PINNs-Torch: Enhancing Speed and Usability of Physics-Informed Neural Networks with PyTorch

Reza Akbarian Bafghi
University of Colorado, Boulder
reza.akbarianbafghi@colorado.edu

Maziar Raissi
University of California, Riverside
maziar.raissi@ucr.edu

Abstract

Physics-informed neural networks (PINNs) stand out for their ability in supervised learning tasks that align with physical laws, especially nonlinear partial differential equations (PDEs). In this paper, we introduce "PINNs-Torch", a Python package that accelerates PINNs implementation using the PyTorch framework and streamlines user interaction by abstracting PDE issues. While we utilize PyTorch's dynamic computational graph for its flexibility, we mitigate its computational overhead in PINNs by compiling it to static computational graphs. In our assessment across 8 diverse examples, covering continuous, discrete, forward, and inverse configurations, naive PyTorch is slower than TensorFlow; however, when integrated with CUDA Graph and JIT compilers, training speeds can increase by up to 9 times relative to TensorFlow implementations. Additionally, through a real-world example, we highlight situations where our package might not deliver speed improvements. For community collaboration and future developments, our package code is accessible at: https://github.com/rezaakb/pinns-torch.

1 Introduction

Physics-informed neural networks (PINNs) have recently emerged as a powerful approach to supervised learning tasks, ensuring that solutions adhere to the laws of physics, particularly as represented by nonlinear partial differential equations (PDEs) [19]. Their effectiveness spans a diverse range of applications [21, 2, 6, 10, 23]. In this paper, we introduce "PINNs-Torch", a new Python package designed to accelerate PINNs using the PyTorch framework [14] and also simplify user interaction by abstracting PDE problems.

While PyTorch [14] is a popular framework for deep learning, its dynamic computational graph technique, which constructs and evaluates the graph in real-time, can introduce computational overheads, especially in PINNs. This is because PINNs often require multiple gradient computations of network outputs with respect to inputs to define PDEs [13]. This overhead can hinder the efficiency and speed of PINN implementations.

Extending prior works [9, 3, 8, 1], we present the "PINNs-Torch" package. It optimizes PINNs training and inference speed by leveraging static computational graphs with CUDA Graph [22] and JIT compilers, particularly for smaller batch sizes. Similar to [1], we leverage Hydra [27] to streamline problem definitions, and additionally, we use Lightning [5] to enhance scalability. This not only simplifies user interactions but also abstracts PDE problems.

Our findings indicate that combining CUDA Graph with JIT provides an optimal balance of speed and accuracy, eliminating repetitive graph creation for gradient calculations in PyTorch's dynamic framework. However, through our experiments, we highlight instances where the use of CUDA Graph falls short. We hope our package will drive advancements in PINNs acceleration.

DLDE-III Workshop in the 37th Conference on Neural Information Processing Systems (NeurIPS 2023).

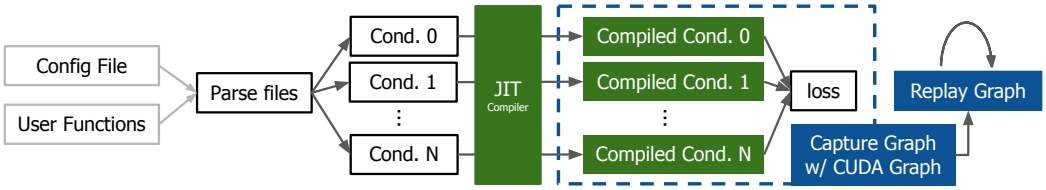

Figure 1: Simplified PINNs-Torch framework: Users input a config file and define data reading and PDE functions. The system then parses files, JIT compiles conditions (e.g., initial conditions, collection points), captures the computational graph, and trains the model through graph replay.

2 PINNs-Torch Package

In this section, we offer a concise overview of the problem setup and describe the implementation of our package.

2.1 Problem Setup

We follow the problem formulation introduced in [19]. The paper deals with a scenario where we examine parametric and nonlinear PDEs characterized by the following general structure:

$$u_t + \mathcal{N}[u; \lambda], \quad x \in \Omega, \quad t \in [0, T]$$

Here, $u(t, x)$ represents the underlying solution that is not directly observable, $\mathcal{N}[.; \lambda]$ stands for a nonlinear operator controlled by the parameter λ, and Ω is a subset of $\mathbb{R}^D$. Two key problems are discussed: the first is concerned with data-driven solution (forward problems) [18, 17], aiming to elucidate the concealed state $u(t, x)$ of the system given fixed model parameters λ. The second involves data-driven discovery (inverse problem) [18, 16, 24], aiming to ascertain the parameter values λ that provide the best explanation for the observed data.

Two algorithm types have been developed based on the data: continuous time models and discrete time models. The former efficiently handles data with new spatio-temporal function approximators, while the latter uses Runge-Kutta [7] methods with flexible staging. For more details, see [19].

2.2 Implementation

PINNs-Torch Workflow. Our package simplifies the solving of forward and inverse problems in discrete and continuous modes related to nonlinear partial differential equations. Following the architecture used by [1], it begins by parsing configuration files to extract parameters like spatial/temporal domains, sample count, boundary conditions, and neural network details. The user-defined PDE function and config files are then read. However, instead of a custom trainer, we use Lightning. Based on these, conditions are compiled and a computational graph is captured using CUDA Graph. Training is executed by replaying this graph. Figure 1 provides a workflow overview.

