PINNs-TF2: Fast and User-Friendly Physics-Informed Neural Networks in TensorFlow V2

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) have gained prominence for their capability to tackle supervised learning tasks that conform to physical laws, notably nonlinear partial differential equations (PDEs). This paper presents "PINNs-TF2", a Python package built on the TensorFlow V2 framework. It not only accelerates PINNs implementation but also simplifies user interactions by abstracting complex PDE challenges. We underscore the pivotal role of compilers in PINNs, highlighting their ability to boost performance by up to 119x. Across eight diverse examples, our package, integrated with XLA compilers, demonstrated its flexibility and achieved an average speed-up of 18.12 times over TensorFlow V1. Moreover, a real-world case study is implemented to underscore the compilers' potential to handle many trainable parameters and large batch sizes. For community engagement and future enhancements, our package's source code is openly available at: `https://github.com/rezaakb/pinns-tf2`.

1 Introduction

Physics-informed neural networks (PINNs) are gaining traction as a potent tool for supervised learning, ensuring solutions align with physics laws, notably nonlinear partial differential equations (PDEs) [17]. Their versatility covers an array of applications [3, 9, 6, 20, 1]. This paper unveils "PINNs-TF2", a novel Python package to bolster PINNs' efficiency and to simplify the integration of machine learning and physical sciences by abstracting complex PDE problems.

We selected TensorFlow V2 (TF2) due to its prominence in the deep learning realm and its provision of static computational graphs. Given that PINNs frequently necessitate multiple gradient computations of network outputs in relation to inputs for PDE definition [12], the advantage of static graphs becomes evident. They reduce the overhead that can be significantly time-consuming in dynamic computational graphs, as seen in frameworks like PyTorch [13, 12].

Building upon previous works [10, 4, 8], the "PINNs-TF2" package utilizes static computational graphs via Accelerated Linear Algebra (XLA) and Just-In-Time (JIT) compilers [22], a technique also adopted by others, to significantly enhance the speed of Physics-Informed Neural Networks (PINNs). This approach yields considerable improvements in training times over TensorFlow V1 [17]. Through our package, we have showcased its versatility by implementing nine distinct examples, achieving an impressive peak speed-up of 119.96x compared to previous implementations in TensorFlow V1. Also, with the incorporation of the Hydra framework [24], "PINNs-TF2" refines user experience by distilling PDE problems into assorted samplers and boundary conditions.

Our results suggest that the exclusive use of the JIT compiler strikes an ideal equilibrium between speed-up and errors by obviating redundant graph constructions during gradient operations within TensorFlow's computational graph. Moreover, we elucidate the influence of batch sizes and the quantity of trainable parameters on the performance of implemented examples with our package. We hope our package serves as an indispensable tool for researchers delving into the PINNs domain.

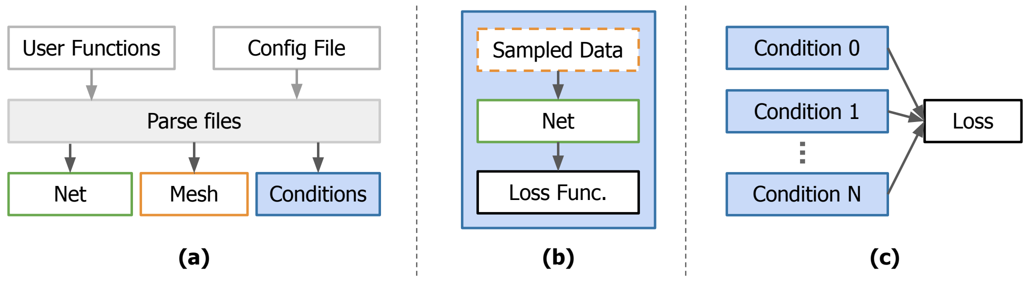


Figure 1: Overview of the simplified PINNs-TF2 framework: **(a)** Users provide a config file and define reading data and PDE functions. The package then processes them to initialize a neural network, formulate a mesh for the data, and establish conditions. **(b)** Elucidates the components of the conditions, including data sampled from the mesh, shared neural networks, and designated loss functions. **(c)** During training, cumulative loss from each condition directs backpropagation. This process is compiled with the XLA compiler.

