Atticus Rex

CMDA 4604
Term Project

May 5, 2023

Introduction

The heat equation was first formulated in the 19th century. Many methods exist to solve the heat equation analytically and numerically, but no method is perfect. In its general form, the heat equation can be written as following:

$$\frac{\partial u}{\partial t}(x,t) - \frac{\partial}{\partial x}[k(x)\frac{\partial u}{\partial x}](x,t) = f(x,t) \tag{1}$$

Within the context of CMDA 4604, we have learned various methods to solve the heat equation analytically and numerically, such as the spectral method, finite element method, and time-stepping methods based on the derivative from the finite-element method. However, none of these methods are perfect. Many are computationally intensive, suffer from stability issues, or simply difficult to solve by hand in the case of when $k(x)$ is an irregular or nonuniform function.

This paper examines the use of a novel technique that uses the auto-differentiation capability of neural networks to solve the heat equation. This is a method that was proposed in Raissi, 2019 et al. [1]. This method makes use of neural networks as universal function approximators to match the solution to a partial differential equation [2].

1 Formluation of Cost Function

Given the following partial differential equation:

$$u_t = (k(x)u_x)_x$$

We seek a function $u(x,t)$ that satisfies this function. First, we need to define a cost function that preserves the initial conditions, boundary conditions and dynamics of the equation. Specifically in this problem, I will use $k(x) = |x|$, but with periodic boundary conditions specified by the following:

$$\begin{aligned} t \in [0,1], x \in [-1,1] \\ u(-1,t) = u(1,t) \\ u_x(-1,t) = u_x(1,t) \\ u(x,0) = u_0(x) \end{aligned}$$

Therefore, the cost function is generally:

$$C = MSE_0 + MSE_b + MSE_f$$

Where $f(x,t)$ is the following:

$$f(x,t) = u_t - (k(x)u_x)_x$$

Thus, each cost function becomes:

$$MSE_0 = \frac{1}{N_0}\sum_{i=1}^{N_0} |\hat{u}(x_i, 0) - u_0(x_i, 0)|^2$$
$$MSE_b = \frac{1}{N_b}\sum_{i=1}^{N_b} |\hat{u}(-1, t_i) - \hat{u}(1, t_i)|^2 + |\hat{u}_x(-1, t_i) - \hat{u}_x(1, t_i)|^2$$
$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_b} |\hat{u}_t(x_i, t_i) - (k(x)\cdot \hat{u}_x(x_i, t_i))_x|^2$$

Using the Finite Element Method with Time-Stepping to Provide Reference

The finite element method uses a discretization of both the spatial and temporal dimension of the differential equations to produce an approximation of the function. Given the heat equation (1):

$$u_t = u_{xx} + f(x,t)$$

We can multiply both sides of the equation by some candidate function $v(x)$ to lead to the weak formulation of the differential equation:

$$\frac{d}{dt}\langle u, v\rangle = \langle u_{xx}, v\rangle + \langle f, v\rangle$$

The inner product $\langle u_{xx}, v\rangle$ can be written as an energy inner product.

$$\langle u_{xx}, v\rangle = -\int_0^1 u_x(x,t)v_x(x,t)dx = -a\langle u, v\rangle$$

This produces the following general weak form:

$$\frac{d}{dt}\langle u, v\rangle + a\langle u, v\rangle = \langle f, v\rangle$$

It's important to note that $v(x)$ has to satisfy the same boundary conditions! Now pick some space $V_n = \text{Span}\{\phi_1, ..., \phi_N\}$ and look for some $u(t) \in V_n$ satisfying:

$$\frac{d}{dt}\langle u, v\rangle + a\langle u, v\rangle = \langle f, v\rangle$$

Note that this space V_n can be any set of candidate functions that we choose; in this paper we will use hat functions to approximate the function. Now, the full solution, $u(x,t)$ can be written as the sum of each of the candidate functions which depends on space multiplied by some constant that depends on time.

$$u(x,t) = \sum_{j=1}^{N} c_j(t)\phi_j(x)$$

$$v(x) = \phi_k(x)$$

We can substitute this result into our weak formulation to yield the following:

$$\frac{d}{dt}\sum_{j=1}^{N} c_j(t)\langle \phi_j, \phi_k\rangle + \sum_{j=1}^{N} c_j(t)a\langle \phi_j, \phi_k\rangle = \langle f, \phi_k\rangle$$