CUDA Graphs. CUDA Graphs [22] were introduced to the PyTorch API, enabling the representation of tasks through a directed acyclic graph (DAG), as opposed to individual kernel operations. A CUDA Graph comprises nodes that symbolize actions such as memory operations and kernel launches. These nodes are linked by edges that indicate dependencies for the order of execution. This approach facilitates an execution sequence characterized by creating the graph once and utilizing it multiple times, achieved by decoupling graph creation from execution. This separation empowers the reuse of graphs for multiple launches [28, 15].

JIT Compiler. In our package, we leverage TorchScript [4], an integral feature of PyTorch, to facilitate smooth transitions between eager execution and graph-based modes, thereby enhancing the performance of models. With dynamic batches (constant input shape but varying data), we employ the TorchScript API for scripting PDE functions. For static batches (unchanging data and input shape), we trace each condition. We use compiled functions for capturing the graph.

AMP. AMP is a popular PyTorch API that dynamically adjusts the numerical precision in deep learning models during training or inference [11]. We utilize 16-bit precision to evaluate its impact on training speed.

Table 1: Comparison of average speed-ups across eight examples discussed in Section 3.1 using various acceleration methods relative to TensorFlow and PyTorch. The table indicates that naive PyTorch underperforms compared to TensorFlow, with the most effective speed-up achieved through the combination of CUDA Graph and JIT Compiler. CG: CUDA Graph.

	PyTorch	JIT	AMP	CG	CG+AMP	CG+JIT
Avg. speed-up w.r.t. TF1	0.74	0.77	0.74	5.29	4.83	**5.43**
Avg. speed-up w.r.t. PyTorch	1	0.99	0.93	7.63	6.68	**7.82**

3 Experiments

In this section, we delve into the experiments conducted to evaluate the performance of CUDA Graph under various conditions, especially its interplay with batch sizes. We also compare its efficacy against other acceleration techniques, such as JIT and AMP, to determine the optimal approach for PINNs and their applications. In all experiments, we only use the Adam optimizer.

Hardware Setup. All experiments were conducted on a single NVIDIA Quadro RTX 8000 GPU to ensure consistency and reproducibility.

Speed-up Metric. We determine the median time taken for a single iteration in each scenario and contrast this with the time required in the original TensorFlow V1 (TF1) implementations[1]. The speed-up is computed by dividing the time from the TF1 implementation by the time from each scenario. However, in Table 1, the speed-up is assessed relative to the PyTorch as well.

Mean Relative Error Metric. We compute the average relative errors for each example. It's important to note that the nature of errors can vary by problem; further discussions on this are available in the Supplementary Materials Section D.

3.1 Evaluation of Various Acceleration Techniques

We evaluate the effectiveness of various acceleration techniques using distinct examples, including the Continuous Forward Schrodinger Equation, Discrete Forward Allen–Cahn (AC) Equation, Continuous Inverse Navier-Stokes (NS) Equation, and Discrete Inverse Korteweg-de Vries (KdV) Equation. Notably, Burgers' Equation is examined in all combinations of continuous, discrete, forward, and inverse configurations. In all the provided examples, we have static batches and employ tracing for JIT compilation. For more information, please refer to Supplementary Materials Section E and the original paper [19]. Our benchmarks span different scenarios, contrasting combinations of CUDA Graph, JIT compiler, and AMP against a baseline without any acceleration.

Figure 2 demonstrates that the combination of CUDA Graph and JIT Compiler achieves a notable speed-up, averaging 5.43, without sacrificing accuracy. The highest speed-up observed is 9.07, achieved in the KdV example. Although the pairing of AMP and CUDA Graph enhances speed-up in some examples, it introduces greater error. Our findings also show that TensorFlow's basic implementation outperforms PyTorch's (with an average speed-up of 0.74 for naive PyTorch), emphasizing the efficacy of static graphs. In Table 1, we present the average speed-up of our implemented examples against both TensorFlow and naive PyTorch. The results indicate that, aside from the CUDA Graph, other acceleration methods alone might not offer significant improvements in training speed.

3.2 Assessing the Impact of Batch Size and Number Trainable Parameters

In this subsection, we examine how variations in batch sizes and the number of trainable parameters impact the efficiency of a model utilizing CUDA Graph. We focus on simulating three-dimensional physiological blood flow within a realistic intracranial aneurysm (ICA) model, based on the 3D Navier-Stokes equation. The dataset contains 29 million data points across spatial and temporal domains for five solutions. In this example, given dynamic batches and random sampling in each iteration, we only script the PDE function for JIT compilation. For additional details about the example, readers are directed to Supplementary Materials Section E and the original papers [20, 21].

[1]For examples in section 3.1, we refer to implementations from github.com/maziarraissi/PINNs, and for the example in section 3.2, we use the implementation from github.com/maziarraissi/HFM.