2 PINNs-TF2 Package

In this section, we provide a brief summary of the problem setup and outline how our package works.

2.1 Problem Setup

We adopt the problem definition from [17]. The study focuses on parametric and nonlinear PDEs with the structure:

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

Where $u(t, x)$ is the unobservable solution, $\mathcal{N}[.; \lambda]$ is a nonlinear operator influenced by the parameter λ, and Ω is a subset of $\mathbb{R}^D$. Two core issues are addressed: Data-driven solution (forward problem) [16, 15] focuses on revealing the hidden state $u(t, x)$ for a given λ. Data-driven discovery (inverse problem) [16, 14, 21] seeks the optimal λ values based on observed data.

There are two algorithm approaches based on data types: continuous time and discrete time models. The former employs new spatio-temporal approximators, and the latter uses specific implicit Runge-Kutta methods. For further information, please refer to [17].

2.2 Implementation

PINNs-TF2 Workflow. Our package streamlines the process of addressing both forward and inverse challenges in discrete and continuous contexts linked to nonlinear partial differential equations. Initially, it processes configuration files using Hydra [24] to retrieve specifications such as spatial/temporal ranges, the number of samples, boundary conditions, and neural network attributes like layer count. Subsequently, it loads the user-specified PDE function and a function for loading data. With this data at hand, the relevant conditions, mesh based on data, and a neural network are initialized. Each condition can have its unique loss function (e.g. periodic boundary conditions calculate loss from the difference between predicted and actual values at the periodic boundary). All conditions share a common neural network. Both the training and evaluation phases are compiled using the XLA compiler. An illustrative overview of this workflow can be seen in Figure 1.

Compile with `tf.function`. When `tf.function` is set to `jit_compile=False`, TensorFlow translates Python functions into a static computational graph, optimized through pattern-matching rewrites. This approach, however, doesn't generate new code and relies on a limited set of predefined kernels, which can sometimes restrict its flexibility and optimization potential. In our work, this mode for compiling training and evaluation steps is referred to as "TF2".

On the other hand, when `tf.function` employs `jit_compile=True`, the XLA compiler [22], leveraging Just-In-Time (JIT) compilation, converts TensorFlow's computation graphs into highly optimized machine code right before execution. JIT offers several advantages: by fusing multiple operations, operating on the High Level Optimiser Internal Representation (HLO IR), and tailoring the

Table 1: Average speed-ups for the eight examples from Section 3.1, benchmarked against TensorFlow V1 using different acceleration methods. Among the methods, the JIT compiler alone stands out as the most effective accelerator.

	TF2	JIT	AMP	AMP+JIT
Avg. speed-up w.r.t. TF1	1.81	**18.12**	1.58	10.76

code for the nuances of the target hardware, it significantly enhances both speed and memory efficiency [7, 22]. JIT-optimized processes reduce overhead in PINNs' repetitive gradient computations, with XLA's static graph enhancing computational efficiency. In our architecture, we maintain consistent input shapes to avoid XLA compiler overhead from shape changes [23]. For the purposes of our experiments, this mode is termed "JIT".

Mixed Precision. We also employ TensorFlow's Mixed Precision using float16, blending FP16 and FP32 to accelerate training and conserve memory on GPUs [11]. We labeled this mode "AMP".

3 Experiments

In this section, we assess the performance of TF2, JIT, and AMP across 8 diverse examples, focusing on error maintenance and speed-up. We also demonstrate their efficacy with a large-scale dataset by implementing a real-world example.

Hardware Setup. All tests were carried out on a single NVIDIA Quadro RTX 8000 GPU to maintain uniformity and repeatability.

Speed-up Metric. We measured the median time for a single iteration in each case and compared it to the original TensorFlow V1 (TF1) implementations[1]. The speed-up is measured by dividing the duration from the TF1 version by the duration of each specific scenario in TensorFlow V2.

