1 Numerical Solution of PDE using PINN

1.1 Setting up the network

We want to calculate the numerical solution of the following system of partial differential equations:

$$\begin{cases} u_t - u_{xx} = f & x \in (0,1) \text{ and } t \in (0,T) \\ u(0,x) = \sin(2\pi x) \\ u(t,0) = 0 \\ u_x(t,1) = 2\pi e^{-t} \end{cases}$$

Its analytical solution is $u(t,x) = e^{-t}\sin(2\pi x)$ and $f = 4e^{-t}\pi^2 \sin(2\pi x) - e^{-t}\sin(2\pi x)$.
The general PINN structure for this equation is shown in figure 1:

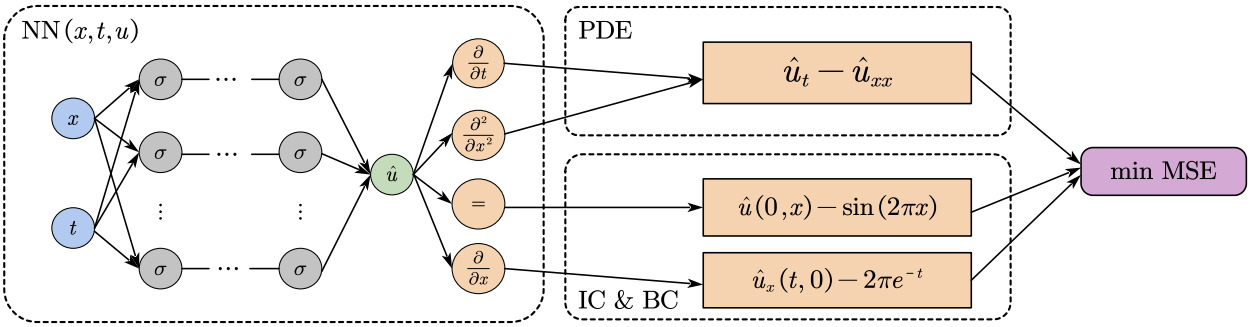


Figure 1: PINN consists of a neural network and loss calculation module.

The primary goal of PINN is to optimize the network by employing the starting values, boundary conditions, and equation definitions in the differential equations to cause the network to converge. This is made possible by the automated differentiation mechanism offered by modern packages for neural network architecture.

Because of the simplicity of PyTorch's implementation of automatic differentiation and network structure, it is used to build the network structure. The implementation of neural network is shown in code snippet 1.

```python
import torch
import torch.nn as nn

class Wave(nn.Module):
    """
    Define the neural network,
    params: layer <int>, neurons <int>
    """

    def __init__(self, layer: int = 5, neurons: int = 20):
        # Input layer
        super(Wave, self).__init__()
        self.linear_in = nn.Linear(2, neurons)
        # Output layer
        self.linear_out = nn.Linear(neurons, 1)
        # Hidden Layers
        self.layers = nn.ModuleList(
            [nn.Linear(neurons, neurons) for i in range(layer)]
        )
        # Activation function
        self.act = nn.modules.Tanh()  # How about LeakyReLU? Or even Swish?

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.linear_in(x)
        for layer in self.layers:
            x = self.act(layer(x))
        x = self.linear_out(x)
```

```
        return x
```

Listing 1: Definition of the neural network using PyTorch

Let us define $f := u_t - u_{xx}$. f can be simply defined in PyTorch as in code snippet 2. Notice that in PyTorch, it is able to calculate high order derivatives by repeatedly using `torch.autograd.grad`.

```
def f(model, x_f, t_f, u_f):
    """
    This function evaluates the PDE at collocation points.
    """
    u = model(torch.stack((x_f, t_f), axis=1))[:, 0]  # Concatenates a seq of tensors along a
     new dimension
    u_t = derivative(u, t_f, order=1)
    u_xx = derivative(u, x_f, order=2)
    return u_t - u_xx - u_f

def derivative(dy: torch.Tensor, x: torch.Tensor, order: int = 1) -> torch.Tensor:
    """
    This function calculates the derivative of the model at x_f
    """
    for i in range(order):
        dy = torch.autograd.grad(
            dy, x, grad_outputs=torch.ones_like(dy), create_graph=True, retain_graph=True)[0]
    return
```

Listing 2: Definition of the output using PyTorch

We use MSE to measure the error. Define MSE_{ic}, MSE_{bc} and MSE_f for initial condition, boundary condition and collocation points, respectively.

