William Mahnke
Paul Aztberger
MATH 132A
March 18, 2025

Final Project Report - Using PINNs to Solve PDEs

Our project aimed to create a physics-informed neural network (PINN) to approximate the solution to the two-dimensional heat equation over a finite boundary and investigate how different optimizers affect the loss of the network. Traditionally, more traditional numerical methods such as finite difference methods were used to computationally approximate solutions to systems modeled by complex PDEs, such as the Navier-Stokes equation. PINNs were introduced by Professor George Kariadakis as a neural network that accounts for physical constraints and initial conditions, two elements crucial for studying systems modeled by PDEs. Additionally, PINNs serve as a better alternative to methods for solving PDEs because of their ability to predict solutions on finer resolutions of space without retraining the model and their use of automatic differentiation.

For our project we specifically dealt with the two-dimensional heat equation, traditionally modeled as:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), (x, y) \in \Omega, t \geq 0 \qquad \Omega = \{(x, y) \in \mathbb{R}^2 : |x| \leq 1, |y| \leq 1\}$$

We chose to model the heat equation over a square of side length two centered at the origin. This region is equivalent to modeling the one-dimensional heat equation on a rod, a problem we all have been introduced to in our previous studies of PDEs. We did research alternative regions to model the equation on including the unit circle, but decided on the unit square because it would be easier to computationally model boundary conditions. For our initial boundary conditions we used Dirichlet boundary conditions for the perimeter of the region and an initial function inspired by my previous work on using neural networks to solve PDEs:

$$u(x, y, 0) = \exp\left(-\frac{1}{1 - (x^2 + y^2)}\right), (x, y) \in \Omega, t \geq 0$$
$$u(x, y, t) = 0, (x, y) \in \partial\Omega, t \geq 0$$

During the duration of the project we used a GitHub repository to collaboratively develop scripts for the different stages of model creation and training. We divided the scripts for creating the training data, the loss function, and the model as well as scripts for training the model and displaying its results. Across all of our scripts, we leverage packages including numpy, torch, matplotlib, and IPython to streamline the computations and create effective visualizations. To generate our data, we divided the sampling of points into three separate regions; the interior, the boundary, and the initial condition. For sampling points on the boundary, we split up the region further into the four sides of the square. The result from our sampling is three tensors of x, y, and t values which can be combined and fed into the model for training. We decided to separate the

sampled points by region so we could experiment with how the volume of sampled points from each region affects the model's training. Additionally, it makes sense to sample more points from the interior than the boundary rather than a uniform amount for each region.

Given our original equation, the initial heat distribution, and the boundary temperature, we analytically developed our loss function before implementing it in model.py. The loss of the model's solution can simply be modeled by the discrete mean squared error formula, adjusted to account for the initial and boundary conditions (shown in the image to the right). When calculating the loss for the model's prediction on the sampled interior points, we leveraged torch's autograd function to compute partial derivatives, an advantage to PINNs outlined earlier. The separation of the region when sampling our points and in calculating the loss of the model is reflected in our code, computing the loss of the three regions before aggregating it in the end.

$$L_{\text{int}} = \frac{1}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} \left(\alpha \Delta u \left(x_{\text{int}}^{(i)}, y_{\text{int}}^{(i)}, t_{>0}^{(i)} \right) - u_t \left(x_{\text{int}}^{(i)}, y_{\text{int}}^{(i)}, t_{>0}^{(i)} \right) \right)^2$$

$$L_{\text{bdry}} = \frac{1}{N_{\text{bdry}}} \sum_{i=1}^{N_{\text{bdry}}} \left[u \left(x_{\text{bdry}}^{(i)}, y_{\text{bdry}}^{(i)}, t_{>0}^{(i)} \right) \right]^2$$

$$L_{\text{init}} = \frac{1}{N_{\text{int}}} \sum_{j=1}^{N_{\text{int}}} |u(x_{\text{int}}^{(j)}, y_{\text{int}}^{(j)}, 0) - f(x_{\text{int}}^{(j)}, y_{\text{int}}^{(j)})|^2$$

where $f(x, y) = u(x, y, 0)$, the initial heat distribution

$$L = L_{\text{int}} + L_{\text{bdry}} + L_{\text{init}}$$

Our model.py includes the architecture of our PINN. We designed our model to be flexible, allowing us to adjust the number of hidden layers, the number of hidden neurons per layer, and our activation functions. The first and last layers of our model always stayed the same, a linear layer taking in tensor with x, y, and t coordinates and a linear layer condensing the result to one value respectively. However, building in this flexible functionality made it easier to experiment with different model structures efficiently and adjust our model accordingly.

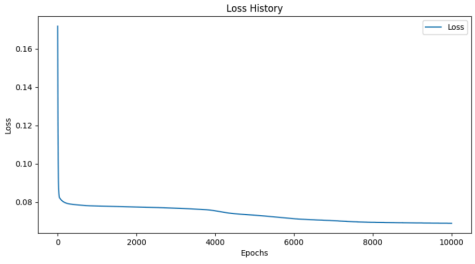
The use of our sampling, loss function, and model architecture scripts culminates in our training.py file which features a Training class. We used object-oriented programming to streamline the data sampling to model the training process as well as efficiently display the loss of the model over epochs. We designed the initializing of the Training class to specify the number of epochs, the optimizer the model uses, and our important parameters before training the model. The default optimizer for our model's backpropagation and weight adjustment is ADAM, an optimizer we were introduced to in class. One of the model parameters we included was a learning rate scheduler to adjust the learning rate if there ever isn't substantial improvement in the model's loss. The Training class also includes our visualization of the model's loss over epochs so we can quickly visualize the model's performance given different optimizers, hidden layers, and hidden neurons.

