Small PINNs
MATH 598 Final Project

Jiajun Zhang* Emile Petit†

December 24, 2025

Abstract

Physics-informed neural networks (PINNs) have emerged as a powerful framework for learning solutions to partial differential equations (PDEs) by combining data-driven learning with physics-informed terms. We propose a smaller neural network model that is inspired from the Tiny Recursion Model (TRM). Our approach leverages deep supervision to simulate a deep residual network and get similar or better performances in a PINN while having less parameters. Our model and source code is available at Github.

Contents

*Department of Mathematics and Statistics, jiajun.zhang@mail.mcgill.ca, McGill ID: 261105766
†Department of Mathematics and Statistics, emile.petit@mail.mcgill.ca, McGill ID: 261182160

1 Introduction

1.1 An Introduction to PINN

Differential equations (DE) play a significant role in scientific and engineering disciplines due to their ability to model complex phenomena involving functions of multiple variables. Many physical processes including heat conduction, fluid dynamics, wave propagation, etc., can be described by partial differential equations (PDEs). Also due to significant nonlinearities, complicated boundary conditions, etc., some PDEs are notoriously to solve using the standard theory. On the other hand, deep learning has recently emerged as a new paradigm of scientific computing thanks to neural networks' universal approximation and great expressivity.

Physics-informed Neural Networks (PINNs) are a scientific machine learning technique used to solve problems involving the solution of a PDE. PINNs approximate PDE solutions by training a neural network to minimize a loss function, which includes terms reflecting the initial and boundary conditions and the PDE residual at selected points in the domain. Suppose we have the PDE with initial and boundary conditions defined as

$$\begin{cases} f\left(\boldsymbol{x}, t, u_{\boldsymbol{x}}, u_t, \boldsymbol{\lambda}\right) = 0 & \boldsymbol{x} \in \Omega, t \in [0, T] \\ u(\boldsymbol{x}, 0) = h(\boldsymbol{x}) & \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}, t) = g(\boldsymbol{x}, t) & \boldsymbol{x} \in \partial\Omega, t \in [0, T] \end{cases}, \tag{1.1}$$

where $\boldsymbol{\lambda}$ is the parameter vector of the PDE, Ω is the domain, $h(\boldsymbol{x})$ is the initial condition and $g(\boldsymbol{x}, t)$ is the boundary condition.

The solution $u(\boldsymbol{x}, t)$ can be approximated by a neural network $NN(\boldsymbol{x}, t, \boldsymbol{\theta})$ with solution $\widetilde{u}(\boldsymbol{x}, t, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the parameter vector of the neural network and the basic framework is shown in figure 1:

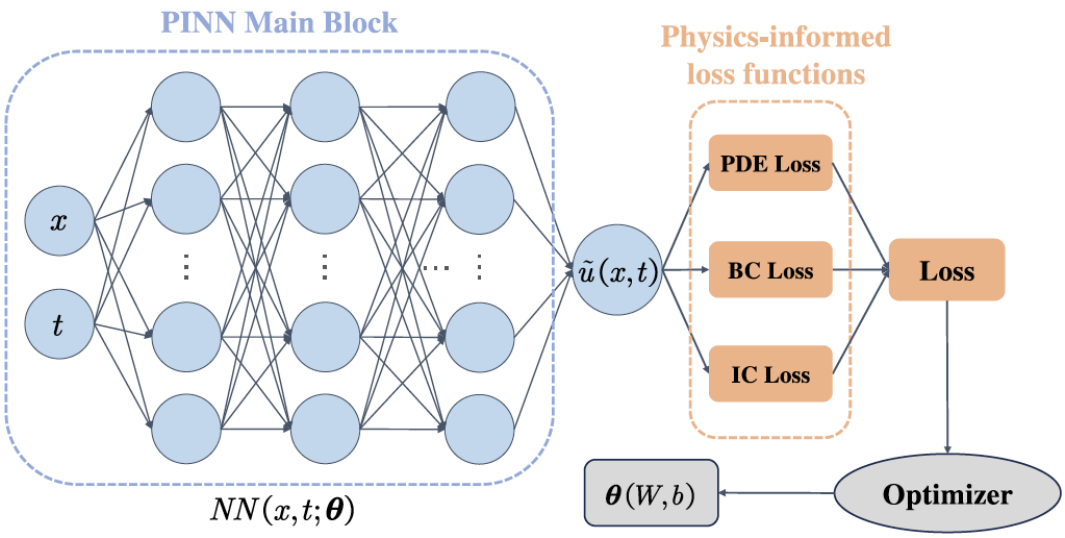

Figure 1: The framework of PINNS. Figure is taken from Luo et. al. [KL25]