Mean Relative Error Metric. We calculate average relative errors for each example. Error nature may vary by problem; see Supplementary Materials Section C for details.

3.1 Evaluation of Various Acceleration Techniques

We measure the efficacy of acceleration techniques across various examples, including the Continuous Forward Schrodinger, Discrete Forward Allen–Cahn (AC), Continuous Inverse Navier-Stokes (NS), and Discrete Inverse Korteweg-de Vries (KdV) Equations. We also explore the Burgers' Equation in all modes. For in-depth insights about examples, see the Supplementary Materials Section D and [17]. Our benchmarks compare JIT compiler and AMP combinations against a non-accelerated baseline.

In Table 1, we present the average speed-ups of our examples compared to TensorFlow V1. By solely utilizing the JIT compiler, we achieved average speed-up of 18.12 without any compromise in accuracy. This advantage is visually represented in Figure 2, which plots both the speed-up and mean relative errors. The KdV example registered the highest speed-up, peaking at 31.75. Conversely, our performance did not benefit from using AMP, a limitation possibly due to hardware constraints or implementation overheads. Notably, TF2 outperformed TF1 with an average speed-up of 1.81. Moreover, our data points towards a decline in speed-up when AMP and JIT are combined, a phenomenon potentially resulting from the mixing precision and JIT overheads.

3.2 Assessing the Impact of Batch Size and Number Trainable Parameters

We explore the influence of batch sizes and the count of trainable parameters on the efficiency of models harnessing our accelerators. This section focuses on the computational intricacies of modeling a three-dimensional physiological blood flow inside a genuine intracranial aneurysm (ICA) using the 3D Navier-Stokes equation. Given the dataset's vastness, encompassing 29 million data points spanning spatial, and temporal domains, and five solutions, we reshuffle it each epoch, sampling according to the batch size. For further insights, consult [18, 19].

[1]For instances in section 3.1, we reference the code from `https://github.com/maziarraissi/PINNs`, and for the instance in section 3.2, we consult the code from `https://github.com/maziarraissi/HFM`.

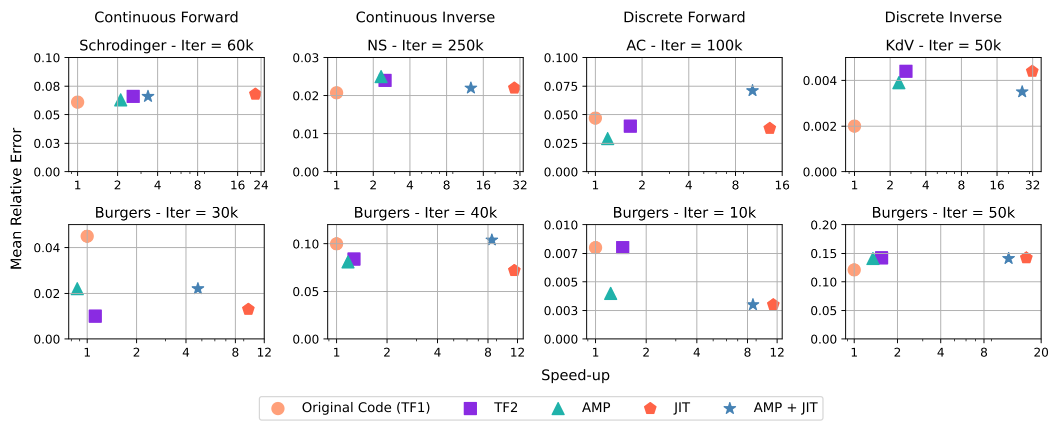


Figure 2: Each subplot denotes a unique problem, with its specific iteration count indicated at the top. The logarithmic x-axis shows speed-up relative to TF1, and the y-axis the mean error, highlighting JIT compiler boosts speed without added error.

