial^2}{\partial x^2}u(x_i, t_n)$$

Replacing the first-order derivative with a forward difference approximation in time and the second-order derivative with a central difference approximation in space yields

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

which is the PDE turned into algebraic equations. We assume that the only unknown quantity is u_i^{n+1}. Solving with respect to this unknown gives

$$u_i^{n+1} = u_i^n + F\Big(u_{i+1}^n - 2u_i^n + u_{i-1}^n\Big), \tag{2.10}$$

where $F = \alpha\Delta t/\Delta x^2$ is a dimensionless number. The stability criterion for the explicit scheme requires that $F \leq 1/2$.

2.4 Solving Differential Equations With Neural Networks

For a primer on feed-forward neural networks, we refer to the project [4] by the authors where the basics are covered.

Definition 2.1 (Squashing function [5, Definition 2.3])**.** A *squashing function* is a function $\psi\colon \mathbb{R} \to [0, 1]$ that is monotonely increasing and with $\lim_{t\to\infty}\psi(t) = 1$ and $\lim_{t\to-\infty} = 0$.

Denote by $\Sigma^n(\psi)$ the set of all output functions for single hidden layer feed-forward networks with n real-valued inputs, having ψ as squashing function at the hidden layer, and no squashing at the output. That is, $\Sigma^n(\psi)$ is the collection of functions $\phi\colon \mathbb{R}^n \to \mathbb{R}$ of the form

$$\phi(x) = \sum_{i=1}^{m}\beta_i\psi(w_i^T x + b_i), \quad x, w_i \in \mathbb{R}^n, \beta_i, b_i \in \mathbb{R}, m \in \mathbb{N}.$$

Given a subspace $K \subseteq \mathbb{R}^n$, let $C(K)$ be the set of all continuous functions from $K \subseteq \mathbb{R}^n$ into $\mathbb{R}$.

Theorem 2.2 (Universal Approximation Theorem [5, Theorem 2.4])**.** *For any squashing function ψ and any $n \in \mathbb{N}$ and $K \subseteq \mathbb{R}^n$ compact, the set $\Sigma^n(\psi)$ is uniformly dense in $C(K)$.*

From Theorem 2.2 it follows that we can approximate the solutions of differential equations on compact subsets of $\mathbb{R}^n$ with neural networks.

In the following, suppose for simplicity that f is a function from a compact interval $[t_0, t_1]$ of $\mathbb{R}$ into $\mathbb{R}$. A similar discussion applies to f with range $\mathbb{R}^n$ and domain in $\mathbb{R}^n$ with any $n, m \in \mathbb{N}$. Let f be given by a differential equation

$$G(t, f'(t), f''(t), \ldots, f^{(k)}(t)) = 0$$

for some $k \in \mathbb{N}$ and all t in the interior of K. Suppose further that $f(t_0) = y_0$. To (approximately) solve this using a neural network N, the solution g will be of the form

$$g(t) = h(t) + F(t)N(t, P), \quad t \in K.$$

Here $N(x, P)$ denotes the output of the neural network at input t with weights and biases contained in P. The functions h and F are chosen such that h satisfies the initial condition (i.e. $h(t_0) = y_0$), and $F(t)$ is zero for the initial t (i.e. $F(t_0) = 0$). This ensures that the solution g satisfies the initial condition.

To find the weights P, we choose some points $t_1, \ldots, t_N \in K$ and minimize the squared sum

$$\sum_{i=0}^{N} (G(t_i, g'(t_i), g''(t_i), \ldots g^{(k)}(t_i)))^2.$$

2.5 Finding Eigenvalues with Differential Equations

Let $A \in \mathbb{R}^{n \times n}$ be a real symmetric matrix. Define $f \colon \mathbb{R}^n \to \mathbb{R}^n$ by

$$f(x) = [x^T x A + (1 - x^T A x) I] x, \quad x \in \mathbb{R}^n, \tag{2.11}$$

where $I \in \mathbb{R}^{n \times n}$ is the identity matrix. Let $x \colon \mathbb{R} \to \mathbb{R}^n$ be a map that satisfies

$$Dx(t) = -x(t) + f(x(t)). \tag{2.12}$$

We then have the following theorem [1]:

Theorem 2.3. *For each solution $x \colon \mathbb{R} \to \mathbb{R}^n$ of Equation 2.12, the limit $\lim_{t \to \infty} x(t)$ exists and converges to an eigenvector of A.*

If λ is the largest eigenvalue of A and the starting point $x(0)$ is not orthogonal to the eigenspace of λ, then $\lim_{t \to \infty} x(t)$ is an eigenvector of A with eigenvalue λ.

Replacing A with $-A$, then if $x(0)$ is not orthogonal to the eigenspace of the smallest eigenvalue σ of A, the limit $\lim_{t \to \infty} x(t)$ converges to an eigenvector corresponding to σ.

Definition 2.4 (Rayleigh quotient [6, p. 234])**.** Given a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$, the *Rayleigh quotient* is defined as

$$r(A, x) = \frac{x^T A x}{x^T x}.$$

Note that if x is an eigenvector of A with eigenvalue λ, then

$$r(A, x) = \frac{x^T A x}{x^T x} = \frac{x^T (\lambda x)}{x^T x} = \lambda.$$

3 Method

3.1 The Heat Equation Problem

Following the discussion in Section 2.3, the computational algorithm [2] for the Forward Euler explicit scheme becomes:

1. Compute $u_i^0 = I(x)$ for $i = 0, ..., N_x$
2. for $n = 0, 1, ..., N_t$:
   (a) apply Equation 2.10 for all the internal spatial points $i = 1, ..., N_x - 1$
   (b) set the boundary values $u_i^{n+1} = 0$ for $i = 0$ and $i = N_x$.

The approach for solving differential equations with FFNN models as described in Section 2.4, require us to reformulate the heat equation, given by Equation 2.6, into an equation a neural network can solve. A possible method we will utilize, is to formulate a trial solution involving the result from the network and construct a cost function which contains information about derivatives of the network.

If $N(t, P)$ denotes the output of the network at input (x, t) with weights and biases contained in P, then our chosen trial function g is given by

$$g(x,t) = (1-t) \cdot g_0 + x(1-x) \cdot t \cdot N((x,t), P),$$

where g_0 is the initial condition given by Equation 2.7. The rationale behind this trial solution is that the first term alone makes g satisfy the given set of conditions and the second term ensure that the output from $N((x,t), P)$ is zero when g is evaluated at the values of x where the given conditions must be satisfied.

We want to find the weights and biases P such that g is close to the solution of Equation 2.6 at all points in the combined spatial and temporal domain $[0, 1] \times [0, T]$, where T is the final simulation time. For this we choose points $0 = x_0 < x_1 < \ldots < N_x = 1$ and $0 = t_0 < t_1 < \ldots < t_N = T$ and employ the Adam optimization algorithm to minimize the cost function. We choose to consider the mean squared error as the cost function for an input (x, t), and the fact that the right-hand side subtracted by the left-hand side of the heat PDE should be equal to zero.

The FFNN model is employed within the **`TensorFlow`** framework, with two hidden layers. The first layer consist of 150 neurons, and the tanh function

$$f(z) = \tanh z$$

is chosen to be the activation function for each hidden neuron. The second layer consist of 50 neurons, and here the sigmoid function

$$f(z) = \frac{1}{1 + e^{-z}}$$

is chosen to be the activation function for each hidden neuron. The neural network is depicted in Figure 3.1

