FYS-STK4155 Project 3: Solving PDEs using neural networks

Jan Ole Åkerholm
(Dated: December 18, 2019)

Abstract

Link to github repository: Click here

I. INTRODUCTION

Partial differential equations are a central topic in a wide range of sciences, including mathematics, physics, biology and many more. Finding analytical solutions of these equations are not always practical, so fast and accurate numerical methods are of great interest. Traditionally the methods of choice have often been finite difference or finite element methods. In particular finite difference methods use discrete numerical differentials to replace the exact mathematical ones from the equations, and calculate the solution from this. The advantage is that it is relatively easy to understand how this works, but the disadvantage is that the method may be inaccurate or even completely wrong if the differentials are not calculated appropriately.

A different approach is to take advantage of the fact that a neural network is able to approximate any function, and use this property to calculate the solution of a differential equation. There may be several advantages to this approach, first and foremost in the ability of the method to provide an accurate result. This has been attempted before, notably by Lagaris et al. [4] and Chiaramonte and Kiener [1], with promising results.

In addition, this method may even be applied to finding the eigenvalues and eigenvectors of a real, symmetric, square matrix. This is also something of great interest in many fields, and has previously been studied by Yi et al. with promising results [5].

II. THEORY AND METHOD

A. First PDE

The first partial differential equation (PDE) I will solve in this project is a specific case of the diffusion equation:

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

with $L = 1$, the initial condition at $t = 0$ is given by:

$$u(x, 0) = \sin(\pi x) \tag{2}$$

And I use Dirichlet boundary conditions:

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

The typical interpretation of the equation in this form is that it describes the temperature of a very thin (1D) rod of length L, which starts with a higher temperature in the middle while the temperature on the edges are kept at a constant of 0.

1. Analytical solution

An analytical solution of the PDE can be found without much problem, and is useful in order to compare the numerical results with what we know to be correct. First assume the solution can be separated into two factors, one dependant only on x and the other only on t, i.e.

$$u(x, t) = X(x)T(t), \quad X(0) = X(L) = 0 \tag{4}$$

Inserting in the differential equation (1):

$$T(t)\frac{dX(x)}{dx} = X(x)\frac{dT(t)}{dt} \tag{5}$$

$$\implies \frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)} \tag{6}$$

Because each side now only depends on the independent variables x and t respectively, each side must equal the same constant, $-\lambda$. This gives the two separate equations:

$$X''(x) + \lambda X(x) = 0 \tag{7}$$

$$T'(x) + \lambda T(t) = 0 \tag{8}$$

Looking at equation 7 first, note that this has different results depending on whether λ is larger than 0, smaller than 0 or equal to 0. First, if $\lambda < 0$, the characteristic equation gives the solution:

$$X(x) = c_1 \cosh(\sqrt{-\lambda}x) + c_2 \sinh(\sqrt{-\lambda}x)$$

From the boundary conditions, $X(0) = 0 = c_1$, and

$$X(L) = c_2 \sinh(\sqrt{-\lambda}L) = 0$$

$$\implies c_2 = 0$$

Because this is a trivial solution it is not very interesting, so all solutions for $\lambda < 0$ can be discarded.

Next, for $\lambda = 0$ the solution is simply $X''(x) = 0 \implies X(x) = c_1 + c_2 x$

From the boundary conditions, $X(0) = 0 = c_1$ and $X(L) = c_2 L = 0 \implies c_2 = 0$.

Again this makes the solution trivial, so solutions with $\lambda = 0$ can be discarded.