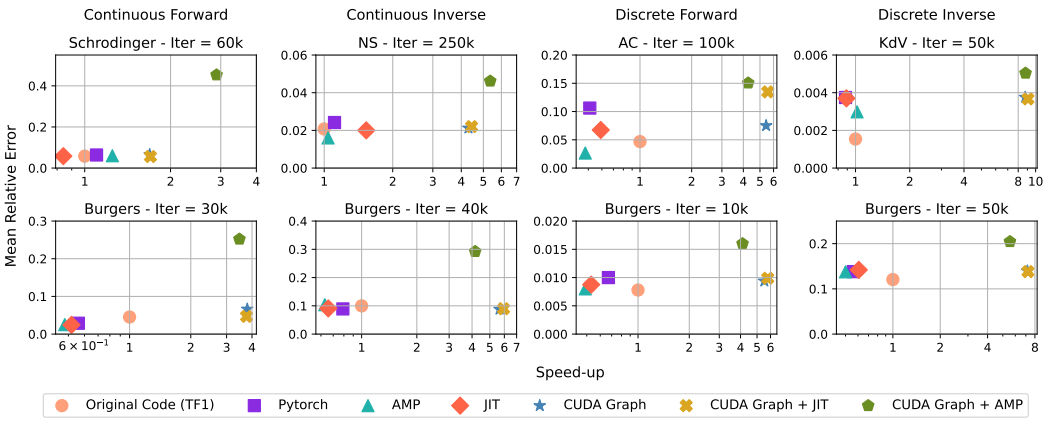

Figure 2: Each subplot corresponds to a distinct problem, with its iteration count displayed at the top. The logarithmic x-axis denotes the speed-up factor w.r.t TF1, and the y-axis illustrates the mean relative error. The plots reveal that using CUDA Graph with JIT Compiler can enhance speed without elevating the error. In contrast, combining CUDA Graph with AMP results in a higher mean error.

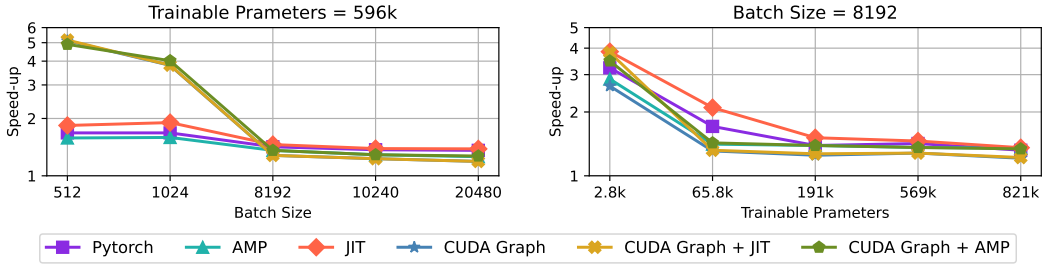

Figure 3: The left plot illustrates how the efficiency gains from using CUDA Graph diminish as the batch size increases. The right plot shows that varying the number of trainable parameters—achieved by changing the number of layers in the neural network—does not significantly affect gain performance across different configurations. The y-axis uses a log scale.

We again evaluate speed-up metrics under different configurations. First, keeping all attributes constant except batch size, the efficiency of CUDA Graph decreases as batch size grows, as seen in Figure 3's left plot. Using CUDA Graph reduces the CPU overheads more effectively when processing smaller batches. Next, with a fixed batch size of 8192 and varying neural network layers, performance is consistent across configurations (Figure 3 right plot). Changes in the total number of trainable parameters affect all methods similarly, but only CUDA Graph's performance notably varies with batch size changes. Thus, based on our results, for large datasets and dynamic batches, the JIT compiler alone is optimal. All configurations tested in PyTorch outperformed the original TensorFlow implementation. This could be attributed to our minor changes in the data loaders.

4 Conclusions and Limitations

In our package, we sought to boost execution speed using PyTorch APIs. Acceleration varies based on the problem and batch size, with potential speed enhancements up to 9x for static batches. However, current compilation APIs don't consistently enhance performance, especially for larger batches, highlighting a need for more efficient PyTorch compilers. Though PyTorch 2.0 [26] introduced a new compilation function, it doesn't support higher-order gradients, making it unsuitable for our use. Additionally, PyTorch's new experimental XLA compiler (used in TensorFlow [25]) wasn't incorporated due to GPU compatibility issues. We believe our package will be valuable for research purposes, and we plan to incorporate these compilers in future iterations of our package.

References

[1] Reza Akbarian Bafghi and Maziar Raissi. Pinns-tf2: Fast and user-friendly physics-informed neural networks in tensorflow v2. 2023.

[2] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: a review. *Acta Mechanica Sinica*, 37:1727 – 1738, 2021.

[3] Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. Neurodiffeq: A python package for solving differential equations with neural networks. *J. Open Source Softw.*, 5:1931, 2020.

[4] Zachary DeVito. Torchscript: Optimized execution of pytorch programs. *Retrieved January*, 2022.

[5] William Falcon and The PyTorch Lightning team. PyTorch Lightning, Mar. 2019.

[6] Ehsan Haghighat, Maziar Raissi, Adrian Moure, Héctor Gómez, and Ruben Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 379:113741, 2021.

[7] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[8] Alexander Koryagin, Roman Khudorozhkov, and Sergey Tsimfer. Pydens: a python framework for solving differential equations with neural networks. *ArXiv*, abs/1909.11544, 2019.