The shared parameters between the neural networks $u(t, x)$ and $f(t, x)$ can be learned by minimizing the mean squared error loss

$$\text{MSE} = \text{MSE}_f + \text{MSE}_{ic} + \text{MSE}_{bc},$$

where

$$\text{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| f(t^i, x^i) \right|^2,$$

and

$$\begin{cases} \text{MSE}_{ic} = \frac{1}{N_{ic}} \sum_{i=1}^{N_{ic}} \left| u(t^i, x^i) - u \right|^2 \\ \text{MSE}_{bc} = \frac{1}{N_{bc}} \sum_{i=1}^{N_{bc}} \left| u(t^i, x^i) - u \right|^2 \end{cases}$$

Correspondingly, MSEs are implemented in PyTorch in code snippet 3.

```
def mse_f(model, x_f, t_f, u_f):
    """
    This function calculates the MSE for the PDE.
    """
    f_u = f(model, x_f, t_f, u_f)
    return (f_u ** 2).mean()


def mse_ic(model, x_ic, t_ic, u_ic):
    """
    This function calculates the MSE for the initial condition.
    u_ic is the real values
    """
    u = model(torch.stack((x_ic, t_ic), axis=1))[:, 0]
    return ((u - u_ic) ** 2).mean()


def mse_bc(model, l_t_bc, u_t_bc):
    """
    This function calculates the MSE for the boundary condition.
    """
    l_x_bc = torch.zeros_like(l_t_bc)
    l_x_bc.requires_grad = True
    l_u_bc = model(torch.stack((l_x_bc, l_t_bc), axis = 1))[:, 0]
    mse_dirichlet = (l_u_bc ** 2).mean()
```

```
    u_x_bc = torch.ones_like(u_t_bc)
    u_x_bc.requires_grad = True
    u_u_bc = model(torch.stack((u_x_bc, u_t_bc), axis=1))[:, 0]
    u_x_b_upper = derivative(u_u_bc, u_x_bc, 1)
    mse_neumann = (((2 * np.pi * torch.exp(-u_t_bc)) - u_x_b_upper) ** 2).mean()
    return mse_dirichlet + mse_neumann
```

Listing 3: Definition of MSEs using PyTorch

The default hyper parameters are listed in table 1.

Table 1: Settings of hyper parameters.

Hyper parameter	value
# Hidden layer	6
# Neurons per layer	50
# Initial & boundary points	50
# Collocation points	9000
# epochs	1000
# Learning rate[a]	1
Optimization method	LBFGS
Activation function	Tanh

[a] The learning rate is for LBFGS.

1.2 Getting the prediction

Figure 2 and 3 show the predicted solution of this equation under the hyper-parameters in Table 1.

