FYS-STK4155 Project 3

Joakim Flatby, Linus Ekstrøm

December 2019

Abstract

In this project we study two different methods of solving the one dimensional heat equation: the forward explicit Euler method as well as using a neural network to solve differential equations. We found the forward method works well for sufficiently large Δx, however the method suffers from a stability requirement which scales with $(\Delta x)^2$. We found the neural network to work well with the heat equation, however the solutions tend to decrease in accuracy as the time increases. Finally we looked at work done in the 2004 paper by Yi. et al [4] where they present a differential equation model for a neural network which converges to the largest and smallest eigenvalue of a $(n \times n)$ symmetric matrix. We did have success in finding the eigenvalues using our neural network.

Contents

1 Introduction

Solving partial differential equations, PDE's, is something done in practically any scientific field of study. They allow us to model a great number of phenomena. In the vast majority of cases they are not analytically solvable, forcing us to rely on numerical methods. This project will be concerned with a few approaches to solving PDE's numerically. The PDE we will be concerned with is the one dimensional diffusion equation, which can be interpreted as the time evolution of the temperature gradient in an idealized rod of length L. First we will solve the system of equations using the explicit forward Euler algorithm, then we will solve the same system using a neural network model. In addition to PDE's, neural networks can also be used to solve for eigenvalues of matrices, this will also be studied in this project. We will compare our neural network to existing numerical diagonalization algorithms form linear algebra.

2 Theory

2.1 The Heat Equation

The heat equation is a partial differential equation in space and time, which describes the evolution of the temperature gradient in a specified region of space over a specified time interval. The equation was first developed and solved by Joseph Fourier in 1822 to describe heat flow, however the equation is used in fields from quantum mechanics to economics to theoretical topology. The derivation is not too hard to follow, however we will not derive it here. A full derivation can be found in [1]. One starts with Fourier's law; *the rate of heat flow through a surface is proportional to the temperature gradient across the surface.* From this along with some clever integrals we end up with the heat equation on the form

$$\frac{k}{c_p\rho}\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$$

For the rest of this project we look at a medium where the constant on the left hand side is equal to one.

2.1.1 Analytic Solution

In the following solution we have grouped the constants into $\alpha^2 = k/c_p\rho$. In order to solve the partial differential equation we assume the solution $u(x,t)$ can be split up as

$$u(x,t) = X(x)T(t)$$

where $X(x)$ contains all the spatial dependence, and $T(t)$ carries all the temporal dependence. Plugging this assumption into the the heat equation we get

$$\begin{aligned}
\alpha^2\frac{\partial^2 XT}{\partial x^2} &= \frac{\partial XT}{\partial t} \\
\textit{Shorthand}: \; u_a &= \frac{\partial u}{\partial a} \\
\alpha^2\partial_x\Big(\partial_x XT\Big) &= \partial_t XT \\
\alpha^2\partial_x\Big(X_xT + XT_x\Big) &= X_xT + XT_x \\
\alpha^2\Big(X_{xx}T + 2X_xT_x + XT_{xx}\Big) &= XT_t \\
\frac{1}{X}\frac{\partial^2 X}{\partial x^2} &= \frac{1}{\alpha^2 T}\frac{\partial T}{\partial t}
\end{aligned}$$

where we have used that the spatial and temporal functions are both equal to zero when differentiated with respect to the other variable. As both sides are independent of the other variable the equality can only be achieved if both sides are constant. This reduces the original equation into two ordinary differential equations

$$\frac{1}{X}\frac{\partial^2 X}{\partial x^2} = k$$

$$\frac{1}{\alpha^2 T}\frac{\partial T}{\partial t} = k$$

with k as an undetermined constant. Depending on this constant, the spatial part X will have different solutions. If $k = 0$ then $X(x) = ax + b$. The solution becomes $X(x) = Ae^{\omega x} + Be^{-\omega x}$ when $k = \omega^2 > 0$, or $X(x) = Ae^{i\omega x} + Be^{-i\omega x}$ when $k = -\omega^2 < 0$. The temporal equation has the solution $T(t) = C$ when $k = 0$, or $T(t) = Ce^{\alpha^2\omega^2 x}$ otherwise.