[9] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *ArXiv*, abs/1907.04502, 2019.

[10] Zhiping Mao, Ameya Dilip Jagtap, and George Em Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 360:112789, 2020.

[11] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Frederick Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *ArXiv*, abs/1710.03740, 2017.

[12] Vinh Nguyen, Michael Carilli, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, Edward Yang, and Sukru Burc Eryilmaz. Accelerating pytorch with cuda graphs. `https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/`.

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zach DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Neural Information Processing Systems*, 2019.

[15] Bo Qiao, M. Akif Özkan, Jürgen Teich, and Frank Hannig. The best of both worlds: Combining cuda graph with an image processing dsl. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[16] Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *ArXiv*, abs/1708.00588, 2017.

[17] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Inferring solutions of differential equations using noisy multi-fidelity data. *J. Comput. Phys.*, 335:736–746, 2016.

[18] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Numerical gaussian processes for time-dependent and nonlinear partial differential equations. *SIAM J. Sci. Comput.*, 40, 2017.

[19] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.

[20] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. *ArXiv*, abs/1808.04327, 2018.

[21] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 367:1026 – 1030, 2020.

[22] Pramod Ramarao. Cuda 10 features revealed: Turing, cuda graphs, and more. `https://developer.nvidia.com/blog/cuda-10-features-revealed/`, Aug 2022.

[23] Majid Rasht-Behesht, Christian Huber, Khemraj Shukla, and George Em Karniadakis. Physics-informed neural networks (pinns) for wave propagation and full waveform inversions. *Journal of Geophysical Research: Solid Earth*, 127, 2021.

[24] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3, 2016.

[25] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[26] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023.

[27] Omry Yadan. Hydra - a framework for elegantly configuring complex applications. Github, 2019.

[28] Chen Yu, Sara Royuela, and Eduardo Quiñones. Openmp to cuda graphs: a compiler-based transformation to enhance the programmability of nvidia devices. *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, 2020.

```yaml
...
train_datasets:
  - mesh_sampler:
      num_sample: ${n_f}
      collection_points:
        - f
  - initial_condition:
      num_sample: ${n_0}
      solution:
        - u
  - dirichlet_boundary_condition:
      num_sample: ${n_b}
      solution:
        - u
...
```

config.yaml

```python
def read_data_fn(root_path):
    data = load_data(root_path, "burgers_shock.mat")
    exact_u = np.real(data["usol"])
    return {"u": exact_u}

def pde_fn(outputs, x, t, extra_variables=None):
    u_x, u_t = gradient(outputs["u"], [x, t])
    u_xx = gradient(u_x, x)
    outputs["f"] = u_t + outputs["u"] * u_x - \
                   (0.01 / np.pi) * u_xx
    return outputs

train(cfg, read_data_fn=read_data_fn, pde_fn=pde_fn)
```

train.py

Figure 4: A simplified config file and user-defined functions for the continuous forward Burgers' equation are provided as an example. User should define a config file and function that enable the package to read the data and calculate PDE.

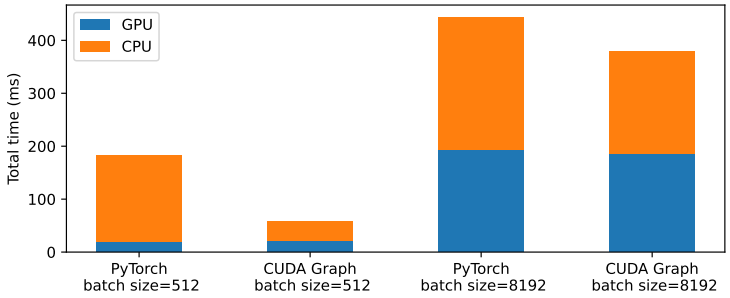

Figure 5: Total elapsed time per iteration in the example from Section 3.2. The plot illustrates that with a smaller batch size (512), CPU processing time significantly exceeds GPU time, suggesting CPU overhead as a primary bottleneck. CUDA Graph effectively reduces this overhead, enhancing training efficiency. Conversely, with larger batch sizes, CPU and GPU times are comparable, thus diminishing the impact of CPU overhead reduction on overall duration.

A Appendix

This document supplements the main paper as follows:

1. More details about the PINNs-Torch Workflow (supplements **Section 2.2**).
2. Insights into relative errors and mean squared errors used for training and testing for each problem (supplements **Section 3**).
3. More details explaining the limited performance of CUDA Graph with larger batch sizes (supplements **Section 3.2**).
4. Extended details, including problem setup, relative errors, and speed-ups, on the examples tested with our package (supplements **Section 3**).

B PINNs-Torch Workflow

Users need to create a config file that Hydra interprets, subsequently initializing the appropriate classes. They also have to define functions for data retrieval and PDE specifications. Figure 4 illustrates a simplified example of such a file and the user-defined functions. The package uses these definitions to solve the PDE.

C The Effect of CUDA Graph

Figure 5 shows the results of the elapsed time per iteration for different batch sizes in naive PyTorch and PyTorch accelerated with CUDA Graph. With large batches, the CPU and GPU contribute similarly to total execution time, making CPU optimization less critical. In contrast, smaller batches see higher CPU overhead, outpacing GPU runtime and causing GPU idleness. This issue is most acute at very small batch sizes, where CPU overhead significantly affects overall performance. CUDA Graph is designed to minimize this CPU overhead [12].