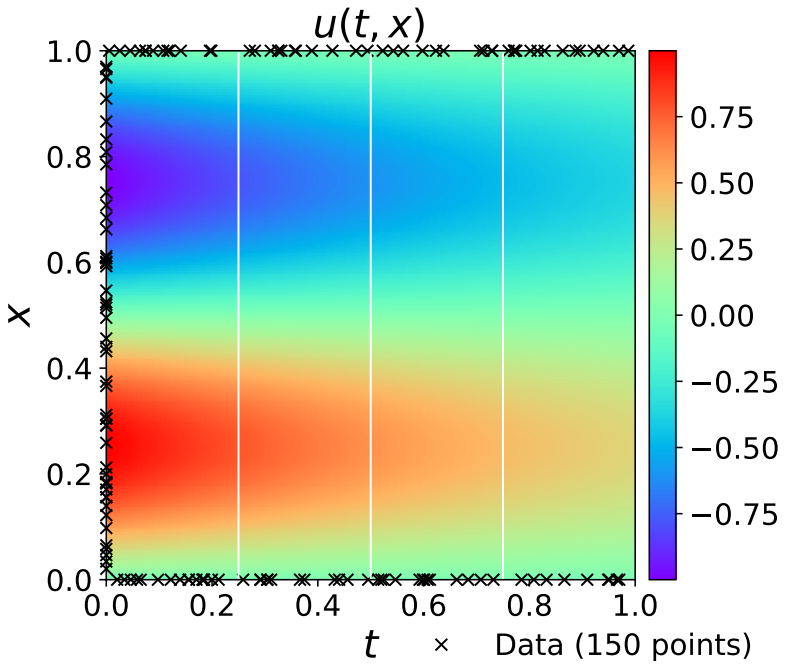


Figure 2: Predicted solution $u(t, x)$ along with the initial and boundary training data. The relative $\mathbb{L}_2$ error for this case is 1.04×10^{-3}. Model training took 21.71s with 1000 epochs.

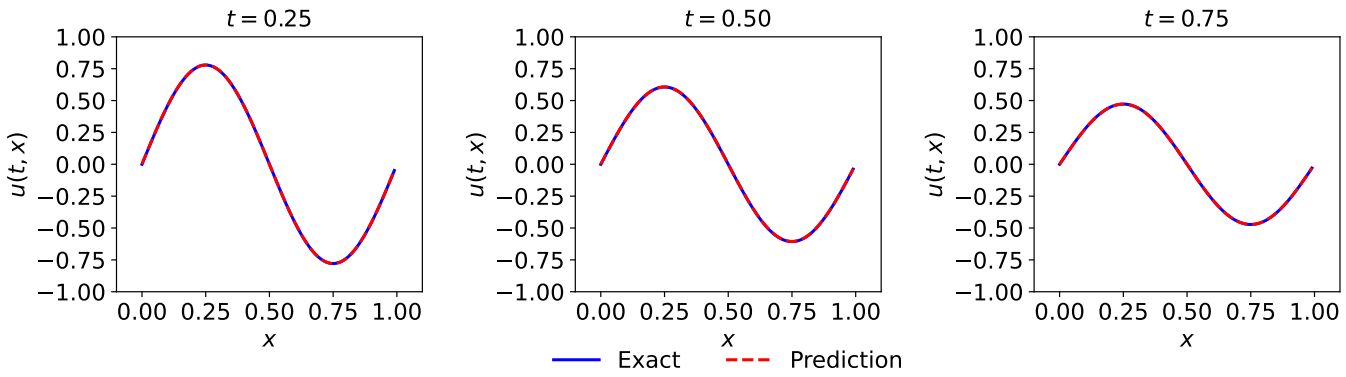


Figure 3: Comparison of the predicted and exact solution corresponding the the three temporal slices depicted by the white vertical lines in Figure 2.

The errors are shown in Figure 4.

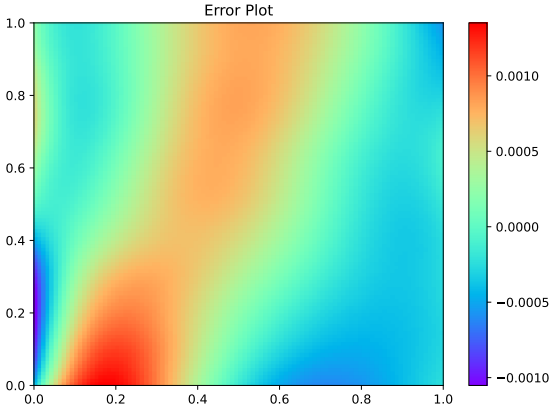
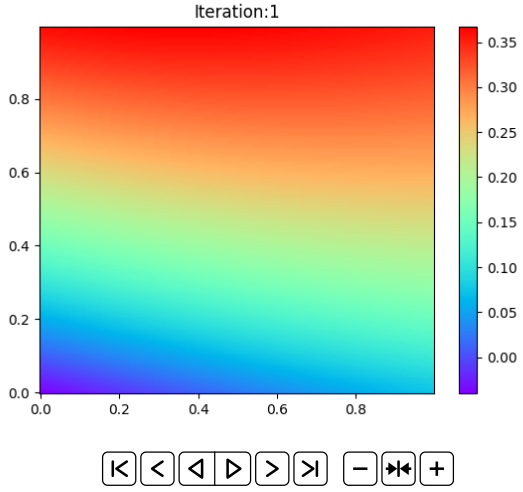