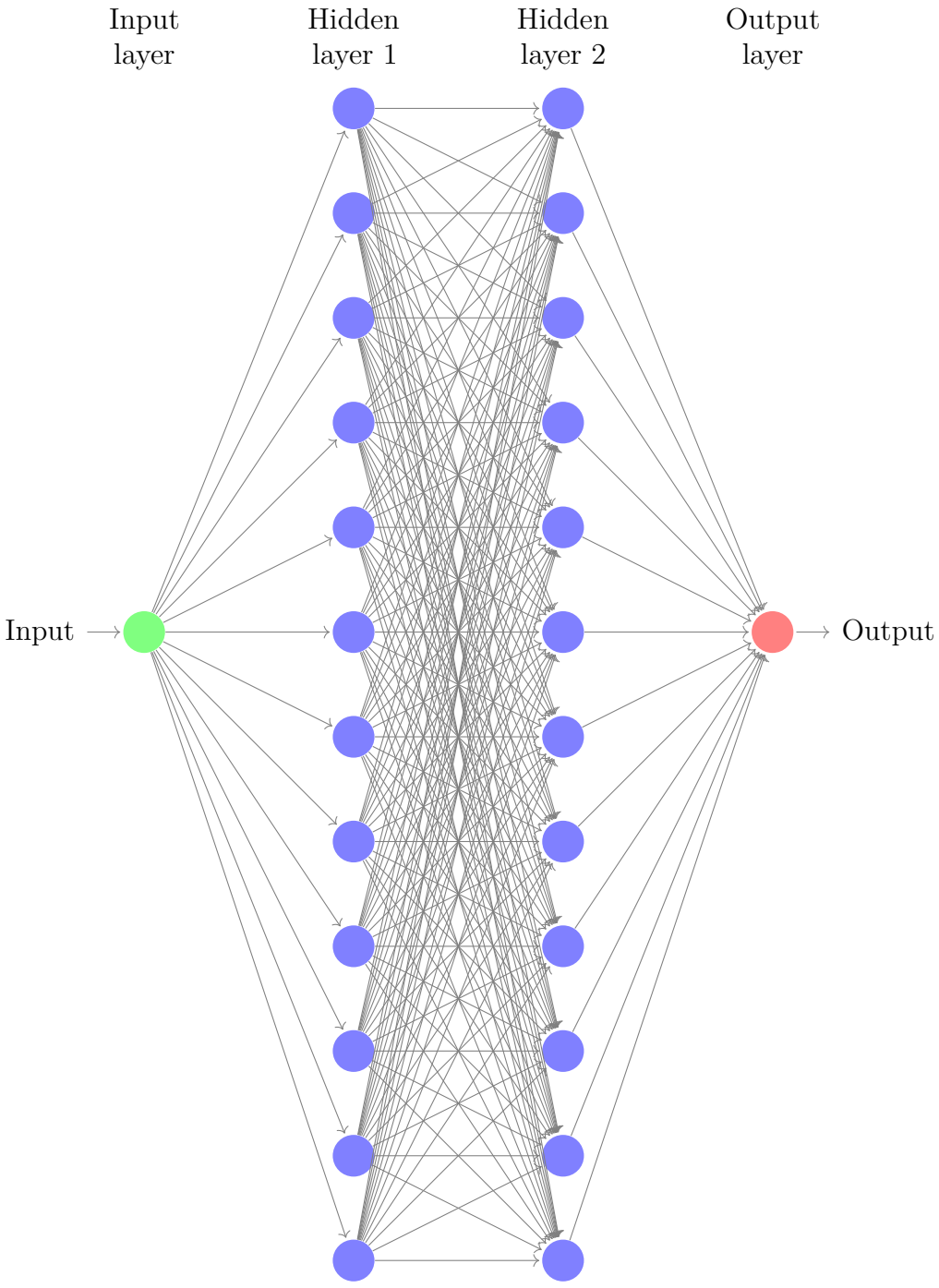


Figure 3.1: *The feed-forward neural network of choice for the heat equation problem. Hidden layer 1 consists of* 150 *nodes with the tanh activation function and hidden layer 2 consists of* 50 *nodes with the sigmoid activation function.*

We train on two different sets of data to investigate the performance of the FFNN model. Both sets of data contain $N_x = 11$ points in the spatial dimension. The first set contains $N_t = 199$ points in the temporal dimension, matching the grid Forward Euler required because of the stability criterion.

The second data set is much sparser in the temporal dimension with $N_t = 11$ points, to if the FFNN model can obtain a sensible solution in a setting Forward Euler would be useless.

3.2 The Eigenvalue Problem

Theorem 2.3 gives a connection between solving differential equations and finding the largest and smallest eigenvalues of real symmetric matrices. We will explore this approach for finding eigenvalues by using the Euler method and the method for solving differential equations with FFNNs as described in Section 2.4.

Let the real symmetric matrix $A \in \mathbb{R}^{n\times n}$ be given. Our FFNN is set up with one node in the input layer, three hidden layers, and n nodes in the output layer. The hidden layer next to the input has 100 nodes, the one in the middle has 50 nodes, and the last one has 25 nodes.

We define the function f in Equation 2.11 with A or with A replaced by $-A$, depending on whether we want to find the largest or the smallest eigenvalue of A.

If $N(t, P)$ denotes the output of the network at input t with weights and biases contained in P, then our chosen trial function g is given by

$$g(t) = \mathrm{e}^{-t}x_0 + (1 - \mathrm{e}^{-t})N(t, P),$$

where $x_0 \in \mathbb{R}^n$ is a chosen starting point of g. In the case where we want to find the largest eigenvalue λ of A, x_0 has to be not orthogonal to the eigenspace corresponding to λ, and in the case where we want to find the smallest, say σ, the vector x_0 has to be not orthogonal to the eigenspace corresponding to σ.

We want to find the weights and biases P such that g is close to the solution of Equation 2.12 at all points in an interval $[0, T]$, where T is some fixed positive number. For this we choose points $0 = t_0 < t_1 < \ldots < t_N = T$ and employ the Adam optimization algorithm to minimize the sum of squares

$$\sum_{i=0}^{N}(Dx(t_i) + x(t_i) - f(x(t_I)))^T(Dx(t_i) + x(t_i) - f(x(t_I))).$$

Finally, we use, by Theorem 2.3, the vector $g(T)$ as an approximation to an eigenvector corresponding to the largest eigenvector λ. The value of λ is then approximated by calculating the the Rayleigh quotient of $g(T)$, as given in Definition 2.4.

The network of choice in the case of $n = 6$ is depicted in Figure 3.2.

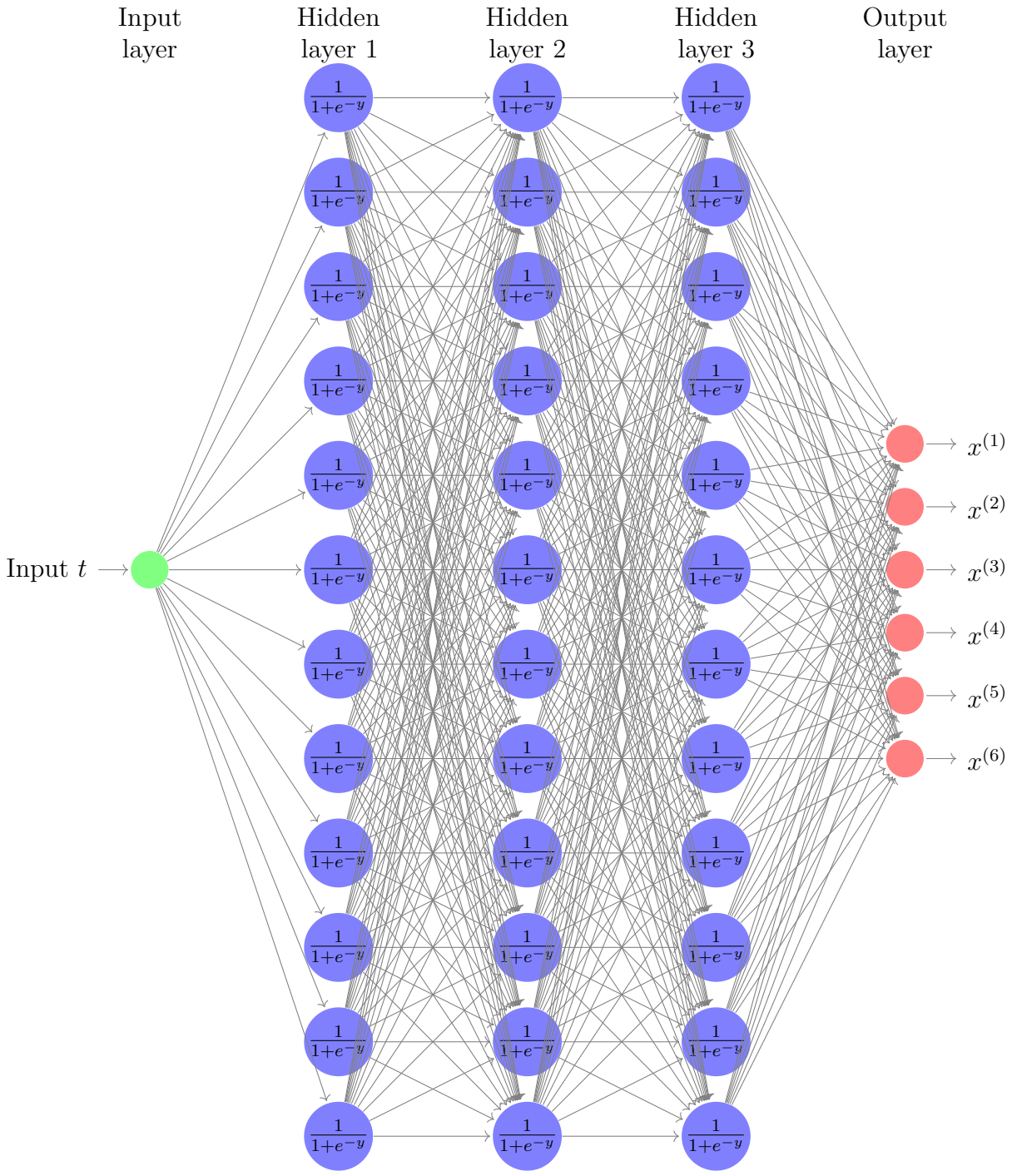


Figure 3.2: *The feed-forward neural network of choice in the case of $n = 6$. Hidden layer 1 consists of 100 nodes, hidden layer 2 consists of 50 nodes, and hidden layer 3 consists of 25 nodes. Sigmoid is the activation function for all hidden layers.*

The FFNN model is employed within the `TensorFlow` framework. Note that in the code the function call `tf.GradientTape().gradient(y, x)` calculates the gradient of the sum of the components in `y` with respect to the variables in `x`.