Finally, with $\lambda > 0$ the characteristic equation gives the solution

$$X(x) = c_1 \cos(\sqrt{\lambda}x) + c_2 \sin(\sqrt{\lambda}x) \tag{9}$$

Now, the boundary conditions gives:

$$X(0) = 0 = c_1$$

and

$$X(L) = 0 = c_2 \sin(\sqrt{\lambda}L) \tag{10}$$

For (10) to be true, the argument in the sine function must take the value $n\pi$ for $n = 1, 2, 3, ...$ ($n = 0$ gives a trivial solution). I.e.

$$\sqrt{\lambda_n}L = n\pi \implies \lambda_n = \left(\frac{n\pi}{L}\right)^2 \tag{11}$$

With $L = 1$ the solution for $X(x)$ then is:

$$X_n(x) = c_2 \sin(n\pi x) \tag{12}$$

With λ known, the solution for $T(t)$ (eq 8) can be found:

$$T_n(t) = C_n e^{-(n\pi)^2 t} \tag{13}$$

Combining (12) and (13), the general solution is obtained:

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

where c_2 has been combined into C_n. Next, the initial condition lets us find the final solution:

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

It's clear that the solution is correct if $n = C_n = 1$, leaving the final analytical solution as:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t} \tag{14}$$

2. Explicit discretization scheme

For the finite difference scheme, the differential equation will be approximated using, as the name implies, finite differences for the differentials. First, the time differential is discretized using the Forward Euler (FE) discretization:

$$\frac{\partial u(x,t)}{\partial t} \simeq \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \tag{15}$$

The spatial differential is a second order derivative, so instead of using FE, this is discretized using a centered difference:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \simeq \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2} \tag{16}$$

where the simulation domain, $x \in [0,1]$ and $t \in (0,1]$ is divided into a discrete grid, such that $t_n = n\Delta t$ and $x_i = i\Delta x$ (note that $t = 0$ is not included in the simulation, as the initial condition is known, but t_0 still exists on the grid).

Using the notation $u_i^n = u(x_i, t_n)$, and the above discretizations (15 and 16), the differential equation can then be approximated as:

$$\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} = \frac{u_i^{n+1} - u_i^n}{\Delta t} \tag{17}$$

The solution can then be computed by calculating each next time step from the previous one, since the initial condition is known. Solving for u_i^{n+1}:

$$u_i^{n+1} = \frac{\Delta t}{(\Delta x)^2}\left[u_{i+1}^n - 2u_i^n + u_{i-1}^n\right] + u_i^n \tag{18}$$

Note that only the state of the previous time step is required to calculate the next one, and since the boundaries are always equal to 0, the solution can be calculated only for the internal points of x_i on the simulation grid.

This method has the stability criterion of

$$\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$$

for which it is stable. For other values the results may be unpredictable and chaotic.

3. Solving the PDE using a deep neural network (DNN)

For an in-depth discussion of neural networks and how to solve differential equations with neural networks, see the notes by Morten Hjorth-Jensen [3] and Kristine Baluka Hein [2].

The general idea behind solving the PDE using a neural network is to create a trial function $u_t(x,t)$ which in some way depends on the output of the neural network, and also makes sure the initial and boundary conditions are met. The trial function is then substituted in the original differential equation (1) in order to find an approximation:

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

The error of the approximation is then found by moving one term to the other side:

$$E = \frac{\partial^2 u_t(x,t)}{\partial x^2} - \frac{\partial u_t(x,t)}{\partial t} \tag{19}$$

A grid of t and x values are created, and the error function is calculated on each point on the grid. The error is squared, and the sum of this is the cost function which is to be minimized using a number of training iterations.

The trial function must be chosen in such a way that it fulfills the criteria given by the initial condition and boundary conditions, and it must also depend on the output from the neural network. In this project I used the following trial function:

$$u_t(x,t) = (1-t)I(x) + x(1-x)tN(x,t,P) \tag{20}$$

Where $I(x)$ is the initial condition (2) and $N(x,t,P)$ is the output from the neural network at position x, time t and with P representing the current set of weights and biases of the network.

Note that when $t = 0$, the trial function reduces to $u_t(x,0) = I(x)$, and when $x = 0$ or $x = 1$, it reduces to 0 (since $I(0) = I(1) = 0$).

For each training iteration, the state of the network with the current weights and biases are calculated. The trial function and its derivatives are then calculated and the cost function evaluated. As the error E approaches 0, the trial function should then approach the solution of the PDE.