The PINN main block is a fully connected, feed-forward neural network, and to better incorporate the PDE into the neural network, the loss function consists of two terms: A data-driven term

that enforces the neural network to fit the available data; a physics-informed term that enforces the PDE constrains. We formulate the loss function as

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{physics}} + \mathcal{L}_{\text{data}}. \tag{1.2}$$

The term $\mathcal{L}_{\text{physics}}$ in (1.2) enforces the PDE constraints by the following

$$\mathcal{L}_{\text{physics}} = w_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}; \mathcal{T}_i),$$

where w_f, w_b, w_i are weights; $\mathcal{T}_f, \mathcal{T}_b, \mathcal{T}_i$ are sets of samples inside the domain Ω, on the boundary conditions $\partial\Omega$, and initial conditions. Each of them is formulated as follows:

$$\begin{aligned}
\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) &= \frac{1}{|\mathcal{T}_f|} \sum_{\boldsymbol{x} \in \mathcal{T}_f} |f(\boldsymbol{x}, t, \boldsymbol{\lambda})|^2, \\
\mathcal{L}_b(\boldsymbol{\theta}, \mathcal{T}_b) &= \frac{1}{|\mathcal{T}_b|} \sum_{\boldsymbol{x} \in \mathcal{T}_b} |\widetilde{u}(\boldsymbol{x}, t, \boldsymbol{\theta}) - g(\boldsymbol{x}, t)|^2, \\
\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}_i) &= \frac{1}{|\mathcal{T}_i|} \sum_{\boldsymbol{x} \in \mathcal{T}_i} |\widetilde{u}(\boldsymbol{x}, 0, \boldsymbol{\theta}) - h(\boldsymbol{x})|^2.
\end{aligned} \tag{1.3}$$

The term $\mathcal{L}_{\text{data}}$ in (1.2) measures the error between the predicted and observed data:

$$\mathcal{L}_{\text{data}} = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} |\widetilde{u}(\boldsymbol{x}, t, \boldsymbol{\theta}) - u(\boldsymbol{x}, t)|^2,$$

where $|\mathcal{T}| = |\mathcal{T}_f| + |\mathcal{T}_b| + |\mathcal{T}_i|$ is the set of all data points. By minimizing the combined loss defined in (1.2) using techniques like gradient descent, PINNs can learn the underlying physics of a system directly from the data, making them powerful tools for solving inverse problems and discovering hidden patterns in complex physical systems [KL25].

1.2 Neural Networks and Activation Functions

Various architectures tailored to address specific challenges have emerged in PINNs. The main architectures covered in PINNs mainly include Multi-layer perceptron (MLP), Convolutional neural networks (CNN), Recurrent neural networks (RNN) and others such as Generative adversarial networks.

The multi-layer perceptron (MLP) is a fundamental architecture within artificial neural networks. The core of this foundation lies the Universal Approximation Theorem which states that a single hidden layer MLP can approximate any continuous function with suitable activations and a finite number of neurons. The input to the MLP consists of a vector $\boldsymbol{x}$ where the elements of $\boldsymbol{x}$ corresponds to different features of the data. For each hidden layer l, $l \in [L]$, we define $\boldsymbol{W}^{(l)}$ as the weight matrix, $\sigma^{(l)}(\cdot)$ as the activation function and $\boldsymbol{e}^{(l)}$ as the bias. We define $\boldsymbol{z}^{(l)}$ as the input vector at layer l, and we have

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l-1)}\Big(\sigma^{(l-1)}(\boldsymbol{z}^{(l-1)})\Big) + \boldsymbol{b}^{(l-1)},$$

and hence the final output $\boldsymbol{y}$ is given by

$$\boldsymbol{y} = \boldsymbol{W}^{(L)}\Big(\sigma^{(l)}(\boldsymbol{z}^{(l)})\Big) + \boldsymbol{b}^{(l)}.$$

Activation functions $\sigma(\cdot)$ are essential in PINNs as they enable the network to learn and represent the complex, non-linear relationships inherent in PDEs [KL25]. Common choices include the hyperbolic tangent (tanh) and sigmoid activation functions. In our simulation study below, we find that GELU (Gaussian Error Linear Units) [HG16] activation function as well as sine activation function also have descent performance.

We consider a neural network trained on velocity data generated from the Navier–Stokes equations. The network takes (t, x, y) as input and outputs the horizontal velocity component $u(t, x, y)$. The neural network is a feed-forward, fully connected network which consists of four hidden layers with 50 neurons per layer. We only change the activation functions in our structure and compare the error given by $\text{err} = |u_{\text{prediction}} - u_{\text{actual}}|$.