We do a series of benchmark runs to check how well this neural network approach performs in regard to computing the largest and smallest eigenvalue for some 3×3 and 6×6 real symmetric matrices. We will also check the convergence of the network when $\mathbf{x}_0$ is chosen to be orthogonal to the eigenvector corresponding to the largest eigenvalue. In order to assess

the performance, we will compare the result with those from Euler's method for solving the same ODE. We also compare the approximative eigenvalues and eigenvectors with those from `Numpy's linalg.eig`.

For each benchmark problem, we decided on 2000 epoch for the Adam optimization algorithm. We always normalize the starting point x_0, as this made the convergence of the weights and biases of the network easier. Note that by the proof of [1, Theorem 2], the norm of $x(t), t \geq 0$ is constant.

3.2.1 Benchmark Problem 1

In this run we use the 3×3 matrix

$$A = \begin{pmatrix} 3 & 2 & 4 \\ 2 & 0 & 2 \\ 4 & 2 & 3 \end{pmatrix}, \tag{3.1}$$

with eigenvalues $\lambda_1 = 8$ and $\lambda_2 = \lambda_3 = -1$. The initial point is $x_0 = (1, 0, 0)$. We let $T = 1$ and train the neural network on 5 evenly spaced timepoints in the interval $[0, T]$.

The Euler method is run with 10 000 time points, just to give a good approximation of the actual solution $x(t)$, for comparison with the neural network solution.

3.2.2 Benchmark Problem 2

In this run we use the matrix

$$A = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \tag{3.2}$$

with eigenvalues $\lambda_1 = 2 + \sqrt{2}$, $\lambda_2 = 2$ and $\lambda_3 = 2 - \sqrt{2}$.

We solve the differential equation for $-A$ to get an approximation of the smallest eigenvalue of A. We use $T = 3.5$ as the end point of the domain of the solution.

3.2.3 Benchmark Problem 3

This time we let

$$A = \begin{pmatrix} 3 & 1 & -1 \\ 1 & 3 & -1 \\ -1 & -1 & 5 \end{pmatrix}, \tag{3.3}$$

which has eigenvalues $\lambda_1 = 6$, $\lambda_2 = 2$ and $\lambda_3 = 3$.

Here we choose initial point $x_0 = (-1, 1, 0)$, which is orthogonal to the eigenspace corresponding to the eigevalues λ_1 and λ_3 of A. We therefore expect $g(T)$ to approximate an eigenvector corresponding to the smallest eigenvalue. Note that since all the eigenvectors of A are orthogonal, x_0 is itself an eigenvector, namely one corresponding to λ_2. We let $T = 1$.

3.2.4 Benchmark Problem 4

We now move on from 3×3-matrices to a 6×6-matrix. We use the randomly generated matrix A which when rounded to four decimal places looks like

$$A = \begin{pmatrix} 0.4967 & 0.7205 & 0.4448 & 0.3075 & -0.3893 & -0.4179 \\ 0.4448 & -1.1914 & -1.7249 & 0.4517 & -1.0819 & 0.1503 \\ 0.3075 & -0.4349 & 0.4517 & -0.2258 & 0.2216 & -1.2412 \\ -0.3893 & -0.1762 & -1.0819 & 0.2216 & -0.6006 & 0.2654 \\ -0.4179 & 0.6933 & 0.1504 & -1.2412 & 0.2654 & -1.2208 \end{pmatrix}, \tag{3.4}$$

and has eigenvalues $\lambda_1 = -3.13078844$, $\lambda_2 = 1.82431291$, $\lambda_3 = 1.21134579$ $\lambda_4 = -0.81694732$, $\lambda_5 = 0.15215664$ and $\lambda_6 = -1.74810717$. The initial point is randomly chosen as

$$x_0 = (0.20819531, 0.34735924, 0.3651539, 0.12346636, 0.6475964, 0.51771987).$$

We solve the differential equation with A and approximate the largest eigenvalue. We do this for different number of time steps $N_t = 5, 11, 51, 101$. The end point of the domain $[0, T]$ of the solution is $T = 10$ for each run.

Here we also want to assess the performance of Euler, and hence we let the number of time steps we use for this method also vary. The number of time steps we consider are $N = 51, 101, 501, 1001$.

4 Results and Discussion

4.1 Heat Equation

Accompanying notebook: heat_pde_with_fe_and_tf.ipynb.

4.1.1 Forward Euler

Figure 4.1 presents the analytical solution, the approximate solution found with Forward Euler, and an error surface describing the difference between the two.

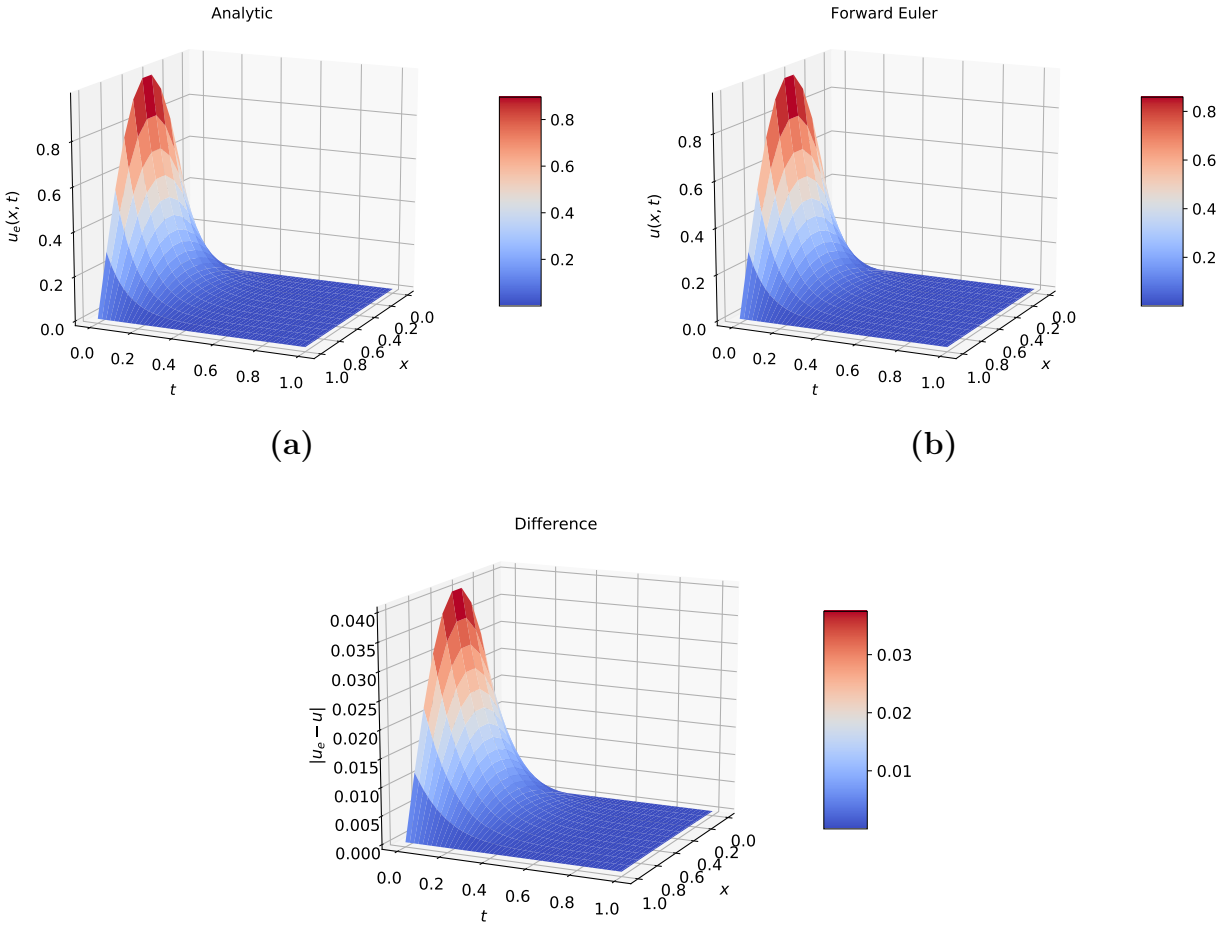


Figure 4.1: *Comparison of analytical solution* ***(a)*** *and solution obtained with Forward Euler* ***(b)*** *of the heat equation. The Forward Euler solution was found using* $Nx = 11$ *spacial points and* $Nt = 199$ *temporal points. The error surface* ***(c)*** *shows the difference between the analytical and approximate solution.*

From Figure 4.1, we see that Forward Euler succeeds in establishing a good approximation for the solution of the heat equation, with the resulting errors presented in Table 4.1. From sub figure **(c)**, we see that the error surface have the same general shape as exact solution itself, although with a much smaller amplitude as the approximation is close to the analytical. One could say that Forward Euler respects the structure of the problem, in the sense that

the discrepancies are produced in an orderly fashion that reflects the shape of the exact solution.