Figure 4: Error between the predicted and exact solution.



Animation only available in Adobe Acrobat.

1.3 Investigation of hyper-parameters

The investigated parameters are listed below:
Network Structure:

(a) Number of layers
(b) Neurons per layer

Optimization:

(a) Optimization method: LBFGS, Adam, Ada, SGD, RMSProp
(b) Activation function: Tanh, GeLU, Mish, SoftPlus

Data:

(a) Points on initial condition
(b) Points on boundary condition
(c) Collocation points

We shall check the relative $\mathbb{L}_2$ by varying these hyper-parameters and record them using TensorBoard.

Table 2 gives the relative error $\mathbb{L}_2$ between the predicted and the exact solution using different number of hidden layers (2, 4, 6, 8, 10) and neurons per hidden layer (10, 30, 50, 70, 90). The optimal scheme is 6 hidden layer and 90 neurons with relative error of 0.046% and loss of 9.6×10^{-6}.

Table 2: Relative error (%) and loss (10^{-4}) between the predicted and the exact solution under different number of hidden layers N_l and neurons per layer N_n.

N_n \ N_l	2	4	6	8	10
10	0.846 (6.779)[a]	0.664 (6.508)	1.079 (6.660)	4.328 (72.123)	4.512 (239.600)
30	0.258 (4.110)	0.245 (1.115)	0.300 (1.234)	0.297 (2.922)	0.255 (1.347)
50	0.252 (2.183)	0.176 (1.156)	0.238 (1.550)	0.128 (1.916)	0.279 (4.296)
70	0.249 (3.240)	0.104 (0.328)	0.088 (0.437)	0.242 (0.736)	0.199 (0.698)
90	0.581 (3.865)	0.335 (1.084)	**0.046** (**0.096**)[b]	0.155 (0.158)	0.112 (0.460)

$N_i = 100$, $N_c = 5000$, epoch=1000 and lr=1, using LBFGS.
[a] The result are show in the form *error* (*loss*)
[b] The smallest error are in bold font.

Table 3 gives the relative error by varying the number of initial,(each) boundary points (30, 40, 50, 60, 70) and collocation points (1000, 3000, 5000, 7000, 9000). The optimal scheme for minimizing relative error is 30 initial points, 60 boundary points (a total of 90 points) and 9000 collocation points with relative error of 0.038% and loss of 2.142×10^{-5} while for minimizing loss is 70 initial points, 140 boundary points (a total of 210 points) and 3000 collocation points with relative error of 0.1% and loss of 1.339×10^{-5}.

Table 3: Relative error (%) and loss (10^{-5}) between the predicted and the exact solution under different number of initial and boundary points N_i and collocation points N_c.

N_i \ N_c	30	40	50	60	70
1000	0.084 (5.104)	0.095 (2.393)	0.150 (4.968)	0.085 (1.845)	0.065 (2.257)
3000	0.090 (2.666)	0.246 (2.942)	0.087 (2.396)	0.211 (3.664)	0.100 (**1.339**)
5000	0.185 (3.591)	0.100 (2.502)	0.051 (1.644)	0.173 (5.367)	0.114 (2.311)
7000	0.163 (4.331)	0.185 (2.108)	0.173 (3.111)	0.197 (4.339)	0.082 (2.638)
9000	**0.038** (2.142)	0.133 (2.194)	0.121 (2.546)	0.049 (1.994)	0.159 (3.470)

$N_l = 4$, $N_n = 40$, epoch=1000 and lr=1, using LBFGS.