Training data are sampled from the cylinder wake flow dataset, where a total of 3000 space–time points are randomly selected from the full domain. The network is trained by minimizing the mean squared error between the predicted and reference velocity values. After training, prediction errors are evaluated on the entire dataset and visualized to assess the spatial distribution of errors across different activation functions. Although the training is data-driven, the model incorporates physical structure through a stream-function formulation consistent with incompressible Navier–Stokes flow. The training error and predicted horizontal velocity component for different activation functions are shown in the figures below.

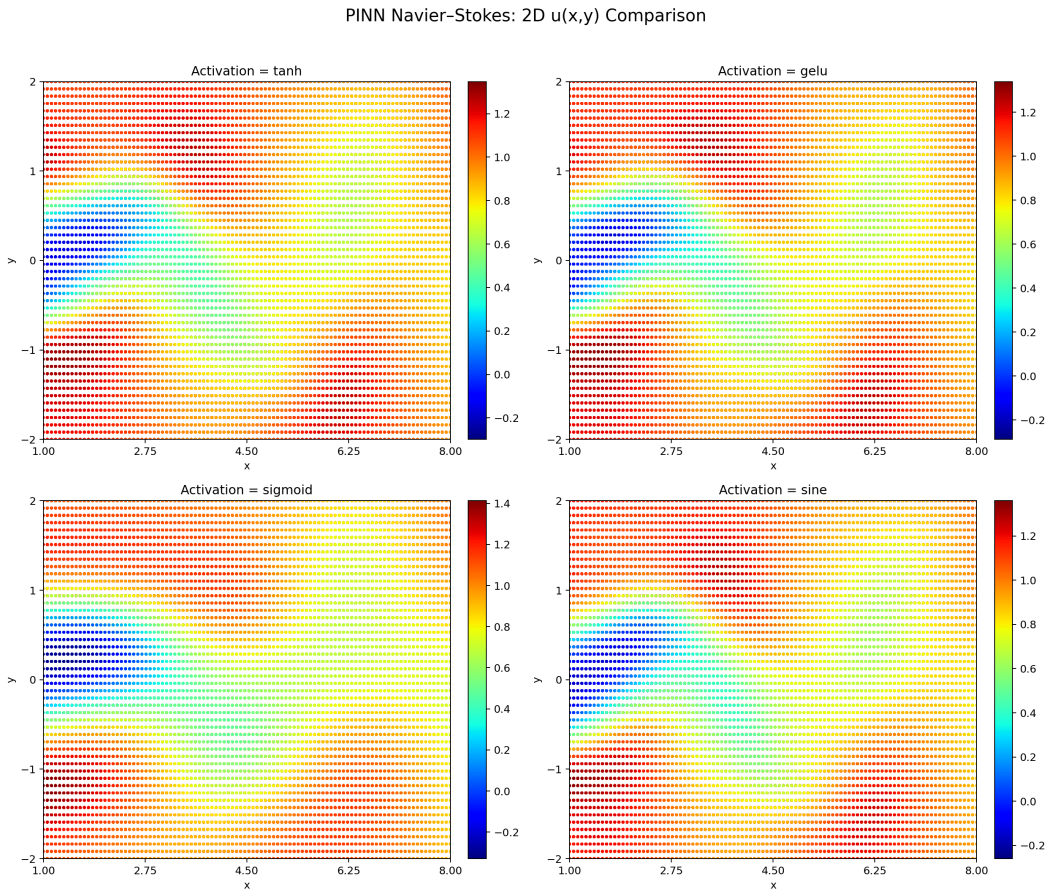

Figure 2: The prediction horizontal velocity with 4 different activation functions.