D Errors

Mean Squared Error and Sum Squared Error. In this paper, we introduce a unified notation, Err, to represent error measures which could be either the mean squared error (MSE) or the sum squared error (SSE). The specific interpretations of Err are as follows: Err_0 signifies the error in the initial condition, Err_b denotes the error related to the boundary condition, Err_c highlights the error in collection points, Err_s captures the error in the sampled exact and predicted solutions, and Err^i indicates the error at the time step i.

Relative Errors. To compute the errors between the predicted and exact solutions, we employ the relative L_2-norm:

$$\frac{\|u_{\text{pred}} - u_{\text{target}}\|_2}{\|u_{\text{target}}\|_2}$$

For additional variables trained in the inverse problem, we utilize:

$$\frac{|\lambda_{\text{pred}} - \lambda_{\text{target}}|}{|\lambda_{\text{target}}|}$$

E Examples

In this section, we provide a brief overview of the examples discussed in our main paper. We strongly encourage readers to consult the original paper [19] for comprehensive details on the first 8 examples and the original paper [20, 21] for the 3D Navier-Stokes equation that was implemented in section 3.2.

Continuous Forward Schrodinger Equation. Following the setting introduced in [19], the nonlinear Schrodinger equation is described as:

$$\begin{aligned} ih_t + 0.5h_{xx} + |h|^2 h &= 0, \\ h(0,x) &= 2\text{sech}(x), \\ h(t,-5) &= h(t,5), \\ h_x(t,-5) &= h_x(t,5), \end{aligned}$$

where $x \in [-5, 5]$, $t \in [0, \pi/2]$, and the function $h(t, x)$ represents the complex solution. By setting a complex-valued neural network foundation on $h(t, x)$ and considering u as the real segment of h and v as its imaginary segment, our foundation on $h(t, x)$ can be expressed as $[u(t, x), v(t, x)]$. Table 2 summarizes the problem setup for this equation.

The prediction error from each code is assessed against the test data for this issue, using the relative L_2-norm as the metric. Table 3 displays the relative L_2-norm errors for $h(t, x)$, $v(t, x)$, $u(t, x)$, along with the mean error referenced in the main paper.

Continuous Inverse Navier-Stokes Equation. The 2D nonlinear Navier-Stokes equation is articulated as:

$$\begin{aligned} u_t + \lambda_1(uu_x + vu_y) &= -p_x + \lambda_2(u_{xx} + u_{yy}), \\ v_t + \lambda_1(uv_x + vv_y) &= -p_y + \lambda_2(v_{xx} + v_{yy}), \end{aligned}$$

Table 2: The problem setup for continuous forward Schrodinger equation.

Continuous Forward Schrodinger Equation	
PDE equations	$f_u = u_t + 0.5v_{xx} + v(u^2 + v^2)$, $f_v = v_t + 0.5u_{xx} + u(u^2 + v^2)$
Initial condition	$u(0, x) = 2\text{sech}(x)$, $v(0, x) = 0$
Periodic boundary conditions	$u(t, -5) = u(t, 5)$, $v(t, -5) = v(t, 5)$, $u_x(t, -5) = u_x(t, 5)$, $v_x(t, -5) = v_x(t, 5)$
The output of net	$[u(t, x), v(t, x)]$
Layers of net	$[2] + 4 \times [100] + [2]$
Sample count from collection points	20000
Sample count from the initial condition	50
Sample count from boundary conditions	50
Loss function	$\text{MSE}_0 + \text{MSE}_b + \text{MSE}_c$

Table 3: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for continuous forward Schrodinger equation.

Method	Relative Errors			Mean Relative Error	Speed-up
	$h(t, x)$	$v(t, x)$	$u(t, x)$		
Original Code (TF1)	0.017	0.104	0.064	0.061	1
PyTorch	0.024	0.102	0.064	0.064	1.10
AMP	0.022	0.097	0.061	0.060	1.25
JIT	0.018	0.098	0.059	0.058	0.84
CUDAGraph	0.022	0.110	0.063	0.065	1.69
CUDAGraph + AMP	0.124	0.735	0.501	0.453	**2.90**
CUDAGraph + JIT	0.016	0.095	0.056	**0.056**	1.70

Table 4: The problem setup for the continuous inverse Navier-Stokes equation.

Continuous Inverse Navier-Stokes Equation	
PDE equations	$f = u_t + \lambda_1(uu_x + vu_y) + p_x - \lambda_2(u_{xx} + u_{yy})$, $g = v_t + \lambda_1(uv_x + vv_y) + p_y - \lambda_2(v_{xx} + v_{yy})$
Assumptions	$u = \psi_y$, $v = -\psi_x$
The output of net	$[\psi(t, x, y), p(t, x, y)]$
Layers of net	$[3] + 8 \times [20] + [2]$
Sample count from collection points	5000^*
Sample count from solutions	5000^*
Loss function	$\text{SSE}_s + \text{SSE}_c$

**Same points used for collocation and solutions.*

Table 5: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for continuous inverse Navier-Stokes equation.