A benchmark of the CPU time of the Forward Euler scheme was carried out with the same resolution as above. The benchmark is based on 1000 simulations and resulted in an average runtime of $0.00122s$ as presented in Table 4.1.

Table 4.1: *Maximum and mean difference between the analytical solution and the numerical solution obtained with Forward Euler as shown in Figure 4.1 The mean CPU time of 1000 Forward Euler simulations is also listed. The CPU Ryzen 9 3900X was used in the CPU benchmark.*

Max Difference	Mean Difference	Mean CPU Time (seconds)
0.04009	0.00318	0.00122

Simulations with the Forward Euler scheme show that the time step restriction, $F \leq 1/2$, which means $\Delta t \leq \Delta x^2/(2\alpha)$, may be relevant in the beginning of the diffusion process, when the solution changes quite fast, but as time increases, the process slows down, and a small Δt may be inconvenient.

4.1.2 FFNN Trained on a Large Number of Temporal Points

Figure 4.2 presents the analytical solution, the approximated solution found with FFNN and an error surface describing the difference between the two. The FFNN model was trained on the same set of grid points as the Forward Euler solution in Figure 4.1. For plotting, the FFNN model was used to interpolate on a much finer grid of 301 points in both space and time.

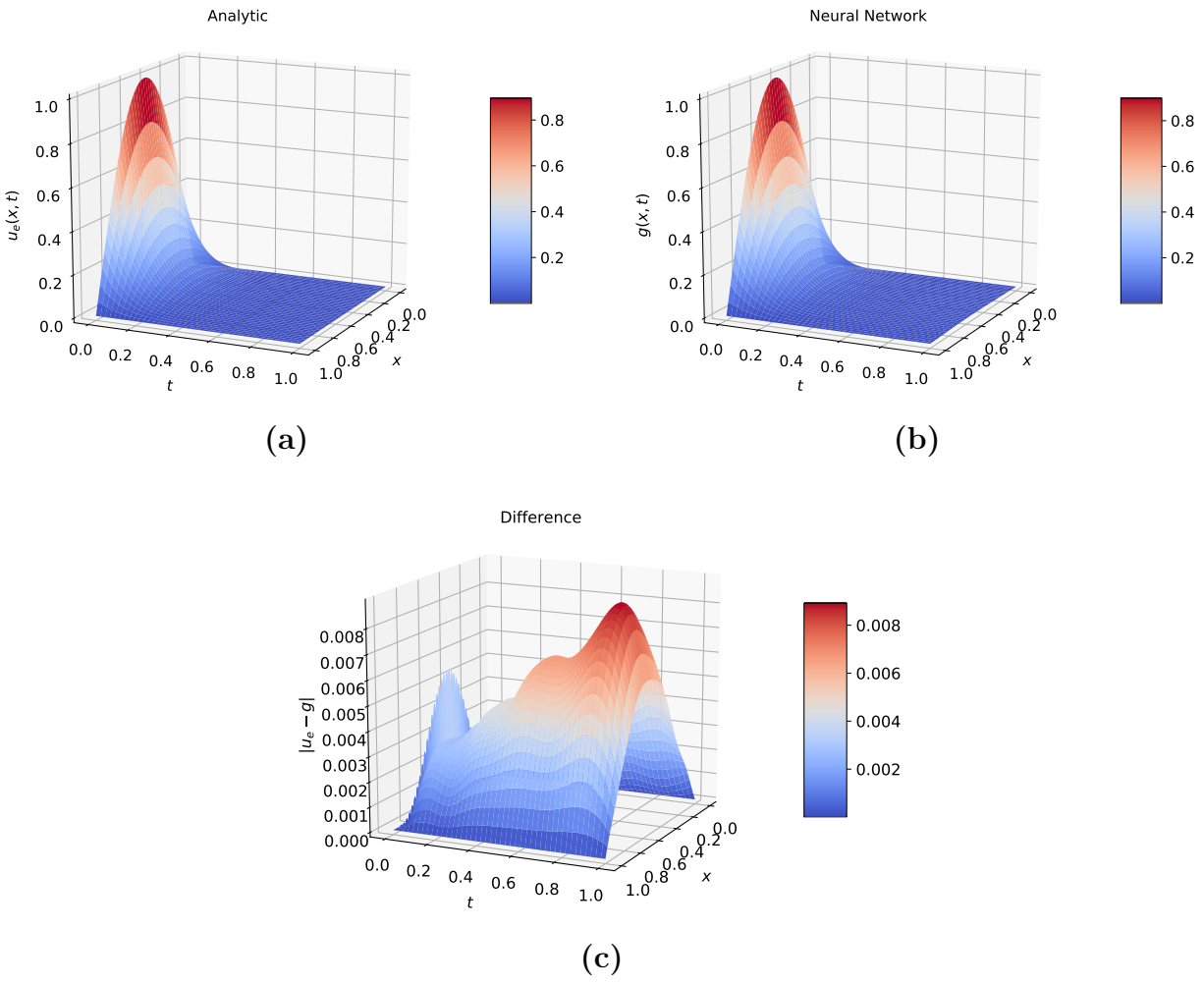


Figure 4.2: *Comparison of analytical solution* ***(a)*** *and solution obtained with FFNN* ***(b)*** *of the heat equation. FFNN solution was found using* $Nx = 11$ *spacial points and* $Nt = 199$ *temporal points. The error surface* ***(c)*** *shows the difference between the analytical and approximate solution.*

The FFNN model too succeeds in establishing a good approximation, as seen from figure Figure 4.2 and the accompanying error estimates in Table 4.2. The FFNN model, while being trained on the same grid points as Forward Euler, actually surpasses it in terms of accuracy. Without getting too optimistic, from Table 4.2, the CPU time clocked in at about 19 seconds, some 4 orders of magnitude longer than Forward Euler.

Further, the error surface of the FFNN model is quite different from Forward Euler. Though small in magnitude, it displays a more erratic behaviour and does not reflect the shape of the exact solution as Forward Euler does. This can be explained by the fact that neural networks are very complex functions of many parameters, and are thus very susceptible to "noise" cause by the repeated transformation through the layers.

Table 4.2: *Maximum and mean difference between the analytical solution and the numerical solution obtained with FFNN. The CPU time of training is also listed. The CPU Ryzen 9 3900X was used in the CPU benchmark.*

	Max Difference	Mean Difference	Spatial Points (#)	Temporal Points (#)	CPU Time (seconds)
Training	0.00895	0.00317	11	199	19.14199
Interpolation	0.00895	0.00351	301	301	

4.1.3 FFNN Trained on a Small Number of Temporal Points

Figure 4.3 presents the analytical solution, the approximated solution found with FFNN and an error surface describing the difference between the two. The FFNN model was trained on 11 points in both space and time. For plotting, the FFNN model was used to interpolate on a much finer grid of 301 points in both space and time.