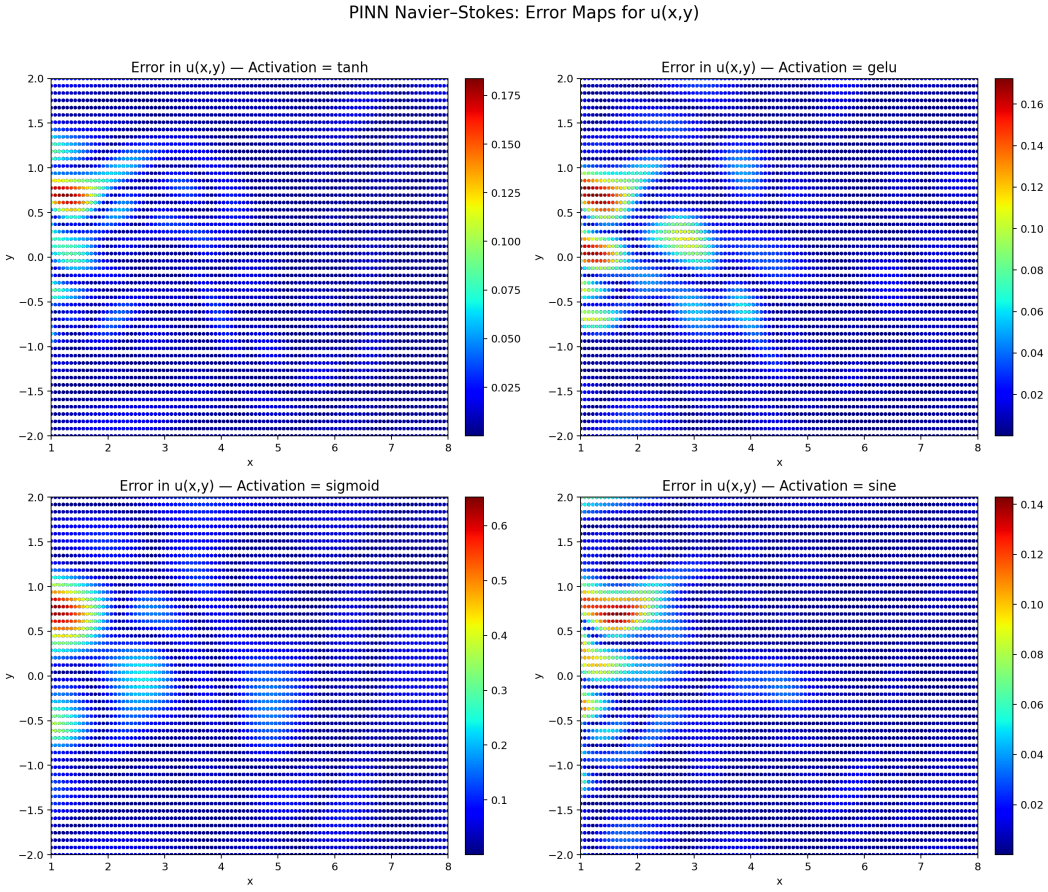

Figure 3: The absolute error of the prediction horizontal velocity with 4 different activation functions.

2 Method

2.1 TRM

The Tiny Recursion Model (TRM) is a novel architecture of neural network that aims to solve tasks that require reasoning by using a very small number of parameters compared to LLMs [JM25]. To do so, a TRM will recursively improve its answer using a single tiny network. This architecture is an improvement over the Hierarchical Reasoning Model (HRM) that improves the generalization power, decreases the number of parameter and gives a clear interpretation of the architecture [JM25]. The TRM model is composed of a single 2 layers network that takes in input x, y, z during the recursion process and only y, z to generate the final answer. In the model, x is the original input, y is a proposed solution and z is a latent space. We can get y, z as follow:

$$z \leftarrow f(x + y + z) \tag{2.1}$$
$$y \leftarrow f(y + z) \tag{2.2}$$

The figure 4 illustrates this process. A single forward pass will execute n recursions to improve the latent space z and then it will produce the final answer y. To improve the effective depth, deep recursion is used. The principle of deep recursion is to reuse the latent space and the proposed