2.1.2 Boundary Conditions

Now in order to find the solution which fits our problem description from the sea of possible solutions to the PDE we have to apply boundary conditions specific to our problem of study. In our case we have $u(0) = u(L) = 0$, known as Dirichlet boundary conditions. In addition the initial state of the system is on the form $u(x; t = 0) = \sin(\pi x)$. We notice that the if the constant k goes to zero then by the boundary condition both A and B also have to vanish. This is also evident for the solutions with $k = \omega^2 > 0$. It follows that we consider the last remaining case for the value of the constant k. We end up with $X(0) = A\cos(\omega x) + B\sin(\omega x) = 0 \rightarrow A = 0$, and $X(L) = B\sin(\omega x) = 0 \rightarrow \omega = \pi n$. This means we obtain an infinite set of solutions, one corresponding with each value of n. The spatial solution, for $t = 0$, can thus be written as an infinite sum of component functions

$$u(x; t = 0) = \sum_{n=1}^{\infty} B_n \sin(n\pi x)$$

To find the coefficients B_n for this infinite sum we apply Fouriers trick

$$B_n = 2\int_0^1 u(x;t=0)\sin(n\pi x)dx$$

Now that we have the value for $\omega = n\pi$ fixed we use this to solve for the temporal solution. We obtain

$$T(t) = e^{-n^2\pi^2\alpha^2 t}$$

Putting it all together we have obtained a Fourier series representation of the initial condition strictly from considering the boundary conditions. We have

$$\begin{aligned} u(x,t) = X(x)T(t) &= \sum_{n=1}^{\infty} B_n \sin(n\pi x)e^{-n^2\pi^2\alpha^2 t} \\ &= \sum_{n=1}^{\infty} B_n \sin(\omega x)e^{-\omega^2\alpha^2 t} \end{aligned}$$

Analysing we notice this solution describes a system of infinite component functions each with a corresponding *frequency* ω which determines how quickly that component decays through time. This concept and the behaviour of the solutions to the heat equation is described incredibly well in the following video by 3Blue1Brown on Youtube [2]. Grant Sanderson, explains this behaviour from the original PDE very intuitively, *paraphrasing*: "if we look at the left and right hand side of the heat equation we see the change in time is proportional to the difference in temperature between the point in question and its neighbors. I.E. regions with larger differences even out quicker than regions with lesser." Moving on, our initial condition $u(x;t=0) = \sin(\pi x)$ allows our particular solution to take a very simple form

$$\begin{aligned} B_n &= 2\int_0^1 \sin(\pi x)\sin(n\pi x)dx \\ &\rightarrow B_1 = 1, \{B_2, ..., B_n\} = 0 \end{aligned}$$

Where the result of the other coefficients zeroing out comes from the orthogonality of $\sin(x)$. Thus our full solution is the following

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

2.2 Explicit Forward Euler

Next we wish to solve the heat equation computationally, to do this we use the rudimentary explicit forward Euler scheme. When working digitally both variables are discretized, and the derivatives are approximated first-order differences and second-order central differences

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}$$

and

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}$$

This gives the following form for the heat equation considered in the previous sections

$$u_j^{n+1} = (1 - 2\beta)u_j^n + \beta(u_{j+1}^n + u_{j-1}^n)$$

where the new $\beta = \alpha^2\Delta t/\Delta x^2 \geq \frac{1}{2}$ is known as the stability criteria for the explicit forward Euler method.

2.3 Differential Equations with Neural Networks

Next we consider solving a partial differential equation using a deep neural network. Deep neural networks are neural networks consisting of more than one hidden layer. We will consider the simplest form of deep neural network, a *multilayer perceptron*. This was elaborated in *Project 2: Classification and regression, neural networks* [3]. As opposed to regular supervised learning, where one uses predictions and labeled data to back-propagate the error and tune the weights and biases accordingly, in the PDE case we are looking for initial values for x and t which minimize the equation

$$\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2} = 0$$

because this would mean solving the partial differential equation. This naturally leads us to our choice of cost function

$$\mathbb{C} = \frac{1}{n}\sum^{n}\Big(\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2}\Big)^2 \tag{1}$$

Unlike the previous explicit Euler method, we now do not have any iterative expression which we solve to obtain the solution to the PDE. Instead here we supply the neural network with a trial function given by

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

The trial functions acts as a way to reign in the neural network to look for solutions where the boundary conditions of our problem are met. Here $N(x,t,P)$ is function describing the output of the network for a given value of x, t and network parameters P (*biases, weights*). We have decided to implement the neural network using Google's Tensorflow library. There are a bunch of choices available for our optimization scheme, we chose to use tensorflow's ADAM optimizer.