And we see that this is essentially a single order linear system of Ordinary Differential Equations with a mass and stiffness matrix (M and K) as well as a load vector, f:

$$M\frac{dc}{dt} + Kc = f(t)$$

Decomposed, we can see this abreviation of the system:

$$\left[\langle \phi_j, \phi_k\rangle\right]\begin{bmatrix} c_1'(t) \\ ... \\ c_N'(t)\end{bmatrix} + \left[a\langle \phi_j, \phi_k\rangle\right]\begin{bmatrix} c_1(t) \\ ... \\ c_N(t)\end{bmatrix} = \begin{bmatrix} \langle f, \phi_1\rangle \\ ... \\ \langle f, \phi_k\rangle\end{bmatrix}$$

Hence, the continuous-time general solution to this system is:

$$c(t) = e^{-M^{-1}Kt}c(0) + \int_0^t e^{-M^{-1}K(t-s)}M^{-1}f(s)ds$$

However, this is cumbersome to compute, and thus we will be using a discrete time-stepping method to calculate this as well, specifically 4th-order Runge Kutte integration solving the following system:

$$c'(t) = -M^{-1}Kc + M^{-1}f(t)$$

Solving this equation determines how the coefficients $c(t)$ evolve over time, and multiplying these results with each $\phi_j(x)$ determines the spatial distribution. The initial conditions used in this paper were:

$$u_0(x) = \begin{cases} -1, x \in [-1, -0.5] \\ 0, x \in [-0.5, 0.5] \\ 1, x \in [0.5, 1] \end{cases}$$

This reference was simulated in MATLAB to show how the conditions within the bar would evolve over time. This is shown in Figure 1.

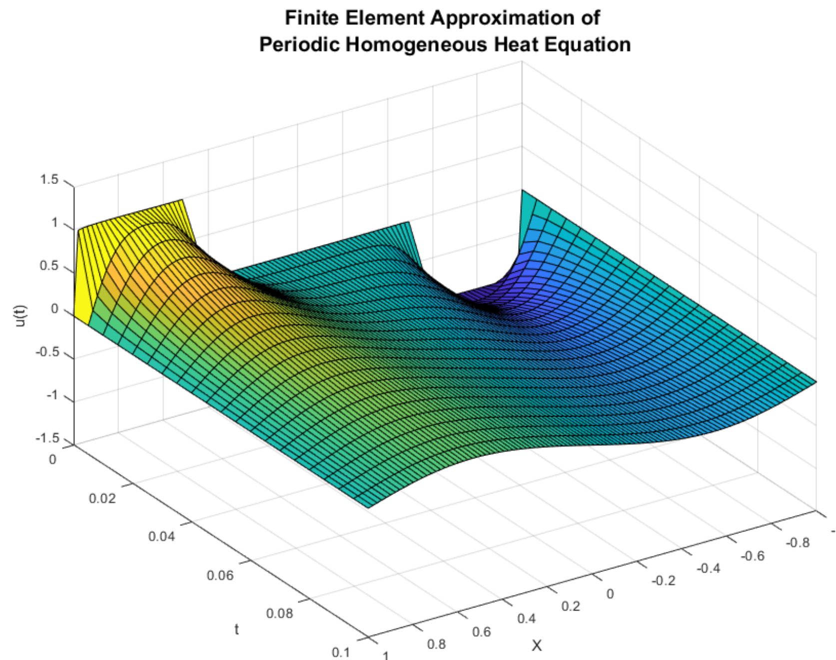

Figure 1: Surface Plot of Heat Distribution within the Bar

Defining Network in PyTorch

The library used to run the neural network and perform the auto-differentiation necessary to minimize this cost-function was PyTorch. The literature recommended using a neural network with 6 hidden layers, 100 hidden neurons per layer and a hyperbolic tangent (tanh) activation function. This model was produced with the following:

```
# Define the neural network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 100)
        self.fc2 = nn.Linear(100, 100)
```

```
        self.fc3 = nn.Linear(100, 100)
        self.fc4 = nn.Linear(100, 100)
        self.fc5 = nn.Linear(100, 100)
        self.fc6 = nn.Linear(100, 100)
        self.fc7 = nn.Linear(100, 1)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = torch.tanh(self.fc3(x))
        x = torch.tanh(self.fc4(x))
        x = torch.tanh(self.fc5(x))
        x = torch.tanh(self.fc6(x))
        x = self.fc7(x)
        return x

net = Net().to(device)
```