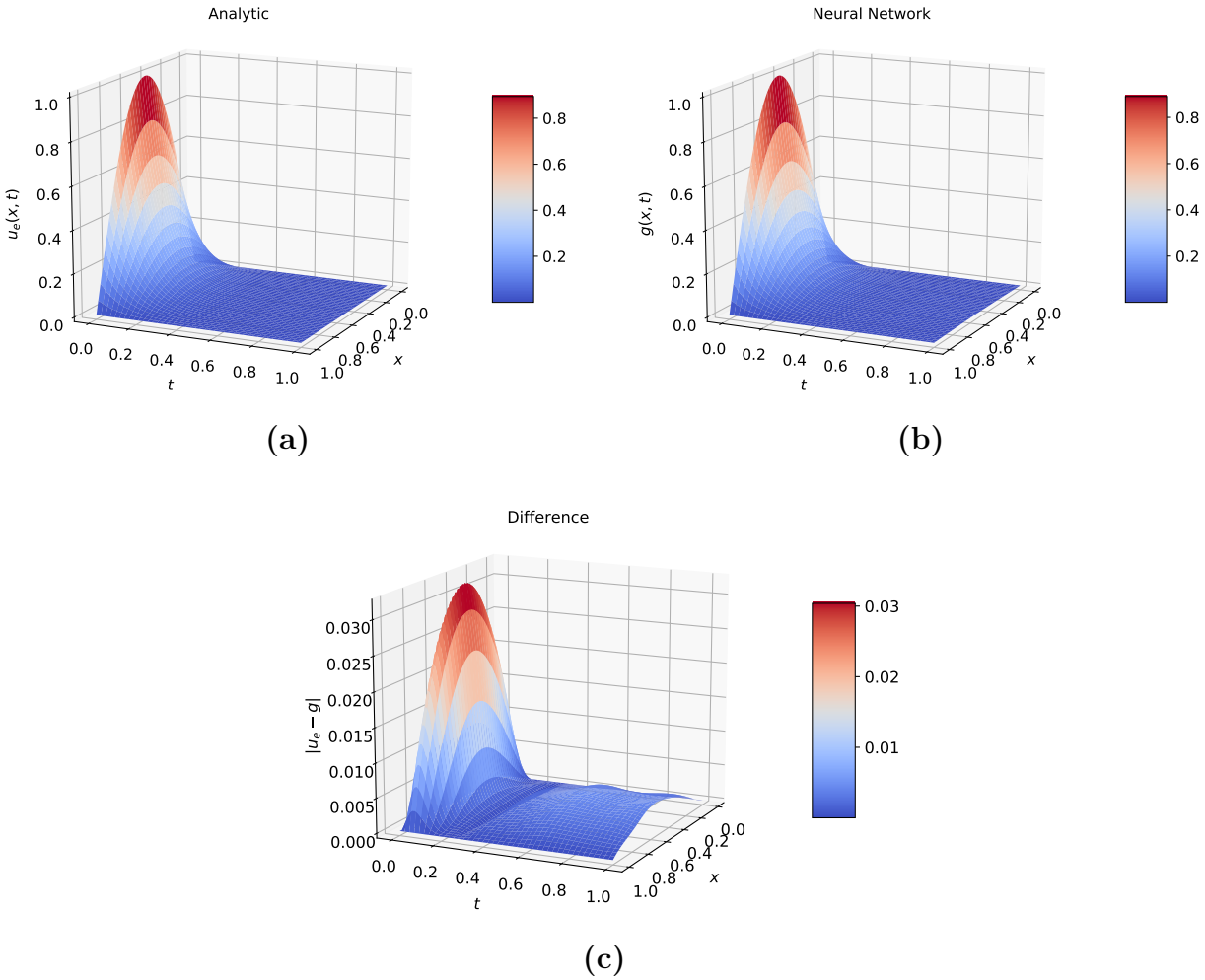


Figure 4.3: *Model 2, Plot solution on a larger grid than FFNN is trained on, i.e., points must be interpolated*

From Figure 4.3, we see that the FFNN model manages to approximate the solution quite well despite having only 11 points in both space and time to train on, when it formerly had

199 points in time. Having fewer points to train on, the network gets less information to base the approximation on. Naturally, the error is higher as can be seen by the amplitude of the error surface in Figure 4.3 and the error estimates in Table 4.3. However, it should be noted that the for such choice of spacial and temporal points, we are way outside Forward Eulers stability criterion, which would promptly produce a nonsense solution. The neural network does not display a similar criterion of stability.

The reduction of training points also sped up the simulation to 4.9 seconds as seen in Table 4.3. However, this is still magnitudes of order larger than Forward Euler when the criterion of stability was fulfilled.

Table 4.3: *Maximum and mean difference between the analytical solution and the numerical solution obtained with FFNN. The CPU time of training is also listed. The CPU Ryzen 9 3900X was used in the CPU benchmark.*

	Max Difference	Mean Difference	Spatial Points (#)	Temporal Points (#)	CPU Time (seconds)
Training	0.02696	0.00298	11	11	4.89815
Interpolation	0.03215	0.00395	301	301	

4.2 Eigenvalue Problem

Accompanying notebook: eigenvalue_tf.ipynb.

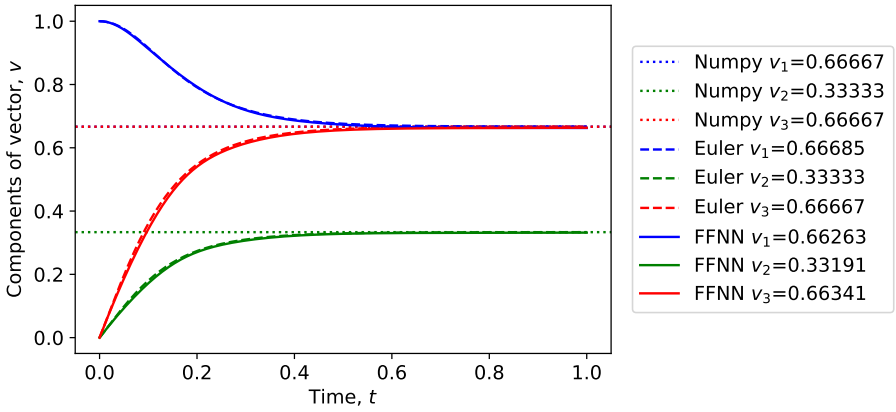
4.2.1 Benchmark Problem 1

Table 4.4 tabulates the problem and model parameters for Benchmark Problem 1 described in Section 3.2.1.

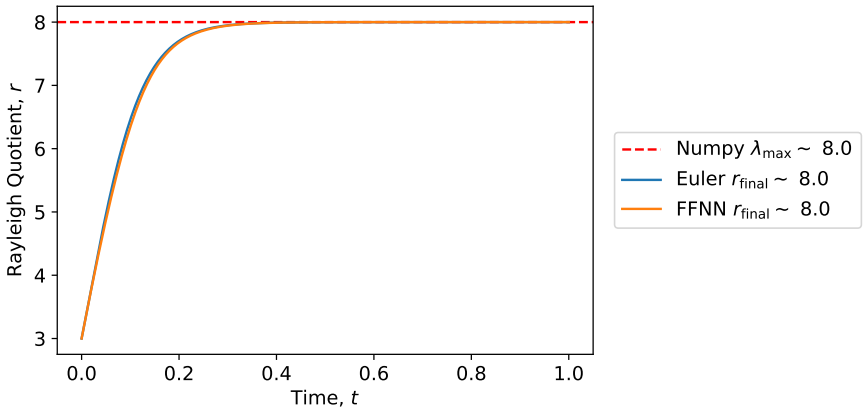
Table 4.4: *Problem and model parameters for Benchmark Problem 1.*

Parameter	Numerical Value
Initial vector	$\boldsymbol{x_0} = (1, 0, 0)$
Simulation time	$T = 1$
Number of time points (Euler)	10 000
Number of time points (FFNN)	11
Number of epochs (FFNN)	2 000

Figure 4.4 shows the results for Benchmark Problem 1. The computed Rayleigh quotients and the absolute error relative to the eigenvalue computed by Numpy are tabulated in Table 4.5.



(a)



(b)

Figure 4.4: *Results for Benchmark Problem 1 with a 3×3 real symmetric matrix A.* ***(a)*** *shows the components of the computed steady-state vector as a function of time. The dashed lines are the components computed by Euler's method and the solid lines are computed by the FFNN model. The dotted lines are the normalized eigenvector components corresponding to the largest eigenvalue computed directly from the matrix by Numpy's linalg.eig.* ***(b)*** *shows the computed Rayleigh quotients, r, as a function of time for both Euler's method and the FFNN model and the largest eigenvalue, $\lambda_{\max}$, of the matrix computed by Numpy's inalg.eig as well. The final Rayleigh quotients are rounded to 5 decimal points.*

Table 4.5: *The computed Rayleigh quotients at the final simulation time for both Euler's method and the FFNN model. The absolute error relative to the eigenvalue computed by Numpy is also listed.*

Method	Rayleigh Quotient	Absolute Error
Numpy	8.0	–
Euler	7.99999983	$1.697 \cdot 10^{-7}$
FFNN	7.99999721	$2.792 \cdot 10^{-6}$

The two solutions shown in Figure 4.4 follow each other closely, and already at $t = 1$ they give fairly accurate approximations of the eigenvector corresponding to $\lambda_{\max}$. Even though

the ODE is more complicated than the previous PDE, the sense that it is strongly coupled and non-linear, the network has no problem representing the solution. An additional layer was however necessary to capture increased complexity. The eigenvalue approximations are even more accurate than the eigenvector, although Euler also in the problem seems as the superior choise. As before, it is much faster to simulate, but also now it is better in terms of accuracy.

4.2.2 Benchmark Problem 2

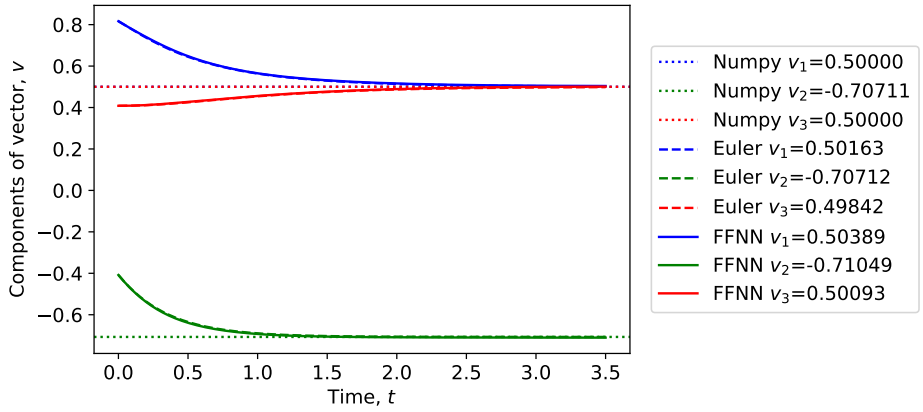
The problem and model parameters for Benchmark Problem 2 are described in Section 3.2.2. In this problem we find the smallest eigenvalue of A.