Method	Relative Errors				Mean Relative Error	Speed-up
	$v(t,x)$	$u(t,x)$	λ_1	λ_2		
Original Code (TF1)	0.018	0.009	0.002	0.054	**0.021**	1
PyTorch	0.024	0.009	0.002	0.062	0.024	1.11
AMP	0.023	0.007	0.001	0.033	0.016	1.04
JIT	0.021	0.007	0.002	0.049	0.020	1.53
CUDAGraph	0.025	0.008	0.001	0.051	**0.021**	4.30
CUDAGraph + AMP	0.052	0.018	0.009	0.106	0.046	**5.36**
CUDAGraph + JIT	0.021	0.007	0.002	0.059	0.022	4.44

Table 6: The problem setup for discrete forward Allen-Cahn equation.

Discrete Forward AC Equation	
PDE equations	$f^{n+c_j} = 5.0u^{n+c_j} - 5.0(u^{n+c_j})^3 + 0.0001u_{xx}^{n+c_j}$
Periodic boundary conditions	$u(t,-1) = u(t,1),$ $u_x(t,-1) = u_x(t,1)$
The output of net	$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)]$
Layers of net	$[1] + 4 \times [200] + [101]$
The number of stages (q)	100
Sample count from collection points at t_0	200*
Sample count from solutions at t_0	200*
$t_0 \to t_1$	$0.1 \to 0.9$
Loss function	$\text{SSE}_s^0 + \text{SSE}_c^0 + \text{SSE}_b^1$

**Same points used for collocation and solutions.*

Here, $u(t,x,y)$ represents the x-component of the velocity field, $v(t,x,y)$ signifies the y-component, and $p(t,x,y)$ indicates the pressure. The unknown parameters are denoted by $\lambda = (\lambda_1, \lambda_2)$. Depending on the specific problem, we also incorporate the following equations:

$$\begin{aligned} 0 &= u_x + v_y, \\ u &= \psi_y, \\ v &= -\psi_x, \end{aligned} \tag{1}$$

We then approximate $[\psi(t,x,y), p(t,x,y)]$ using a neural network with dual outputs. This assumption, combined with equations (1), gives rise to a physics-informed neural network $[f(t,x,y), g(t,x,y)]$. Table 4 summarizes the problem setup.

The prediction discrepancies from each code are evaluated against the test dataset. Table 5 lists the relative L_2-norm errors for $u(t,x)$ and $v(t,x)$, as well as the relative errors for λ_1 and λ_2, complemented by the average error as mentioned in the primary paper.

Discrete Forward Allen-Cahn Equation. The non-linear AC equation can be expressed as:

$$\begin{aligned} u_t - 0.0001u_{xx} + 5u^3 - 5u &= 0, \\ u(0,x) &= x^2\cos(\pi x), \\ u(t,-1) &= u(t,1), \\ u_x(t,-1) &= u_x(t,1), \end{aligned}$$

where $x \in [-1,1]$, $t \in [0,1]$. Due to the fact that this problem is discrete, we follow terminology from [19, 7] for defining Runge–Kutta methods with q stages. Thus, the output of the neural network

Table 7: The problem setup for discrete inverse Korteweg–de Vries equation.

Discrete Inverse KdV Equation	
PDE equations	$f^{n+c_j} = -\lambda_1 u^{n+c_j} u_x^{n+c_j} - \lambda_2 u_{xxx}^{n+c_j}$
The output of net	$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x)]$
Layers of net	$[1] + 3 \times [50] + [50]$
The number of stages (q)	50
Sample count from solutions at t_0	199*
Sample count from collection points at t_0	199*
Sample count from solutions at t_1	201*
Sample count from collection points at t_1	201*
$t_0 \rightarrow t_1$	$0.2 \rightarrow 0.8$
Loss function	$\text{SSE}_s^0 + \text{SSE}_c^0 + \text{SSE}_s^1 + \text{SSE}_c^1$

**Same points used for collocation and solutions.*

Table 8: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete forward Allen-Cahn equation.

Method	Relative Error $u(t,x)$	Mean Relative Error	Speed-up
Original Code (TF1)	0.047	0.047	1
PyTorch	0.106	0.106	0.51
AMP	0.027	**0.027**	0.48
JIT	0.067	0.067	0.59
CUDAGraph	0.075	0.075	5.43
CUDAGraph + AMP	0.150	0.150	4.27
CUDAGraph + JIT	0.135	0.135	**5.55**

for this problem is:

$$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)]$$

In this context, $u^{n+c_j} = u(t^n + c_j \Delta t, x)$ for $j = 1, \ldots, q$ represents the information at time-step t^n and u^{n+1} represent the prediction at time-step t^{n+1}. Table 6 summarizes the problem setup for this equation. In this specific instance, we selectively extract data from the precise solution at $t_0 = 0.1$, with the objective of forecasting the solution at $t_1 = 0.9$. This is achieved by employing a solitary time-step of magnitude $\Delta t = 0.8$. Table 8 displays L_2-norm errors for $u(x)$ at t_1.