2.4 Eigenvalues of Symmetric Matrices with Neural Network

For the last part of this project we will follow along with the work done in the 2004 paper by Yi et al. [4] Proofs and more discussion is available in that paper. We will here go over the main result, which is a differential equation to solve using a neural network in order to obtain

the maximum and minimum eigenvalues of a $(n \times n)$ symmetric matrix. Yi et al proposes the following model for the neural network

$$\frac{dx(t)}{dt} = -x(t) + f(x(t) \tag{2}$$

where

$$f(x) = \left[x^T x \mathbf{A} + (1 - x^T \mathbf{A} x)\mathbf{I}\right] x$$

3 Method

In this section we present our approach to both the explicit forward Euler solution as well as our neural network PDE solver. We start of with the simple integration script we wrote for the one dimensional heat equation with Dirichlet boundary conditions. Then we move on to our neural network PDE solver and finally we discuss our efforts to predict the eigenvalues of symmetric matrices using our neural network.

3.1 Numerical Solution to PDE

In order to calculate the numerical solution to the one dimensional heat equation using the explicit forward Euler scheme, we first discretize both the spatial and temporal dimension. We set up the step Δx and fix the Δt according to the stability criterion defined in the theory section for the explicit Euler. This is one of the negative sides of the algorithm. The Δt decreases as $\frac{1}{2}\Delta x$, so when Δx becomes very small the required Δt becomes even smaller. This results in long computation times when we decrease Δx. After discretizing we perform the actual explicit forward loop

1. for each time t we calculate $u^{n+1}(x,t)$ according to 2.2.
2. this updates the state for each time step and results in solving for the specified time we wish to solve the equation up to.

3.2 Tensorflow Neural Network PDE Solver

To solve differential equations using tensorflow, we first set up our datagrid containing values for x and t. Using numpy's meshgrid, ravel and reshape we make sure the data is in the right configuration for our purposes. Lastly we use tensorflow's convert_to_tensor, and concatenate, to make sure tensorflow can handle the data properly. Next, we set up the network by defining the amount of hidden layers and neurons. We iterate through each layer, and use tf.layers.dense to calculate the output of each layer using the output from the previous layer. After this we have the output from the final layer, and are ready to define our cost function. We calculate the cost function using equation (1). Next, we define how the network should be trained, using tensorflow.train.AdamOptimizer with a chosen training rate. We now have everything ready and can run the entire process using tf.Session().

3.3 Eigenvalues of Symmetric Matrices

Computing eigenvalues using the neural network is very similar to the PDE solver. The network itself is prepared and set up in the same way, but instead of using equation (1) as our cost function, we use equation (2) to define our cost function. Based on a constant k, this cost function will guide the network towards the components of the highest or lowest eigenvector. When the accuracy is high enough(i.e. when the loss is below a certain ϵ), the network stops iterating and we have arrived at our final approximation for the eigenvector. With this in hand we can calculate the corresponding eigenvalue, and thus we have completed our goal.

4 Results

In this section we present the main findings of this project in the order which our methods as described in the previous section.

4.1 Explicit Forward Euler

Below is a collection of figures of our solutions to the one dimensional heat equation solved using the explicit forward Euler method. We notice the Dirichlet boundary conditions acting like 'heat sinks' for the system. We start of with a heat distribution of $u(x; t = 0) = \sin(\pi x)$ which then exponentially decays following the solution to the temporal solution.