Subsequently, a custom cost function had to be defined based on the second section. This was defined with the following:

```
# Define the loss function
def custom_loss(u0, x0, x):
    criterion = nn.MSELoss()
    # Calculating the initial condition loss
    MSE_0 = criterion(net(x0), u0)

    # Calculating the Boundary Loss
    N = 100
    xn1 = get_boundary_vals(N, -1, 0, 0.1).to(device)
    x1 = get_boundary_vals(N, 1, 0, 0.1).to(device)
    MSE_b_f = criterion(net(xn1), net(x1))

    # Calculating the Boundary Loss (Derivative)
    upredn1 = net(xn1)
    du_din = torch.autograd.grad(upredn1, xn1, grad_outputs=torch.
   ones_like(upredn1).squeeze(-2), retain_graph=True, create_graph = True)
   [0]
    du_dx_n1 = du_din[:,1]

    upred1 = net(x1)
    du_din = torch.autograd.grad(upred1, x1, grad_outputs=torch.ones_like(
   upred1).squeeze(-2), retain_graph=True, create_graph = True)[0]
    du_dx_1 = du_din[:,1]

    MSE_b_d = criterion(du_dx_n1, du_dx_1)

    # Calculating the PDE Equation Loss
    upred = net(x)
    du_din = torch.autograd.grad(upred, x, grad_outputs=torch.ones_like(
```

```
    upred), retain_graph=True, create_graph = True)[0]
27      #du_din = torch.tensor(du_din, requires_grad=True)
28      du_dt = du_din[:,0]
29      d2u_din2 = torch.autograd.grad(du_din, x, grad_outputs=torch.ones_like
    (du_din), retain_graph=True)[0]
30      d2u_dx2 = d2u_din2[:,1]
31
32      MSE_f = criterion(du_dt, d2u_dx2)
33
34      #return MSE_0 + 0.5*MSE_b_f + 0.5*MSE_b_d + MSE_f
35      return MSE_0 + MSE_b_f + MSE_b_d + MSE_f
```

Lastly, the network needed to be run for many epochs to converge. This was accomplished with the following code. I'm leaving out a lot of the boring nut-and-bolt code here but I wanted to include the important code in the report.

```
1  # Define the loss function and optimizer
2  optimizer = optim.Adamax(net.parameters(), lr=1e-4)
3
4  # Train the network
5  for epoch in range(10000):
6      optimizer.zero_grad()
7      outputs = net(x)
8      loss = custom_loss(u0, x0, x)
9      loss.backward()
10     optimizer.step()
11
12     if epoch % 100 == 0:
13         print(f"Epoch {epoch}, loss = {loss.item()}")
```

Training Model and Evaluating Results

After 10,000 epochs, the model converged to a final cost function on the order of 10^{-5}. The first test would be to simulate the initial conditions to see whether the PINN matches the initial conditions.

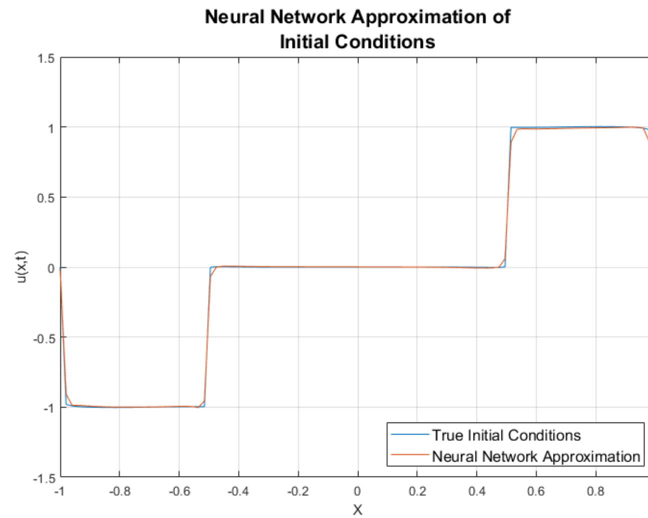

Figure 2: Comparison of initial conditions with Neural Network and True Initial Conditions.

So good so far! The next step would be to see if the surface plot of the neural network approximation looks anything like the FE Approximation. This is depicted in Figure 3.

Looks pretty good to me! However, what we're really here for is the error. We can plot this as a surface plot as well in Figure 4.