Discrete Inverse Korteweg–de Vries Equation. The non-linear KdV equation can be expressed as:

$$u_t + \lambda_1 u u_x + \lambda_2 u_{xxx} = 0$$

We employ Runge–Kutta methods with q stages to learn the parameters $\lambda = (\lambda_1, \lambda_2)$ of the KdV equation. The network's output in this problem is represented as:

$$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x)]$$

where $u^{n+c_j} = u(t^n + c_j \Delta t, x)$ for $j = 1, \ldots, q$ indicate the data at time-step t^n. We sample two solution snapshots at times $t^n = 0.2$ and $t^{n+1} = 0.8$. The problem setup is summarized in Table7. Also, Table 9 presents the relative errors for λ_1 and λ_2.

Continuous Forward Burgers' Equation. The Burgers' equation is given by:

$$u_t + u u_x - (0.01/\pi) u_{xx} = 0,$$

Table 9: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete inverse Korteweg–de Vries equation.

Method	Relative Errors		Mean Relative Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.0005	**0.002**	1
PyTorch	0.001	0.006	0.004	0.88
AMP	0.001	0.005	0.003	1.02
JIT	0.001	0.006	0.004	0.89
CUDAGraph	0.001	0.006	0.004	8.75
CUDAGraph + AMP	0.003	0.008	0.005	8.79
CUDAGraph + JIT	0.001	0.006	0.004	**9.07**

Table 10: The problem setup for continuous forward Burgers' equation.

Continuous Forward Burgers' Equation	
PDE equations	$f = u_t + uu_x - (0.01/\pi)u_{xx}$
Initial conditions	$u(0, x) = -\sin(\pi x)$
Dirichlet boundary conditions	$u(t, -1) = u(t, 1) = 0$
The output of net	$[u(t, x)]$
Layers of net	$[2] + 8 \times [20] + [1]$
Sample count from collection points	10000
Sample count from the initial condition	50
Sample count from boundary conditions	50
Loss function	$\text{MSE}_0 + \text{MSE}_b + \text{MSE}_c$

where $x \in [-1, 1]$, $t \in [0, 1]$. The initial and boundary conditions are:

$$\begin{aligned} u(0, x) &= -\sin(\pi x), \\ u(t, -1) &= 0, \\ u(t, 1) &= 0. \end{aligned}$$

In this problem, our goal is to find the solution $u(t, x)$. The problem setup is summarized in Table 10, and Table 11 presents the relative error for $u(t, x)$.

Continuous Inverse Burgers' Equation. In this scenario, we are dealing with the equation:

$$u_t + \lambda_1 uu_x - \lambda_2 u_{xx} = 0.$$

Table 11: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for continuous forward Burgers' equation.

Method	Relative Error	Mean Relative Error	Speed-up
	$u(t, x)$		
Original Code (TF1)	0.045	0.045	1
PyTorch	0.029	0.029	0.56
AMP	0.025	0.025	0.52
JIT	0.024	**0.024**	0.59
CUDAGraph	0.066	0.066	**3.79**
CUDAGraph + AMP	0.252	0.252	3.47
CUDAGraph + JIT	0.047	0.047	3.76

Table 12: The problem setup for continuous inverse Burgers' equation.

Continuous Inverse Burgers' Equation	
PDE equations	$f = u_t + \lambda_1 u u_x - \lambda_2 u_{xx}$
The output of net	$[u(t, x)]$
Layers of net	$[2] + 8 \times [20] + [1]$
Sample count from collection points	2000^*
Sample count from solutions	2000^*
Loss function	$\text{MSE}_s + \text{MSE}_c$

**Same points used for collocation and solutions.*

Table 13: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for continuous inverse Burgers' equation.

Method	Relative Errors		Mean Relative Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.196	0.100	1
PyTorch	0.004	0.174	0.089	0.79
AMP	0.006	0.202	0.104	0.63
JIT	0.003	0.178	0.091	0.66
CUDAGraph	0.004	0.172	**0.088**	5.65
CUDAGraph + AMP	0.013	0.572	0.292	4.16
CUDAGraph + JIT	0.003	0.176	0.090	**5.94**

Our objective is twofold: to predict the complete solution denoted as $u(t, x)$, and to estimate the unknown parameters $\lambda = (\lambda_1, \lambda_2)$. You can find the problem setup details in Table 12, while Table 13 presents the relative errors for $u(t, x)$, λ_1, and λ_2.

Discrete Forward Burgers' Equation. In this problem, we gather data from time step $t_1 = 0.1$ and aim to predict solutions at time $t_2 = 0.9$ using Runge-Kutta methods with q stages. The equation is defined as:

$$f^{n+c_j} = u_t + u^{n+c_j} u_x^{n+c_j} - (0.01/\pi) u_{xx}^{n+c_j}$$

where u^n represents the information at time-step t^n. The specific configuration of the problem is available in Table 15, and you can also refer to Table 14 for the relative errors concerning $u(t, x)$.

Table 14: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete forward Burgers' equation.

Method	Relative Error	Mean Relative Error	Speed-up
	$u(t, x)$		
Original Code (TF1)	0.008	**0.008**	1
PyTorch	0.010	0.010	0.67
AMP	0.009	0.009	0.49
JIT	0.024	0.024	0.53
CUDAGraph	0.009	0.009	5.52
CUDAGraph + AMP	0.016	0.016	4.11
CUDAGraph + JIT	0.010	0.010	**5.78**

Table 15: The problem setup for discrete forward Burgers' equation.