Table 4.6

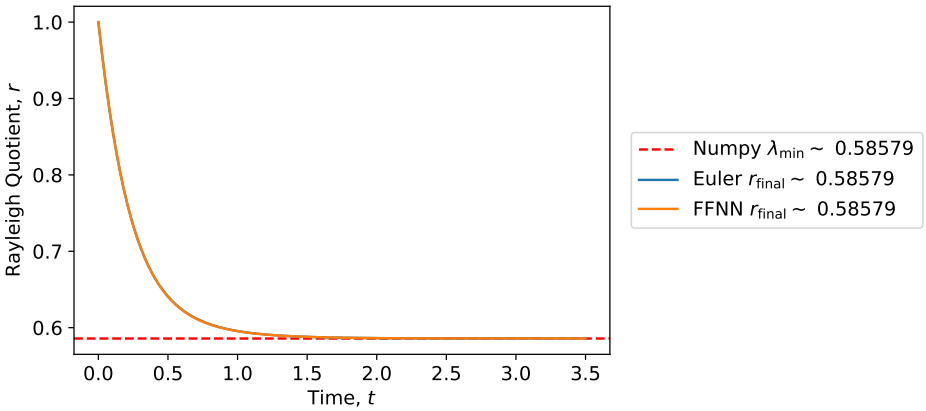
Table 4.6: *Problem and model parameters for Benchmark Problem 2.*

Parameter	Numerical Value
Initial vector	$\boldsymbol{x_0} = (0.8165, -0.4082, 0.4082)$
Simulation time	$T = 3.5$
Number of time points (Euler)	10 000
Number of time points (FFNN)	11
Number of epochs (FFNN)	2 000

Figure 4.5 shows the results. The computed Rayleigh quotients and the absolute error relative to the eigenvalue computed by Numpy are tabulated in Table 4.7.



(a)



(b)

Figure 4.5: *Results for Benchmark Problem 2 with a 3×3 real symmetric matrix A.* ***(a)*** *shows the components of the computed steady-state vector as a function of time. The dashed lines are the components computed by Euler's method and the solid lines are computed by the FFNN model. The dotted lines are the normalized eigenvector components corresponding to the largest eigenvalue computed directly from the matrix by Numpy's linalg.eig.* ***(b)*** *shows the computed Rayleigh quotients, r, as a function of time for both Euler's method and the FFNN model, and the largest eigenvalue $\lambda_{\max}$ of the matrix as computed by Numpy's linalg.eig as well. The final Rayleigh quotients are rounded to 5 decimal points.*

Table 4.7: *The computed Rayleigh quotients at the final simulation time for both Euler's method and the FFNN model. The absolute error relative to the eigenvalue computed by Numpy is also listed.*

Method	Rayleigh Quotient	Absolute Error
Numpy	0.58578644	–
Euler	0.58579371	$7.273 \cdot 10^{-6}$
FFNN	0.58579257	$6.129 \cdot 10^{-6}$

The convergence as $t \to \infty$ was slower this time than in benchmark problem 1, so a choice of $t = 1$ is not garanteed to generalize to produce reliable results. When increased to $t = 3.5$, the

Rayleigh quotients produced with both Euler and FFNN give an eigenvalue approximation that is accurate to five decimal places.

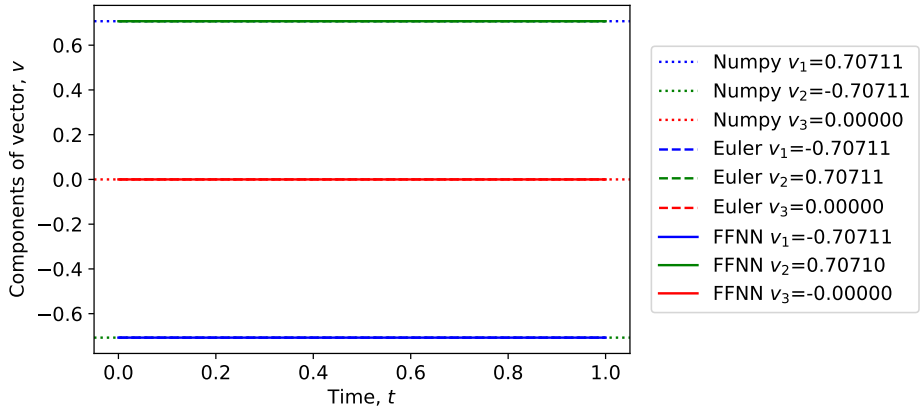
4.2.3 Benchmark Problem 3

Table 4.8 tabulates the problem and model parameters for Benchmark Problem 3 described in Section 3.2.3.

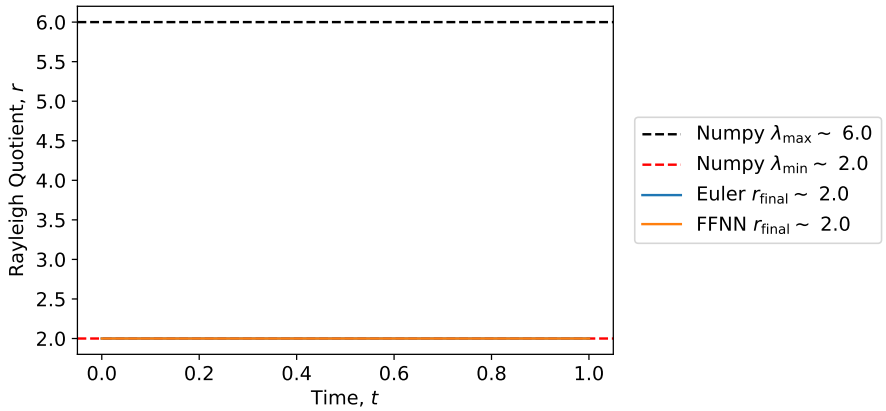
Table 4.8: *Problem and model parameters for Benchmark Problem 3.*

Parameter	Numerical Value
Initial vector	$\boldsymbol{x_0} = (-0.707, 0.707, 0)$
Simulation time	T=1
Number of time points (Euler)	10 000
Number of time points (FFNN)	11
Number of epochs (FFNN)	2000

Figure 4.6 shows the results for Benchmark Problem 3. The computed Rayleigh quotients and the absolute error relative to the eigenvalue computed by Numpy are tabulated in Table 4.9.



(a)



(b)

Figure 4.6: *Results for Benchmark Problem 2 with a 3×3 real symmetric matrix $-A$. **(a)** shows the components of the computed steady-state vector as a function of time. The dashed lines are the components computed by Euler's method and the solid lines are computed by the FFNN model. The dotted lines are the normalized components of the eigenvector corresponding to the smallest eigenvalue, as computed by Numpy's linalg.eig. All lines corresponding to the same vector component are here on top of each other. **(b)** shows the computed Rayleigh quotients, r, as a function of time for both Euler's method and the FFNN model. Both the largest and the smallest eigenvalue as calculated by linalg.eig are also shown. Three of the four lines are all at the same spot in the bottom. The numbers are rounded to 5 decimal points.*

Table 4.9: *The computed Rayleigh quotients at the final simulation time for both Euler's method and the FFNN model. The absolute error relative to the eigenvalue computed by Numpy is also listed.*

Method	Rayleigh Quotient	Absolute Error
Numpy	2.0	–
Euler	2.0	$3.553 \cdot 10^{-15}$
FFNN	2.0	$3.526 \cdot 10^{-12}$

As mentioned in Section 3.2.3, the starting vector $\boldsymbol{x_0}$ is itself an eigenvector of A corresponding to its smallest eigenvalue, and henche orthonormal to the eigenspace associated with the two bigger eigenvalues. Inline with theorem 2.3, the vector must converge to the eigenvector corresponding to the smallest eigenvalue, and henche the solution becomes constant since this is the initial point. Both methods did well in approximating the constant function. This is particularly easy for the Euler method, which as usual came out on top.

4.2.4 Benchmark Problem 4

Table 4.10 tabulates the problem and model parameters for Benchmark Problem 4 described in Section 3.2.4.