Table 4 gives the relative error by changing the learning epochs (500, 750, 1000, 1250, 1500) and learning rate (0.01, 0.1, 0.5, 0.1, 2, 5). The optimal schemes are found when learning rate is 1 and epoch are 1250 or 1500 with relative error of 0.02% and loss of 1.280×10^{-5}.

Table 4: Relative error (%) and loss (10^{-4}) between the predicted and the exact solution under different epochs E and learning rate.

E \ lr	0.01	0.1	0.5	1	2	5
500	2.407 (50.059)	0.987 (12.172)	0.404 (8.446)	0.192 (0.842)	0.283 (4.592)	0.279 (2.868)
750	1.052 (16.980)	0.441 (2.743)	0.125 (1.866)	0.077 (0.316)	0.149 (2.477)	0.286 (1.159)
1000	0.722 (5.912)	0.276 (1.345)	0.125 (0.634)	0.051 (0.164)	0.139 (2.078)	0.222 (0.557)
1250	0.441 (3.648)	0.240 (1.017)	0.081 (0.337)	**0.020** (**0.128**)[a]	0.192 (1.391)	0.216 (0.301)
1500	0.494 (2.457)	0.204 (0.839)	0.076 (0.166)	**0.020** (**0.128**)[a]	0.134 (0.954)	0.195 (0.244)

$N_l = 6$, $N_n = 50$, $N_i = 50$ and $N_c = 9000$, using LBFGS.
[a] Both are smallest.

Then, as shown in Table 5 and Figure 5, we investigated the impact of several optimization methods on convergence speed and final error. In terms of convergence speed and final error, the second-order method L-BFGS offers an advantage over alternative first-order methods. [1] It is worth noting that first-order algorithms, like Adam, are effective of performing error-insensitive computations.

Table 5: Final loss and error (%) between the predicted and the exact solution using different optimization methods

Attribute \ Opt. Method	L-BFGS	SGD	Ada	RMSProp	Adam
Loss	$\mathbf{3.315 \times 10^{-5}}$	2.656	0.140	7.48×10^{-2}	0.223
Rel. Error	**0.105**	147.430	6.352	17.950	7.596

$N_l = 6$, $N_n = 50$, $N_i = 50$ and $N_c = 9000$, using Tanh.