Discrete Forward Burgers' Equation	
PDE equations	$f^{n+c_j} = u_t + u^{n+c_j} u_x^{n+c_j} - (0.01/\pi) u_{xx}^{n+c_j}$
Dirichlet boundary conditions	$u(t,-1) = u(t,1) = 0$
The output of net	$[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)]$
Layers of net	$[1] + 3 \times [50] + [501]$
The number of stages (q)	500
Sample count from collection points at t_0	250^*
Sample count from solutions at t_0	250^*
$t_0 \to t_1$	$0.1 \to 0.9$
Loss function	$\text{SSE}_s^0 + \text{SSE}_c^0 + \text{SSE}_b^1$

**Same points used for collocation and solutions.*

Table 16: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete inverse Burgers' equation.

Method	Error		Mean Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.239	**0.121**	1
PyTorch	0.003	0.273	0.138	0.56
AMP	0.003	0.273	0.138	0.50
JIT	0.006	0.278	0.142	0.61
CUDAGraph	0.003	0.276	0.140	7.16
CUDAGraph + AMP	0.005	0.404	0.205	5.55
CUDAGraph + JIT	0.003	0.273	0.138	**7.23**

Discrete Inverse Burgers' Equation. Much like the discrete forward Burgers' equation, we employ Runge-Kutta methods with q stages. However, in this instance, the equation is formulated as:

$$f^{n+c_j} = u_t + \lambda_1 u^{n+c_j} u_x^{n+c_j} - \lambda_2 u_{xx}^{n+c_j}$$

The aim is to predict the unknown variables of λ_1 and λ_2. Data are randomly sampled from $t = 0.1$ and $t = 0.9$. You can refer to Table 17 for the problem setup details and Table 16 for the associated relative errors.

Continuous Forward 3D Navier-Stokes Equation. In this example, the fluid's dynamics are represented by the non-dimensional Navier-Stokes and continuity equations:

$$\begin{aligned}
c_t + uc_x + vc_y + wc_z &= \text{Pec}^{-1}(c_{xx} + c_{yy} + c_{zz}), \\
u_t + uu_x + vu_y + wu_z &= -p_x + \text{Re}^{-1}(u_{xx} + u_{yy} + u_{zz}), \\
v_t + uv_x + vv_y + wv_z &= -p_y + \text{Re}^{-1}(v_{xx} + v_{yy} + v_{zz}), \\
w_t + uw_x + vw_y + ww_z &= -p_z + \text{Re}^{-1}(w_{xx} + w_{yy} + w_{zz}), \\
u_x + v_y + w_z &= 0.
\end{aligned}$$

This describes the development of the normalized concentration $c(t, x, y, z)$ in a passive scalar carried by an incompressible Newtonian fluid. The velocity components are represented by $\mathbf{u} = (u, v, w)$, and p denotes pressure. See Table 18 for the original problem setup. We adjusted the batch sizes for collection points and solutions, and modified the number of hidden layers to alter trainable parameters.

Table 17: The problem setup for discrete inverse Burgers' equation.

Discrete Inverse Burgers' Equation	
PDE equations	$f^{n+c_j} = u_t + \lambda_1 u^{n+c_j} u_x^{n+c_j} - \lambda_2 u_{xx}^{n+c_j}$
The output of net	$[u^{n+c_1}(x), \ldots, u^{n+c_{q-1}}(x), u^{n+c_q}(x)]$
Layers of net	$[1] + 4 \times [50] + [81]$
The number of stages (q)	81
Sample count from collection points at t_0	199*
Sample count from solutions at t_0	199*
Sample count from collection points at t_1	201*
Sample count from solutions at t_1	201*
$t_0 \rightarrow t_1$	$0.1 \rightarrow 0.9$
Loss function	$\text{SSE}_s^0 + \text{SSE}_c^0 + \text{SSE}_s^1 + \text{SSE}_c^1$

**Same points used for collocation and solutions at each time step.*

Table 18: The problem setup for continuous forward 3D Navier Stokes equation.

Continuous Forward 3D NS	
PDE equations	$e_1 = c_t + (uc_x + vc_y + wc_z) - (1.0/\text{Pec})(c_x x + c_y y + c_z z)$ $e_2 = u_t + (uu_x + vu_y + wu_z) + p_x - (1.0/\text{Rey})(u_x x + u_y y + u_z z)$ $e_3 = v_t + (uv_x + vv_y + wv_z) + p_y - (1.0/\text{Rey})(v_x x + v_y y + v_z z)$ $e_4 = w_t + (uw_x + vw_y + ww_z) + p_z - (1.0/\text{Rey})(w_x x + w_y y + w_z z)$ $e_5 = u_x + v_y + w_z$
The output of net	$[c(t,x,y,z), u(t,x,y,z), v(t,x,y,z), w(t,x,y,z), p(t,x,y,z)]$
Layers of net	$[4] + 10 \times [250] + [5]$
Batch size of collection points	10000
Batch size of solutions in $c(t,x,y,z)$	10000
Loss function	$\text{MSE}_s + \text{MSE}_c$