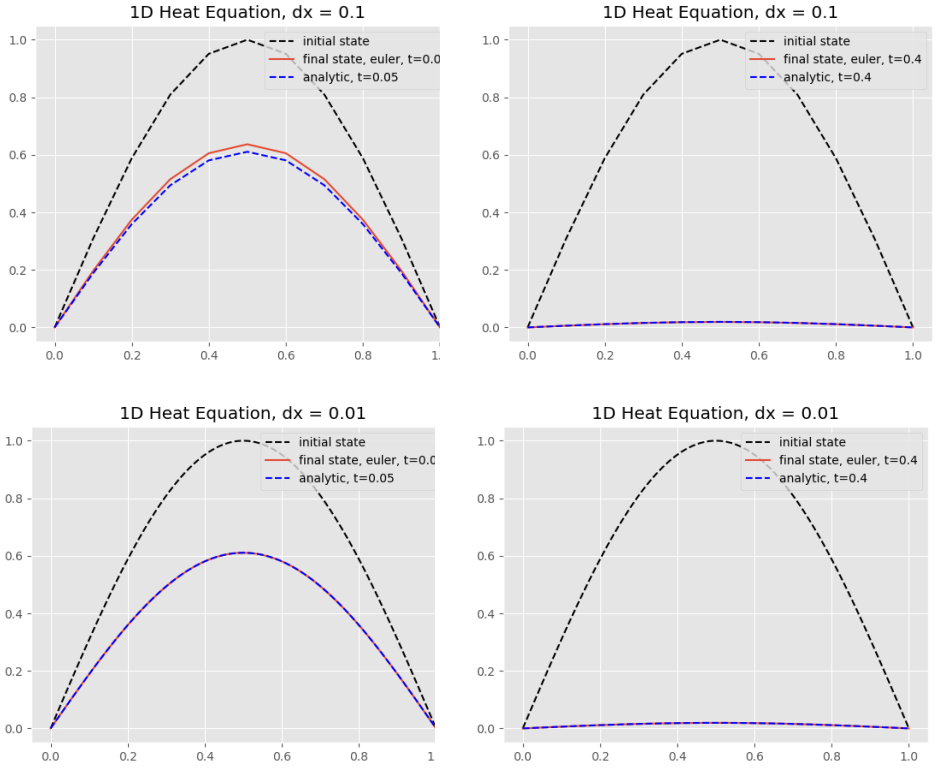

Figure 1: Solutions from the explicit forward Euler compared to the analytic solution for $\Delta x \in \{0.1, 0.01\}$ and Δt set according to the stability criteria. We used times $t = 0.05$ and $t = 0.4$ to obtain respectively the upper and lower graphs.

4.2 Solution of Heat Equation with Neural Network

Below we present the solution our neural network had to the one dimensional heat equation

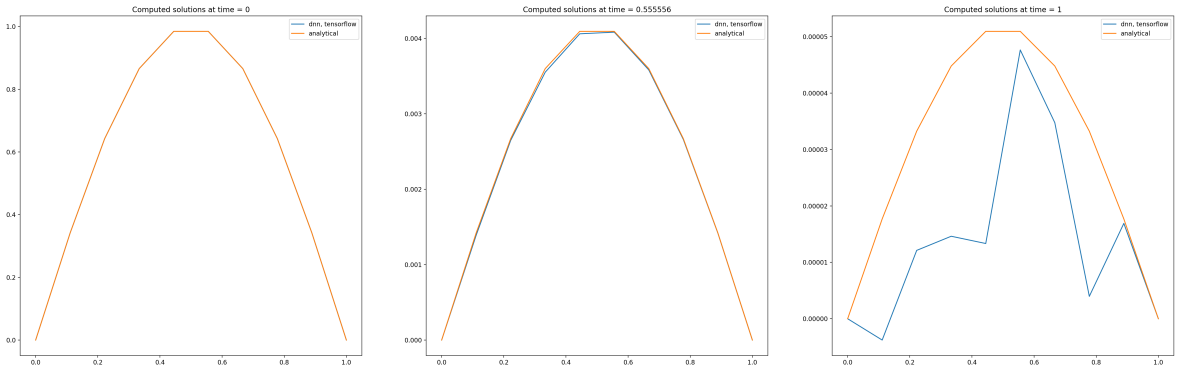

Figure 2: Solutions the heat equations at respectively $t = 0$, $t = 0.555...$, $t = 1$.

We see the solution diverge from the analytic solution the greatest after more time has passed. However, it still sort of follows the analytic solution even then. For the initial time and $t = 0.555..$ we have very good convergence to the analytic solution. This representation makes it look like the analytic solution stays the same, however this is due to the automatic scaling of the y-axis. Here the y-axis is representing some temperature and the x-axis is representing the spatial dimension of a rod-like object.