Finally, our Network.py further streamlines the model pipeline from sampling to model training to evaluating performance. In our Wrapper class we specify how many points we want to sample, the model optimizer, the activation function, the number of epochs, and the learning rate. While we chose the diffusivity term to be one for our project, we included the option to adjust the diffusivity term. When run, the file will sample points from the square region, train the model over the specified number of epochs, and display the model's loss every thousand epochs. After training, the script will display the model's loss over all epochs with the option of also displaying graphs to show the model's prediction at different times. One visual is an animation that slowly shows the model's predictions at each time step while the other includes a slider so

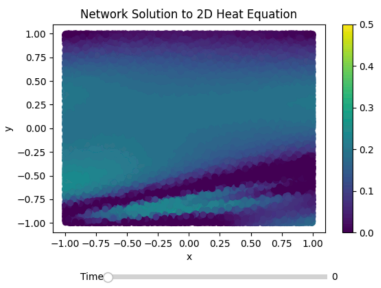
the user can look at and investigate specific time intervals with more ease. While the visual with the slider is better at conveying the diffusion of temperature over the region, the functionality of the visual worsens as the number of sampled points increases.

During our discussion of machine learning and optimization throughout the quarter, we discussed how optimizers such as ADAM or SGD are effective methods to minimize functions. A drawback to implementing these optimizers was their reliance on gradients to adjust the weights of models, but that problem was solved by our use of torch.autograd. Additionally, our discussion of implementing machine learning included the Universal Approximation Theorem which guarantees our model's ability to accurately approximate continuous functions like the solution to our heat equation. As mentioned at the beginning of the report and further supported by our research, PINNs have demonstrated their ability to accurately model complex dynamic systems while using less computational resources than alternative strategies.

To optimize our model's prediction of the solution, we ran numerous tests adjusting previously mentioned parameters. Our number of sampled data points varied from slightly more than three thousand to over a million. We tested different optimizers including SGD, ADAM, ADAMax, RMSprop, and NADAM. And we chose between the sigmoid, tanh, and elu activation functions. It's also important to note while we implemented the option to adjust the number of epochs, the models were trained over ten thousand epochs. From our tests, we found the best optimizer changed based on the activation function. When using the sigmoid function, the models with the ADAM and ADAMax optimizer performed the best with a loss of 0.0833 after 10000 epochs. When comparing models using ADAM or ADAMax with models using different activation functions, we found the model using tanh outperformed elu and sigmoid with a loss of 0.0802 after the standard number of epochs. With the optimal combination of using ADAM/ADAMax and the tanh function, we investigated how the number of sampled points would affect performance. To our surprise, the model using 130 thousand sampled points outperformed the model using 1.3 million points, with the model using 6500 points performing the worst. Models training on less sampled points such as 3250, 6500, and 13000 aren't able to capture the complexity of the system while the model with over a million points is overfitting rather than converging to the solution. So our tests concluded with a model using ADAM optimizer, the tanh activation function, and 130 thousand sampled points (100 thousand for the interior, 10 thousand for the initial distribution, and 20 thousand for the boundary condition).



The



graph on the left shows the change in the model's loss over the 10 thousand

epochs. The model initially shows substantial improvement in the loss before slowing down and converging to the final loss of 0.0689 after 10 thousand epochs. The shallower dips in the loss for later epochs demonstrate how the learning rate of the model adjusted when loss was decreasing substantially. The graph on the right shows the model's prediction for the initial heat distribution. When run in a Python IDE, the slider can be used to display the model's prediction at points in time.

From our testing phase, we also discovered some other insights into how optimizers and activation function after the model's performance. We found that ADAM and ADAMax performed very similarly with ADAM performing slightly better. This can likely be attributed to ADAM utilizing both the mean and variance of gradients compared to ADAMax's approach of just using the gradients' infinity norms. Additionally, we found the sigmoid activation function performed the worst out of the three, contrary to the results from a paper we found during our research (referenced below). We reasoned this is because the gradients near the boundary which were computed using sigmoid are small and thus not helpful when adjusting the weights.

To summarize our results, we found that ADAM and ADAMax were the best optimizers (with ADAM slightly outperforming) while the tanh activation function consistently outperformed the elu and sigmoid functions. While we didn't find an exact optimal number of data points to sample, our findings indicate the best option is about 100 thousand. Further research into this topic could include a variety of different avenues. The heat equation is a relatively simple model which serves as a good starting point for computationally solving dynamic systems. Further research into the area could be investigating whether optimizers' performances or other parameters mentioned, are consistent across multiple dynamic systems. Within the context of the heat equation, we could also explore alternative structures for our PINN to improve the model's loss to higher degrees of magnitude. Finally, since we couldn't narrow in on an optimal number of sampled data points, we could investigate the tradeoff of increasing the number of sampled points with the computational cost and performance of the model.

Sources

1. Maczuga, P. et al (2024, February 19). Physics informed neural network code for 2D transient problems (pinn-2dt) compatible with Google Colab. https://arxiv.org/html/2310.03755v2
2. Wikimedia Foundation. (2025, January 19). *Physics-informed Neural Networks*. Wikipedia. https://en.wikipedia.org/wiki/Physics-informed_neural_networks