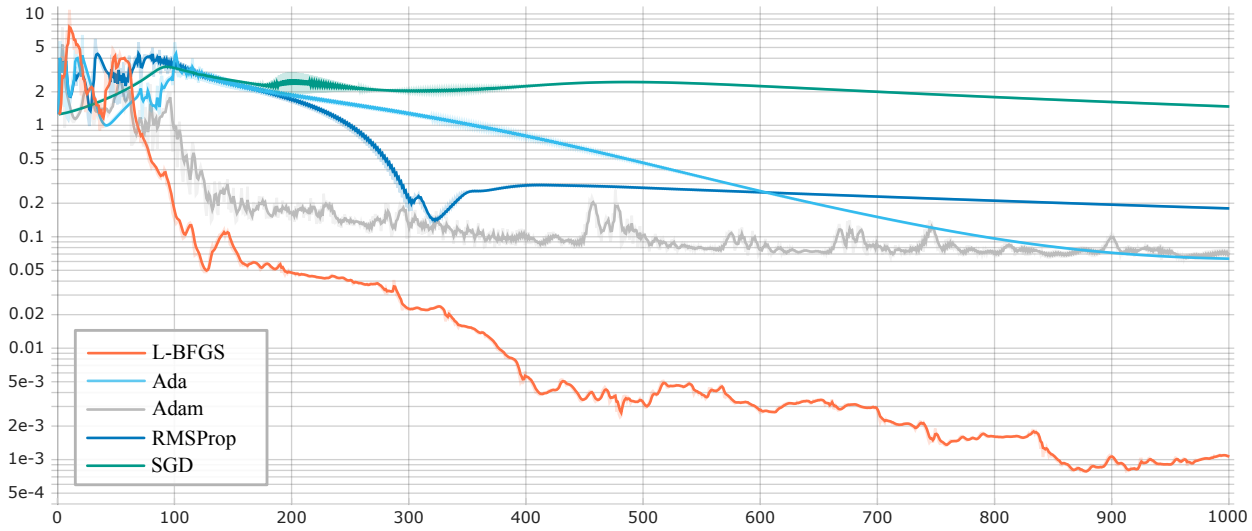


Figure 5: Relative error using different Optimization method. L-BFGS outperforms among other optimization algorithms.

Finally, as seen in Table 6 and Figure 6, we study the convergence rate and final error/loss using various activation functions with default settings. According to the results, Mish has the fastest convergence and best relative error and Tanh has smallest loss comparing to other activation functions. On other tasks, Mish generally outperforms than other activation functions. [2]

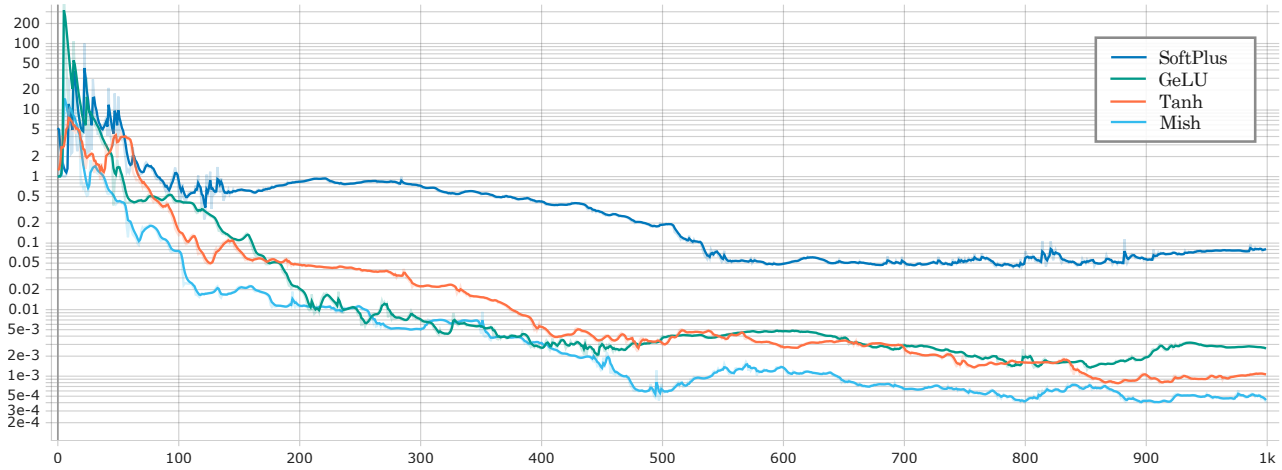


Figure 6: Relative error under different activation functions. Mish outperforms among other activation functions.

Table 6: Final loss and error (%) between the predicted and the exact solution using different activation functions

Attribute \ Act. Func.	Tanh	GeLU	Mish	SoftPlus
Loss	$\mathbf{3.315 \times 10^{-5}}$	8.267×10^{-5}	3.433×10^{-5}	7.484×10^{-2}
Rel. Error	0.105	0.255	**0.0490**	8.303

$N_l = 6$, $N_n = 50$, $N_i = 50$ and $N_c = 9000$, using LBFGS.

1.4 Codes

All the code using for this report can be found in the next url: https://github.com/BeteixZ/PINN_from_beginning.

References

[1] Le, Quoc V., et al. On optimization methods for deep learning. *Proceedings of the 28th international conference on international conference on machine learning.* 2011.

[2] Misra, Diganta. Mish: A self regularized non-monotonic activation function. *arXiv preprint arXiv:1908.08681* 2020.