Table 4.10: *Problem and model parameters for Benchmark Problem 4.*

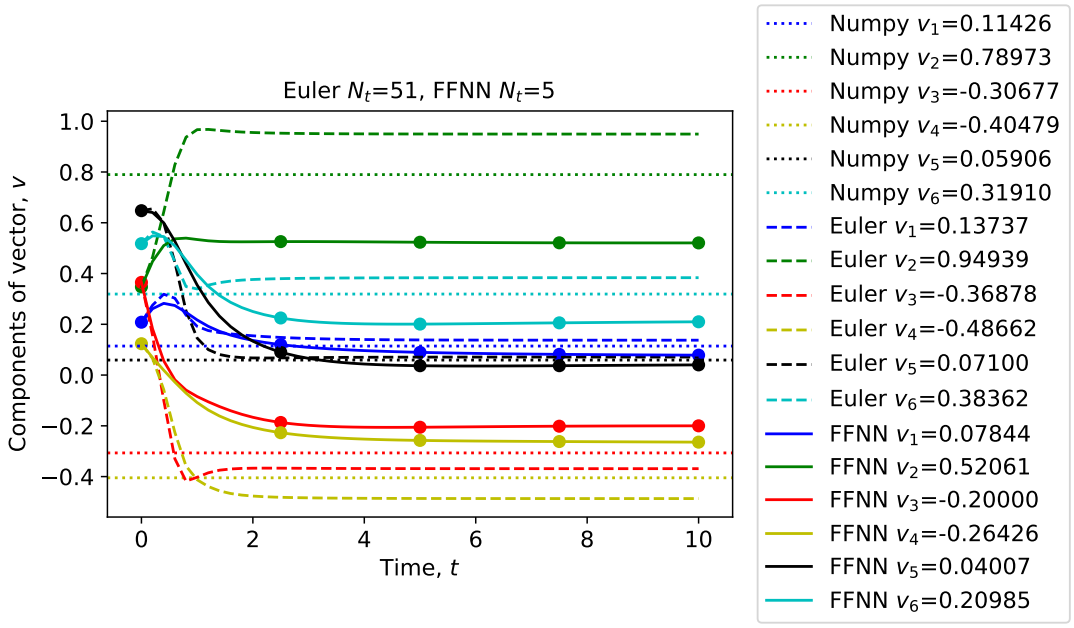
Parameter	Numerical Value
Initial vector	$\boldsymbol{x_0} = (0.208, 0.347, 0.365, 0.123, 0.648, 0.518)$
Simulation time	$T = 10$
Number of time points (Euler)	51, 101, 501, 1001
Number of time points (FFNN)	5, 11, 51, 101
Number of epochs (FFNN)	2 000

Figure 4.7 and Figure 4.8 shows the same convergence of the approximate eigenvector produced by both Euler and FFNN as with benchmark problem 2, but now for a 6×6 matrix and varying number of timepoints.

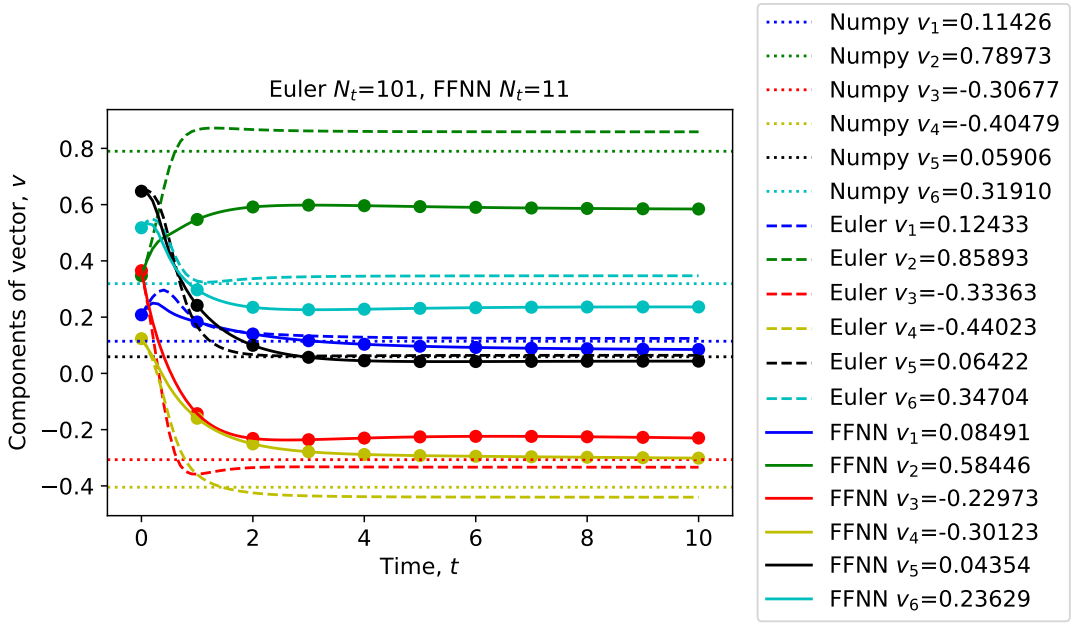
As seen from figure Figure 4.7, our earlier parameters such as number and time points $Nt = 11$ and final time $t = 3.5$ did not generalise to bigger matrices. To ensure convergence, the final time was increased to $t = 10$. However, both Euler and FFNN struggled to converge to the appropriate eigenvector.

Since a bigger matrix yields a more complicated ODE to solve, its reasonable to think that an increased number of time points is necessary to capture the finer details of the solution and thus increase the accuracy. Figure 4.8 confirms this is the case. However, Table 4.11 shows that eigenvalues are not as sensitive to the number of time points as the eigenvector. Even though the eigenvector did not converge fully for small number of time points, the eigenvalue still turned out fairly accurate. As allways, Euler was faster and more accurate than FFNN.

Further, the eigenvalue approximation actually, and perhaps interestingly, got worse with smaller time steps for the Euler method, even though the eigenvalue approximation clearly improved. For the FFNN solution we saw improvements for both the eigenvector and the eigenvalue until the number of time points reached 51. At 101 time points, the accuracy fell slightly. At least for dense time points, the accuracy of the FFNN can be improved by increasing the number of epochs, but of course at the cost of increased computational requirements.

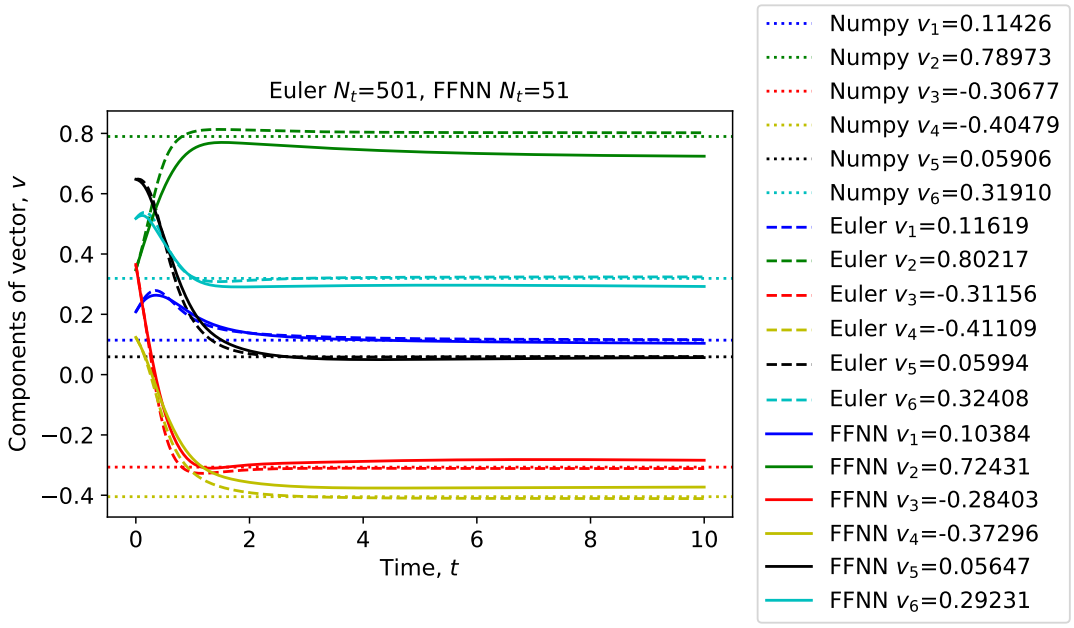


(a)

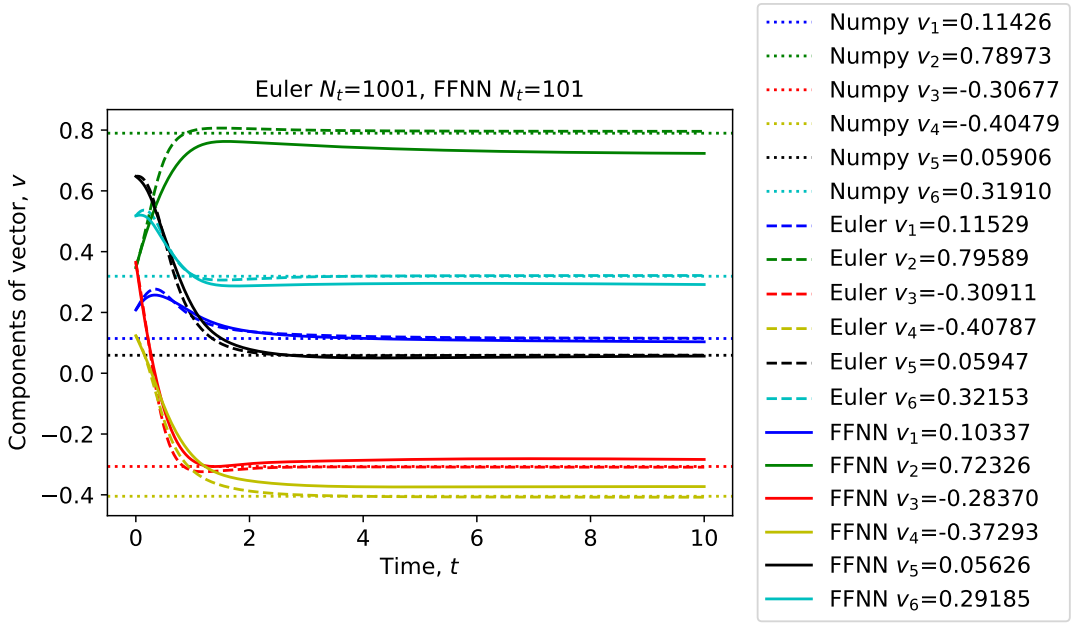


(b)