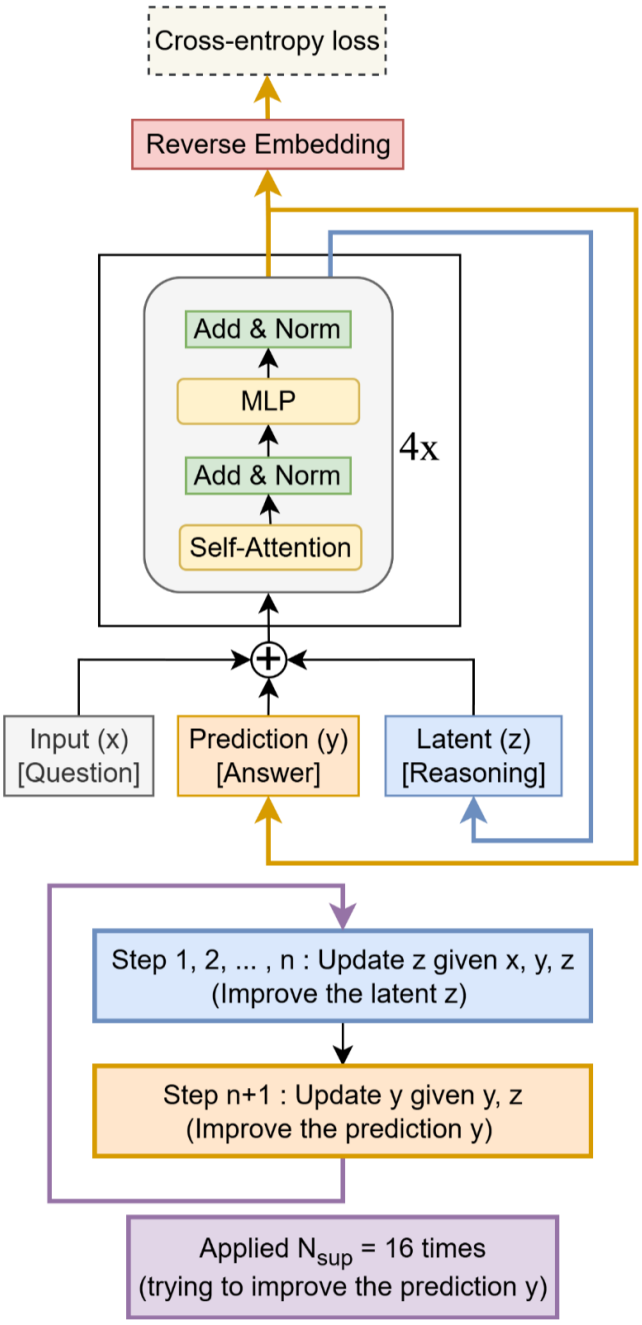

Figure 4: Tiny Recursion Model (TRM) recursively improves its predicted answer y with a tiny network. It starts with the embedded input question x and initial embedded answer y, and latent z. For up to $N_sup = 16$ improvements steps, it tries to improve its answer y. It does so by i) recursively updating n times its latent z given the question x, current answer y, and current latent z (recursive reasoning), and then ii) updating its answer y given the current answer y and current latent z. This recursive process allows the model to progressively improve its answer (potentially addressing any errors from its previous answer) in an extremely parameter-efficient manner while minimizing overfitting [JM25].

solution (z and y respectively) of the previous pass as the initial latent space and solution of the next pass. The goal is to allow the model to iteratively improve it's latent space and prediction

until it converges to the correct solution. The maximum number of step T used is $N_{sup} = 16$ in the TRM paper [JM25].

Deep supervision is expensive so to make the training faster an Adaptive Computational Time (ACT) is introduced. It allows the model to stop the recursion before $N_{sup} = 16$ is reached and hence allow the model to have a better coverage of the dataset given a fixed number of training iterations. The model learns a probability of halting using a Binary-Cross-Entropy loss of having reached the correct solution.

Through testing, the author of TRM paper found that the optimal number of recursions is $T = 3$ and $n = 6$ for a specific task (Sudoku-Extreme). As much as we can virtually increase T to infinity, this will massively slow down the model. In the case of increasing n, the model will also slow down but more importantly, the usage of memory will increase drastically leading to Out Of Memory (OOM) errors. However, increasing both T and n will lead to better results in complex problems.

To avoid over fitting and improve stability, an Exponential Moving Average (EMA) of the weight is introduced in the TRM. This is a common technique in GANs and diffusion models to improve stability [SE20] [BDS19].

The TRM may or may not use self attention depending on the task. Self-Attention thrives when using long sequences that depends on a small number of dimensions: $L \gg D$ since it only requires a matrix of $[D, 3D]$. But when the context length is short and fixed, self-attention can be replaced by a simple MLP block that will be faster and cheaper. A common way to do that is to use a MLP-Mixer [THK+21].

2.2 The TRM is ill-suited for PINNs

The TRM architecture as it is presented in its paper might not be suited for PINNs for multiple reasons. The TRM iteratively refine its result in order to converge to the correct token. In fact, the output of the TRM is approximated to the nearest token in the embedding. This is like solving a discrete problem as opposed to PINN that must approximate non-linear functions.

The input of the TRM itself is also a problem as it is expecting a sequence of tokens but in a PINN for solving 2D SWE, the input is only (t, x, y). This input makes the use of self-attention not recommended and force the use of simple MLP blocks.

The adaptive computational time is simply not applicable as it relies on a Binary-Cross Entropy loss of having reached the correct solution. The Binary-Cross Entropy loss is used in classification problem not estimation one. We could use another loss function to replace it but it wouldn't solve the other core problems so this wasn't tested.

In the TRM, multiple normalizations (RMSNorm) are applied to stabilize the training but this is detrimental for PINNs as it disrupts the direct relationship between the inputs and the output. This prevent the physic-informed loss from converging which remove the physical consistency required for reliable PDE solutions [JHYX25].

The biggest problem of all is the structure of the network of a TRM. It is a simple 2 layers network. Using MLP blocks this structure doesn't allow the output of the TRM to be non-linear as the PDE. In fact the output of the TRM is almost linear as shown in Figure (5).