4.3 Eigenvalues of Symmetric Matrices

Below is a table of our neural networks prediction of the smallest eigenvalue of the following matrix(generated randomly):

$$\mathbf{A} = \begin{bmatrix} 0.5507979 & 0.41686657 & 0.47002439 & 0.38504003 & 0.71879799 & 0.93614426 \\ 0.41686657 & 0.20724288 & 0.16497724 & 0.42795552 & 0.40509549 & 0.56460845 \\ 0.47002439 & 0.16497724 & 0.6762549 & 0.43719395 & 0.16517271 & 0.7308441 \\ 0.38504003 & 0.42795552 & 0.43719395 & 0.69313792 & 0.3312058 & 0.5013093 \\ 0.71879799 & 0.40509549 & 0.16517271 & 0.3312058 & 0.38797126 & 0.65718885 \\ 0.93614426 & 0.56460845 & 0.7308441 & 0.5013093 & 0.65718885 & 0.09221701 \end{bmatrix}$$

	eigenvalue	v_1	v_2	v_3	v_4	v_5	v_6
numpy eig	-0.7904	-0.3121	-0.2265	0.2019	0.2986	0.0246	0.8490
30 000 iter	-0.6990	-0.4683	-0.4377	-0.1997	0.0494	0.1374	0.7266
50 000 iter	-0.7225	-0.4339	-0.4180	-0.2089	0.0607	0.0966	0.7618
100 000 iter	-0.7509	-0.4058	-0.3846	-0.2302	0.0465	0.0290	0.7946
200 000 iter	-0.7602	-0.4002	-0.3746	-0.2445	0.0322	-0.0086	0.7992
300 000 iter	-0.7657	-0.3888	-0.3585	-0.2429	0.0353	-0.0143	0.8123

Table 1: Value of the smallest eigenvalue as a function of the number of iterations.

as we can see from table 1, the network is undoubtedly mostly headed the right direction. There is still a lot that doesn't make sense to us though, like why the third component is flipped, and why the fifth component decides to go further away from the correct value. Maybe that one is also "flipped" and is converging towards -v_5 and not v_5..

Below you can also see plots, as we can see, v_2 and v_5 are the ones that do not converge, which makes sense from the data in table 1. Due to other exams and time restrictions we have not been able to test with huge amounts of data or let it run for longer than 300 000 iterations with t ranging from 0 to 3 with a dt of 0.1. Our code might be working better than we think, if we were to run the code with 0.01 dt from 0 to 10 t with 500 000 iterations, but sadly we don't have time to test this.

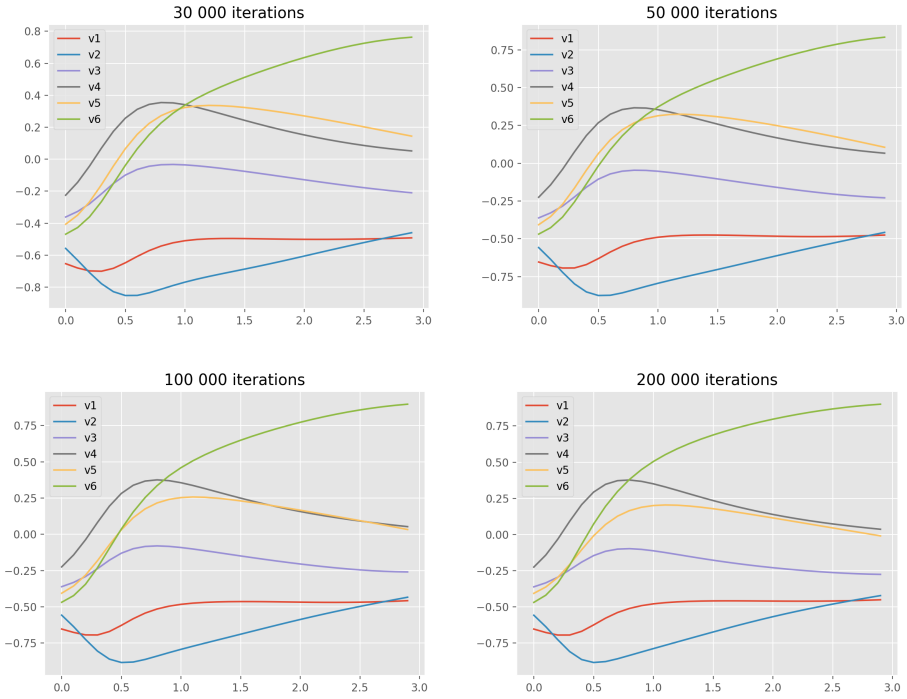

Figure 3: Components of the eigenvector corresponding to the smallest eigenvalue for the matrix $\mathbf{A}$. This same analysis can be done for the largest eigenvalue by swapping the sign of the matrix $\mathbf{A}$. We see the values do not fully converge as smooth as in the paper by Yi. et al.