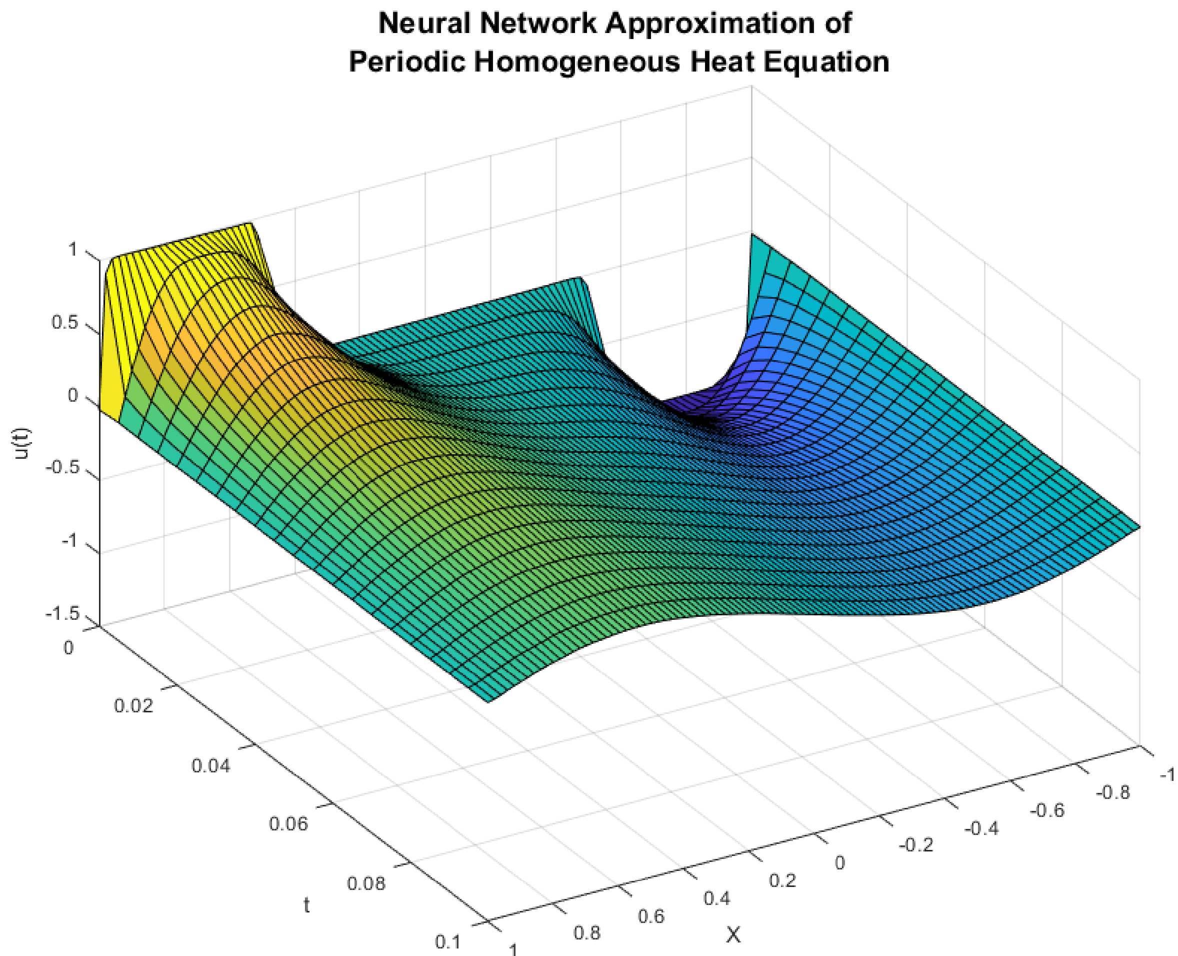

Figure 3: Neural Network Approximation of Heat Equation with Periodic B.C.'s.

As shown in the Figure 4, the PINN has some trouble in the beginning when the dynamics are changing quite a bit with respect to time, however the error gets quite small once the behavior is a bit more stable. This said, this shows a maximum error of less than 0.1.