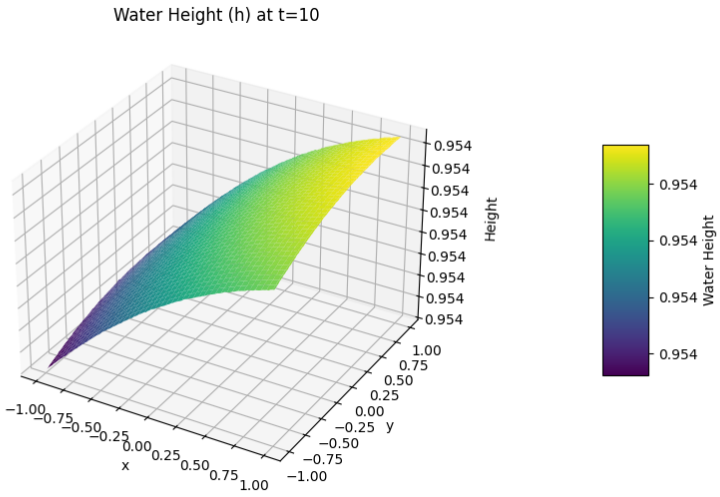

Figure 5: Example of the output of a TRM using MLP blocks trained using 50 Adams epochs and 5 LBFGS epochs

2.3 Our Model

We propose a model that is inspired from the TRM but that doesn't have the same issues. The goal of our approach is to leverage deep supervision to simulate a deep residual network to get similar or better performances in a PINN while having less parameters. First we are using Fourier Features to project the inputs onto a bigger space. The projection is given by the following equation:

$$\gamma(\mathbf{x}) = \begin{bmatrix} \cos(2\pi \mathbf{B}\mathbf{x}) \\ \sin(2\pi \mathbf{B}\mathbf{x}) \end{bmatrix}, \tag{2.3}$$

where $\mathbf{x}$ is the initial input vector and $\mathbf{B} \in \mathbb{R}^{m \times d}$ is sampled from a normal distribution $\mathcal{N}(0, \sigma^2)$ with $\sigma > 0$. The use of Fourier Features enhance the network ability to learn high-frequency functions [DCML25].

Then $\gamma(\mathbf{x})$ is treated as the real input in the rest of the model. We are using a small network f that takes into input $\gamma(\mathbf{x})$ and the latent space y. It outputs the improved latent space and this process is repeated n times to (hopefully) converge to a better answer. The final latent space doesn't have the correct dimension as it is in the same space as the projection. So we have a single dense layer that maps it to the desired output.

This approach allows us to simulate a deep residual network but it requires a lot of memory as the gradient must be computed accross all the recursions. We used deep supervision to improve the latent space T times without any gradient before doing one forward pass with the gradients. Another problem of this approach is the gradient explosion and vanishing. To alliviate this problem, we use the mask proposed by [JHYX25]. For each layer, we compute a mask using the following function

$$F(\mathbf{x}) = 1 - \exp(-(\alpha \cdot \mathbf{x})^2), \tag{2.4}$$

where α is a learnable vector that controls the sharpness of the mask function. It is initialized with positive values. The output of a layer is then given by

$$y = F(\mathbf{x}) \times \theta(\mathbf{x}), \tag{2.5}$$

where θ is the layer and it's activation function that we apply the mask on. Figure (6) shows how the masks function behave with respect to α.