5 Discussion

5.1 Explicit Forward Euler

Our program for the explicit forward Euler method seems to perform exactly as expected. The main drawback of this method is the stability criteria which must be met to obtain a good solution. This means that if we wish to obtain a great accuracy in the spatial dimension we must have an accuracy in the temporal dimension which is $\Delta t \propto (\Delta x)^2$. So when Δx shrinks one could suddenly end up with having a simulation which is too time consuming for any practical use.

5.2 Heat Equation with Neural Network

We did obtain comparable results using our neural network, however we see deviation when proportional to the time passed since the initial time. However, this method does not have the same stability requirement and could conceivably by tune to handle greater accuracy than the explicit forward Euler method. We, sadly, did not have enough time to look into this as much as we would like and this is definitely a point which we would focus on with further study into the subject. We see an exponential decline in the temporal dimension and a decline in the amplitude of a sinusoidal-like function in the spatial dimension, which is exactly what is expected from the known behavior of the heat equation with Dirichlet boundary conditions.

5.3 Eigenvalues of Symmetric Matrices

We are not able to fully wrap our head around some of the results from our eigenvalue-computations. Assuming we let it run for a sufficient amount of time, we always end up at the correct eigenvalue. However, when comparing the eigenvector(the one we used to calculate the eigenvalue) to the eigenvector given by numpy for this eigenvalue, they don't match that well. Some of the components are really close, some of them are close but with the wrong sign, and some of them are up 10 times as large or small as they are supposed to be..(e.g. the component given by numpy is 0.1 and we get 0.01). We have tried normalizing the vectors and they just don't seem to be right. We are uncertain as to what makes this work even though it seems so wrong. As we've learned through the years, the rules of eigenvectors are heavily governed by things like symmetry, so maybe there is something going on here which makes sense but that we don't understand. So for some reason, our neural network computes a set of components for a vector which does not seem to correspond to any of the real eigenvectors. But, depending on the value of k(-1 or 1), it always finds the correct(smallest or largest) eigenvalue, assuming we let it run for enough iterations.

6 Conclusion

In this project we have studied differential equations of various kinds, as well as different ways to solve the problems. We took a look at probably one of the simplest methods: the explicit forward Euler and had success in implementing the method using Python. We found that the method performs sufficiently when the required precision is not too small. This is because of the stability requirement present in the method. In addition we also looked at solving the heat equation using an implementation with Tensorflow, where we minimized what boils down to the left and right hand side of the one dimensional heat equation. This work could be adapted to solve the heat equation in multiple spacial dimensions with not too much extra work. This is definitely an area of study we could do with granted more time for study. We found that the neural network solves the equation pretty well when the time is not too large, however that it deviates from the analytic solution as the time increases. For the eigenvalue problem of the symmetric matrices we did obtain fairly good results for the eigenvalues, but as described in the respective discussion section there is some weird behaviour of the eigenvectors we have not yet identified the cause of. This is also something we would like to look further into granted more time to study this problem. All in all we have learned a bunch about both differential equations and implementing neural networks to solve them. This is a very different, but also familiar, way of looking at neural networks. The cost function is certainly a different beast than the ones we are used to from the previous projects, but the network does function in a fairly similar way, it is just the formulation which has changed.

References

[1] Cannon, John Rozier (1984), p.13 The One–Dimensional Heat Equation, Encyclopedia of Mathematics and Its Applications, 23 (1st ed.)

[2] Sanderson, Grant (2019), Solving the Heat Equation, from the video series on differential equations. `https://www.youtube.com/watch?v=ToIXSwZ1pJU&t=10s`

[3] Flatby, Ekstrøm, Project 2 FYS-STK4155, on the multilayer perceptron and more. `https://github.com/jflatby/FYS-STK4155/blob/master/Project2/Report/FYS_STK4155_Project_2.pdf`

[4] Yi. et al. Computers Mathematics with Applications, Volume 47, April-May 2004, p.1155-1164, `https://www.sciencedirect.com/science/article/pii/S0898122104901101`