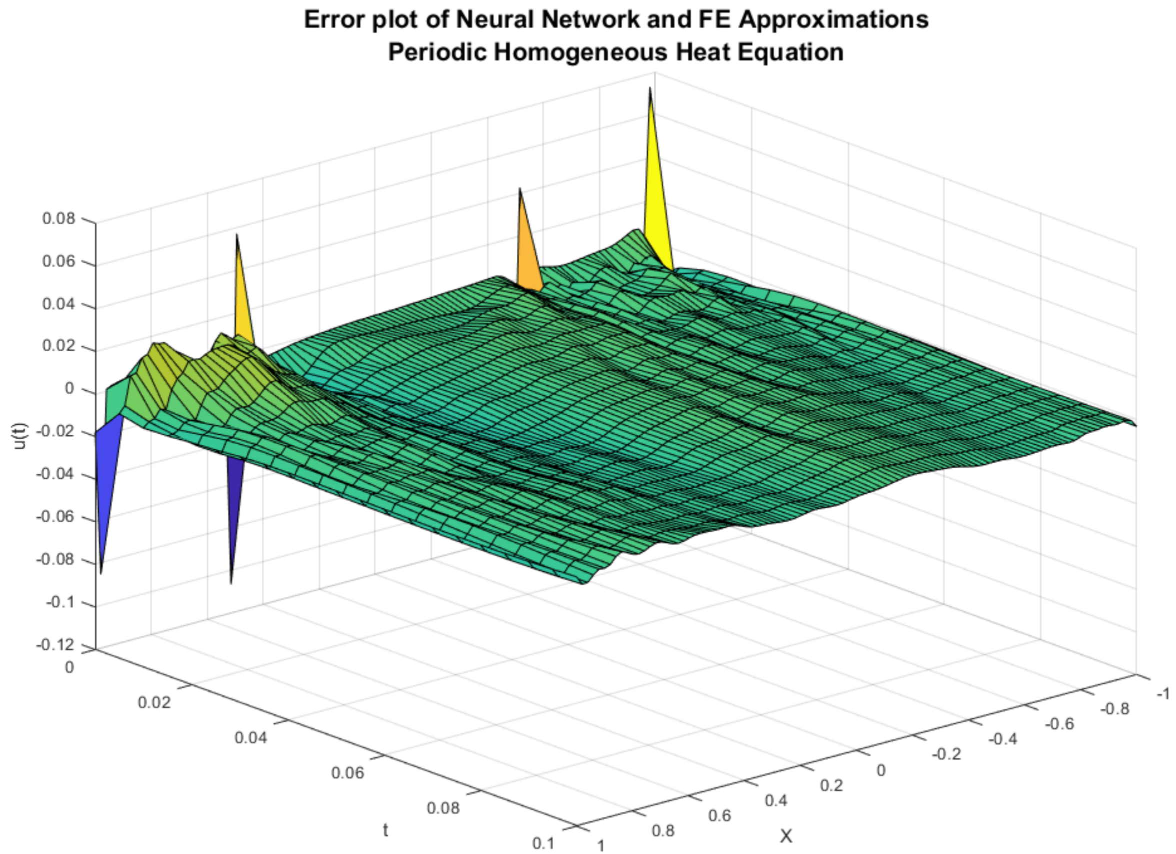

Figure 4: Error between Neural Network and Finite Element Approximations.

Conclusion and Discussion of Results

In this paper, we demonstrated how neural networks can act as universal function approximations to approximate the solutions to partial differential equations. This was accomplished through the potent automatic differentiation capabilities present in Neural Networks to efficiently compute the gradient. Auto-differentiation involves a computational graph of each tiny step involved in taking some input to some output and utilizing the chain rule to compute the derivative at each teeny step.

With this algorithm, we can optimize some seriously complex cost functions, making this an exceedingly useful optimization tool. I only became aware of automatic differentiation within the past year. My guess is this will be used to solve quite a bit of open computational problems in the coming years.

In this paper, we solved a cost function that minimized the three stipulations of the solution to a partial differential equation: 1.) The function matches the initial conditions when $t = 0$ 2.) the function matches the boundary conditions at all times 3.) the function satisfies the differential equation.

Then, this cost function was implemented in PyTorch (not a trivial task, by the way - very headache inducing) and a reasonably sized neural network with about 60,000 parameters was trained to approximate the solution to the differential equation. And after many hours of failure and crazy debugging, it actually did it pretty accurately! I am sure if I had trained it for longer, the error could have been lowered, but it did prove itself as an effective tool for approximating the solution to partial differential equations without having to do extensive math to compute the spectral or finite element method solutions.

References

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed Neural Networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," Journal of Computational Physics, vol. 378, pp. 686–707, 2019.

[2] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," Neural Networks, vol. 2, no. 5, pp. 359–366, 1989.