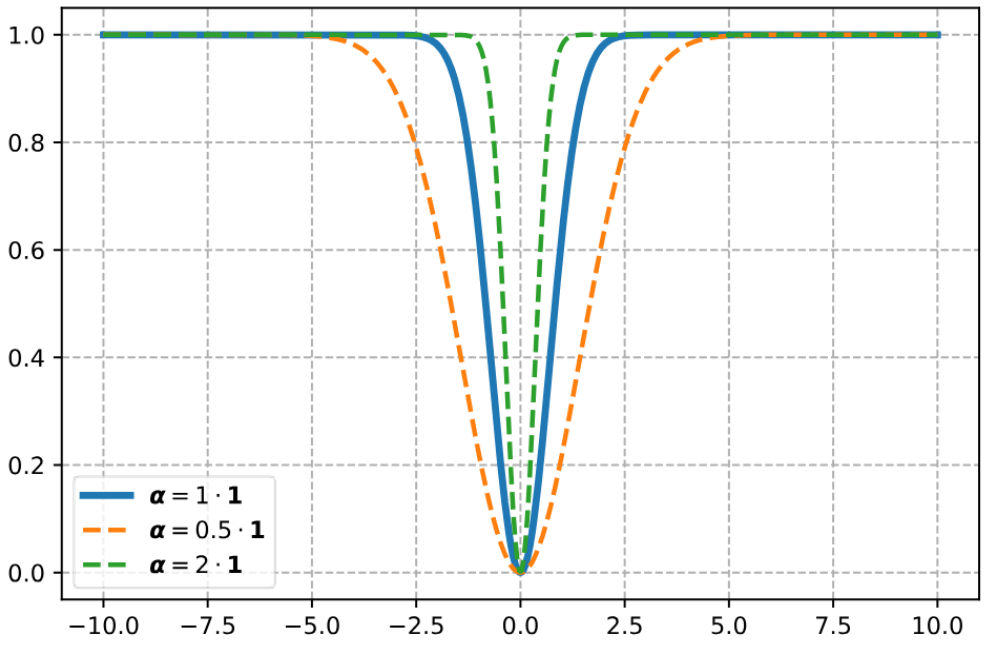

Figure 6: A mask function with respect to α

3 Results

For all the training and testing we used data generated using the github repository provided by PDEBench [TPL+24] with some small modifications to have a the x, y coordinates in $[-1, 1]$ and a greater amount of water at height 2 at the beginning of the simulation. The figure (7) shows the starting state of the simulation.

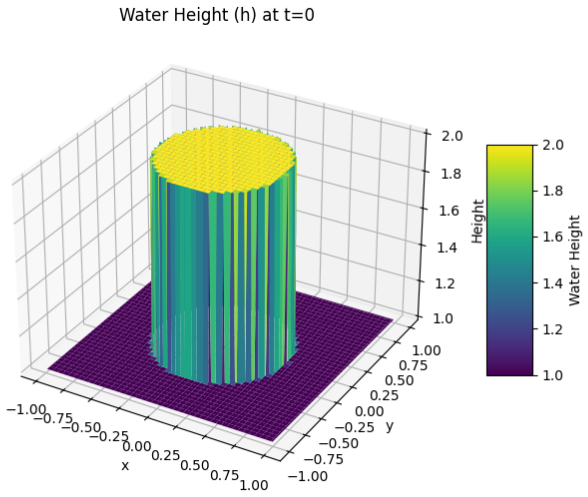

Figure 7: Initial state of the simulation

We trained each model during 10000 epochs using the Adam optimizer. For each epochs 30000 points are randomly chosen from the simulation and shuffled. The simulation contains a total of 1.6 million points. We used an L_1 loss for the data loss, we multiplied the physical loss by 0.01 and we used the L_1 distance for regularization making the model more sparse and less prone to overfitting [LSWH24]. The entire loss function is given by

$$loss = l_{data} + 0.01 l_{physical} + 10^{-5} l_{regularization} \tag{3.1}$$

$$l_{data} = L_1(y_{pred}, y_{true}) \tag{3.2}$$

$$l_{physical} = MSE(\text{gradients for the mass, momentum in x and y}) \tag{3.3}$$

$$l_{regularization} = L_1(w) \tag{3.4}$$

where w represents all the weights of the model. Additionally we trained the simples models using L-BFGS optimizer for 1000 epochs after the training with Adam's optimizer for the fine tuning. With did not do that for our model because we lack time and since L-BFGS requires the second derivative it is really slow on our model due to the recursions. Our models consists of 3 layers of 64 neurones while the other models are made of 7 layers of 64 neurones.
We can see in figure (8) that during the training, our model isn't the one that converge the fastest but it does converge well and the loss is stable. Figure (9) shows the loss of our model when we increase the number of recursion before doing a final prediction. We can clearly see that when $T = 5$, the model converges faster and has a lower loss at the end. This indicates that deep supervision can be useful for estimating continuous values.