B. Finding eigenvectors and eigenvalues with neural networks

As explained by Yi et al. [5], the largest eigenvector of an $n \times n$, real and symmetric matrix A can be found by

solving the ordinary differential equation

$$\frac{dv(t)}{dt} = -v(t) + f(v(t)), \quad t \geq 0 \tag{21}$$

where

$$v = [v_1, v_2, ..., v_n]^T$$

and

$$f(v) = [v^T vA + (1 - v^T Av)I]x$$

where I is the $n \times n$ identity matrix.

When $t \to +\infty$, $v(t)$ will then approach the largest eigenvector $v_{\max}$ of the matrix A, provided that $v(t)$ is initialized as some initial vector which is non-zero and not orthogonal to $v_{\max}$. The smallest eigenvector can be computed by substituting A with $-A$ for the computation.

Once an eigenvector is found, its corresponding eigenvalue w can be found from:

$$w = \frac{v^T Av}{v^T v} \tag{22}$$

1. Trial function

In order to use the same method as in the section for the PDE to solve the differential equation, a trial function is required. Because $v(t)$ is a vector containing n values dependant on time, this can be described as a two-dimensional equation in x and t, where $x = [1, 2, ..., n]$. The trial function used in this project is then:

$$v_t(x, t) = v_0 + tN(x, t, P) \tag{23}$$

where as before, $N(x, t, P)$ is the output from the neural network, and P describes the weights and biases at any iteration. v_0 is the initial v.

This can then be inserted in the differential equation (21):

$$\frac{\partial v_t(x, t)}{\partial t} = -v_t(x, t) + f(v_t(x, t)) \tag{24}$$

The error is then given by:

$$E = -v_t(x, t) + f(v_t(x, t)) - \frac{\partial v_t(x, t)}{\partial t} \tag{25}$$

This error squared and summed over all the time steps is then used as the cost function.

III. IMPLEMENTATION

The implementation for both the PDE and the eigenvalue problem is done with TensorFlow, by using the methods outlined by Kristine B. Hein [2]. The finite difference method is implemented as a simple Python function with a loop over the time steps. To avoid a (slow) Python loop for the spatial direction, Numpy array slicing is used instead. The integration loop is as follows:

```
for n in range(N_t - 1):
    # Vectorized method for the
    # spatial calculation skips one Python loop
    u_array[n + 1, 1:-1] = (C*(u_array[n, 2:]
                            - 2*u_array[n, 1:-1]
                            + u_array[n, 0:-2])
                            + u_array[n, 1:-1])

    t_array[n + 1] = t_array[n] + dt
```

Because Numpy slicing is *much* faster than a regular Python loop, the speedup over a more traditional method is significant. The same method using two loops would look as follows:

```
for n in range(N_t - 1):
    for i in range(1, N_x - 1):
        # only loop through the internal
        # points since the edges are always 0.
        # Outermost edge is at index N_x - 1
        u_array[n + 1, i] = (C*(u_array[n, i + 1]
                             - 2*u_array[n, i]
                             + u_array[n, i - 1])
                             + u_array[n, i])

    t_array[n + 1] = t_array[n] + dt
```

IV. RESULTS AND DISCUSSION

V. CONCLUSION

REFERENCES

[1] M. M. Chiaramonte and M. Kiener. "Solving differential equations using neural networks." In: (2013). URL: http://cs229.stanford.edu/proj2013/ChiaramonteKiener-SolvingDifferentialEquationsUsingNeuralNetworks.pdf.

[2] Kristine Baluka Hein. *Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs.* https://compphysics.github.io/MachineLearning/doc/pub/odenn/html/odenn-bs.html. Accessed: 2019-12-18.

[3] Morten Hjorth-Jensen. *Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning.* https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html. Accessed: 2019-12-18.

[4] I E Lagaris, A Likas, and D I Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

[5] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix." In: (2004). URL: https://www.sciencedirect.com/science/article/pii/S0898122104901101.

VI. APPENDIX