Figure 4.7: *Results for Benchmark Problem 4 with a 6 × 6 real symmetric matrix A. The figure shows the time evolution of the vector components of the solution computed by Euler's method and the FFNN solution with, respectively, the following time steps:* ***(a)*** *51 and 5,* ***(b)*** *101 and 11. The points where we have trained the FFNN are marked with dots on the graphs.*



(a)



(b)

Figure 4.8: *Results for Benchmark Problem 4 with a 6 × 6 real symmetric matrix A. The figure shows the time evolution of the vector components of the solution computed by Euler's method and the FFNN solution with, respectively, the following time steps:* ***(a)*** *501 and 51,* ***(b)*** *1001 and 101.*

Figure 4.9

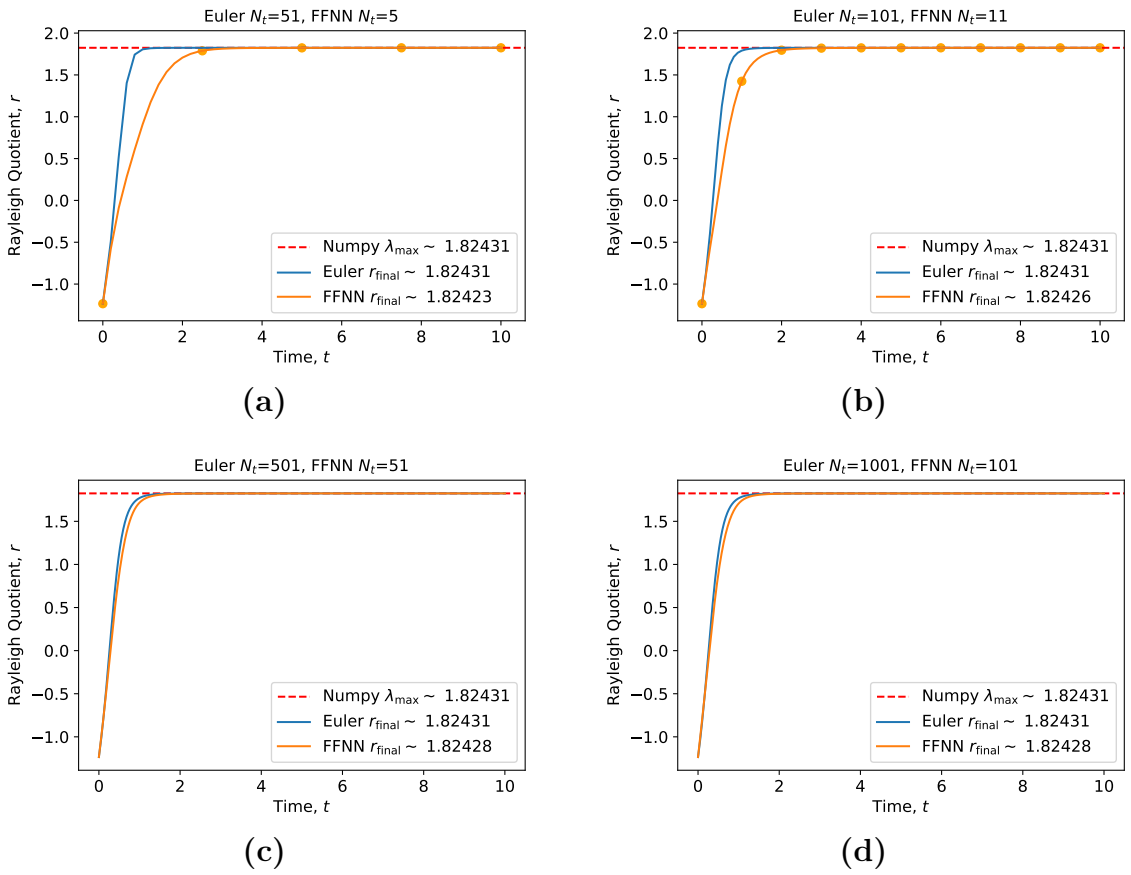


Figure 4.9: *Results for Benchmark Problem 5 with a 6×6 real symmetric matrix, A. The figure shows the time evolution of the Rayleigh quotient computed by Euler's method and the FFNN model with, respectively, the following time steps:* ***(a)*** *51 and 5,* ***(b)*** *101 and 11,* ***(c)*** *501 and 51,* ***(d)*** *1001 and 101. For number of time steps equal to 5 and 11, the points where we have trained the FFNN are marked with dots on the graph.*

The final Rayleigh quotients computed at different time steps and the absolute error relative to the eigenvalue computed by Numpy are tabulated in Table 4.11.

Table 4.11: *The computed Rayleigh quotients at the final simulation time for both Euler's method and the FFNN model with different time steps. The absolute error relative to the eigenvalue computed by Numpy is also listed.*

Method	Number of Time Steps	Rayleigh Quotient	Absolute Error
Euler	51	1.82431291	$4.781 \cdot 10^{-11}$
Euler	101	1.82431291	$2.060 \cdot 10^{-9}$
Euler	501	1.82431290	$1.505 \cdot 10^{-8}$
Euler	1001	1.82431289	$1.847 \cdot 10^{-8}$
FFNN	5	1.82423239	$8.052 \cdot 10^{-5}$
FFNN	11	1.82425943	$5.348 \cdot 10^{-5}$
FFNN	51	1.82428079	$3.212 \cdot 10^{-5}$
FFNN	101	1.82427677	$3.615 \cdot 10^{-5}$

5 Conclusion

In conclusion, neural networks succeeded in being trained to find and represent the solution of various differential equations, both the PDE heat equation and coupled ODE approximating eigenvalues. While achieving high accuracy, the FFNN models had several shortcomings with respect to more conventional methods such as simple Forward Euler. Most notably was the required time to train. Training a neural network is a complicated non-convex optimisation problem that generally requires many epochs. Thus, obtaining results of similar accuracy as Forward Euler could require 4 orders of magnitude longer time, which is very unpractical.

On the other hand, FFNN required way fewer grid points to obtain an accurate solution than Euler because of the lack of stability criterion. While not a useful result, since the number of grid points is not a rare commodity, it is an interesting result nonetheless. As usual, the FFNN model has the added benefit of offering interpolation out of the box, as it is a continuous function, rather than piece-wise.

Another shortcoming was revealed by the eigenvalue problem. A weakness of the method was that it relied on many hyper-parameters, which was difficult to tune without guidance from the exact solution itself. This partly made the FFNN method dependant on other, more reliable, methods, which in turn renders it unnecessary. It was especially hard to tune the final time t and number of time points N_t. The parameters seemed to generalize to some degree between matrices of the same size. By choosing sufficiently large t and N_t to reproduce the eigenvalues of some known matrices, one can generalize to other matrices of same size with some confidence. However, increasing the size requires to readjust the parameters.

6 Future Work

We did not study the computation time for the methods used in the eigenvalue problem in any detail, even though the Euler method clearly was faster. Another thing to investigate is the optimal distribution of the time points. Since the slope of the graphs for both the vector components and the Rayleigh quotient are highest near the start point, we could try e.g. time points numbers spaced evenly on a logarithmic scale.

In the problems we studied, the network did not outperform traditional methods. It might be interesting to look at cases where there is a possibility that the network performs better than the traditional methods.

References

[1] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix". In: *Computers & Mathematics with Applications* 47.8 (2004), pp. 1155–1164. ISSN: 0898-1221. DOI: https://doi.org/10.1016/S0898-1221(04)90110-1. URL: http://www.sciencedirect.com/science/article/pii/S0898122104901101 (cit. on pp. 5, 10).

[2] H.P. Langtangen. *Finite difference methods for diffusion processes*. Lecture notes for the course IN5270 at University of Oslo. 2016 (cit. on pp. 2, 4, 6).

[3] J. Dieudonné. *Foundations of Modern Analysis*. Foundations of Modern Analysis v. 1;v. 10. Academic Press, 1969 (cit. on p. 3).

[4] N. Haug, T. K. Netskar, and K. Wold. *FYS-STK4155 Project 2 - Classifying Phases of the 2D Ising Model with Logistic Regression and Deep Neural Networks*. URL: https://github.com/nicolossus/FYS-STK4155-Project2/blob/master/report/FYS_STK4155_Project2.pdf (visited on 12/19/2019) (cit. on p. 4).

[5] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: http://www.sciencedirect.com/science/article/pii/0893608089900208 (cit. on pp. 4, 5).

[6] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. 2nd ed. Cambridge University Press, 2013. ISBN: 9780521839402 Hardback (cit. on p. 5).