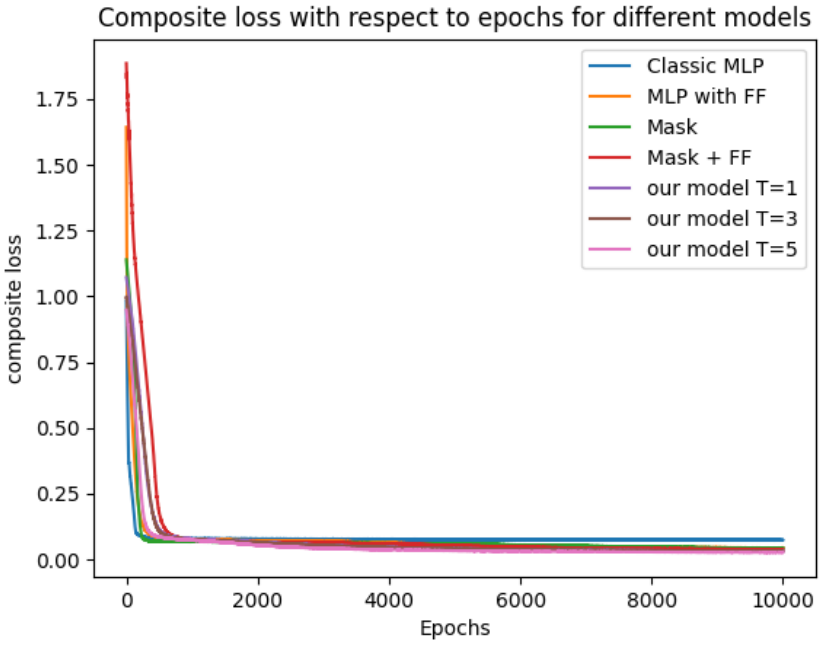

Figure 8: Loss of all the models with respect to the epochs. Our models only use 3 layers of 64 neurons while the other models are using 7 layers of 64 neurones. For our model, T is the number of recursions before doing the final prediction. We used $n = 3$ which means that we are improving the latent space 3 times before making a prediction.

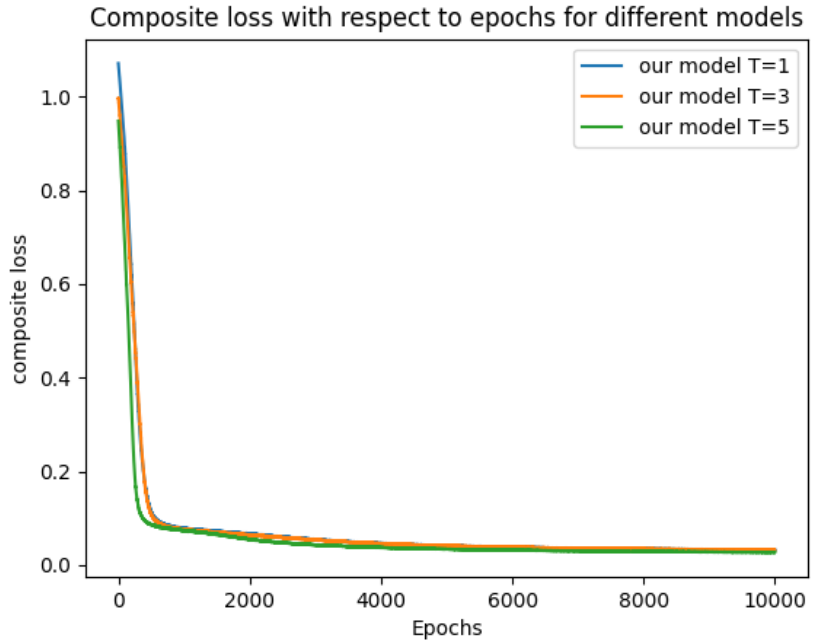

Figure 9: Loss of all the models with respect to the epochs. Our models only use 3 layers of 64 neurons while the other models are using 7 layers of 64 neurones. For our model, T is the number of recursions before doing the final prediction. We used $n = 3$ which means that we are improving the latent space 3 times before making a prediction.

4 Conclusion

While the TRM architecture is not working for PINN, the deep supervision part can be implemented and give good results. Our model is able to have similar results as other classic PINN implementation while using less parameters and it did not suffer from gradient explosion/vanishing. It is not clear that increasing the value of T will improve the results of our model and same goes for increasing n. We were not able to train our model with $n > 3$ as the memory requirements where too big for consummer GPUs. We did not try to include ACT in our model as it is doing approximation. But it might be possible to include using a different loss than Binary-Cross Entropy or using a separate small network that give the probability of the output being correct.

5 References

[BDS19] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.

[DCML25] Yi Ding, Su Chen, Hiroe Miyake, and Xiaojun Li. Physics-informed neural networks with fourier features for seismic wavefield simulation in time-domain nonsmooth complex media. *IEEE Transactions on Geoscience and Remote Sensing*, 63:1–13, 2025.

[HG16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units. 2016.

[JHYX25] Feilong Jiang, Xiaonan Hou, Jianqiao Ye, and Min Xia. Mask-pinns: Mitigating internal covariate shift in physics-informed neural networks, 2025.

[JM25] Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks, 2025.

[KL25] et. al. Kuang Luo. Physics-informed neural networks for pde problems: a comprehensive review. *Artificial Intelligence Review*, 2025.

[LSWH24] Yanling Li, Qianxing Sun, Junfang Wei, and Chunyan Huang. An improved pinn algorithm for shallow water equations driven by deep learning. *Symmetry*, 16(10), 2024.

[SE20] Yang Song and Stefano Ermon. Improved techniques for training score-based generative models, 2020.

[THK+21] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: an all-mlp architecture for vision. 2021.

[TPL+24] Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Dan MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. Pdebench: An extensive benchmark for scientific machine learning, 2024.