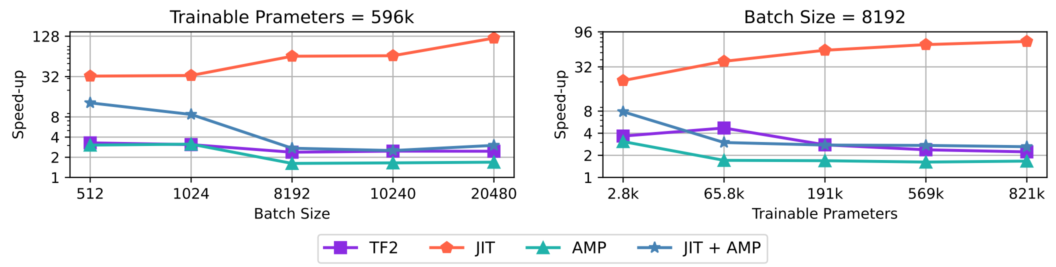


Figure 3: The left plot depicts the escalating efficiency benefits from the JIT compiler with increasing batch sizes. In contrast, the right plot reveals that adjusting the number of trainable parameters by altering the neural network's layer count enhances the JIT's effectiveness, while the speed-up in other configurations diminishes.

We assess speed-up metrics across various settings. Initially, with all other attributes fixed, we solely vary the batch size for a model with 596k trainable parameters. The left plot of Figure 3 reveals that the JIT's efficiency surges with larger batch sizes. At a batch size of 20480, the JIT's speed-up peaks at 119.96, significantly reducing training durations. This is likely due to the JIT compiler's capacity to optimize memory usage of computational graphs, such as through fusion, allowing more room for increased batch sizes. In our subsequent experiment, with a fixed batch size of 8192, we adjusted the number of trainable parameters by modifying the neural network's layer depth. While other configurations witnessed a performance decline, JIT's efficacy rose, as showcased in the right plot of Figure 3. The highest speed-up relative to TF1 reaches 70.99 with over 821k trainable parameters and a batch size of 8192. This section underscores the JIT compiler's advantage, especially for large batch sizes and a high count of trainable parameters.

4 Conclusions

In this package, we underscore the significance of compilers in TensorFlow, demonstrating their capability to boost performance over standard TensorFlow implementations. Through 9 varied examples, we illustrate the versatility of "PINNs-TF2" across diverse challenges. Especially for large batch sizes, the use of XLA and JIT compilers has yielded a remarkable 119x speed-up compared to TensorFlow V1. Interestingly, in our tests, mixed precision reduced the speed-up, suggesting that newer GPUs (i.g. NVIDIA A100) and the adoption of TensorFloat-32 might address this issue [2, 4]. We believe our package will be valuable to research across various domains.

References

[1] Yuyao Chen, Lu Lu, George Em Karniadakis, and Luca Dal Negro. Physics-informed neural networks for inverse problems in nano-optics and metamaterials. *Optics express*, 28 8:11618–11633, 2019.

[2] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41:29–35, 2021.

[3] 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.

[4] Oliver Hennigh, Susheela Narasimhan, Mohammad Amin Nabian, Akshay Subramaniam, Kaustubh Mahesh Tangsali, Max Rietmann, José del Águila Ferrandis, Wonmin Byeon, Zhiwei Fang, and Sanjay Choudhry. Nvidia simnet: an ai-accelerated multi-physics simulation framework. In *International Conference on Conceptual Structures*, 2020.

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

[6] Hyogu Jeong, C.P. Batuwatta-Gamage, Jinshuai Bai, Yi Min Xie, Charith Rathnayaka, Ying Zhou, and Yuantong Gu. A complete physics-informed neural network-based framework for structural topology optimization. *Computer Methods in Applied Mechanics and Engineering*, 2023.

[7] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[8] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[9] 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.

[10] Levi D. McClenny, Mulugeta A. Haile, and Ulisses M. Braga-Neto. Tensordiffeq: Scalable multi-gpu forward and inverse solvers for physics informed neural networks. *ArXiv*, abs/2103.16034, 2021.

[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] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

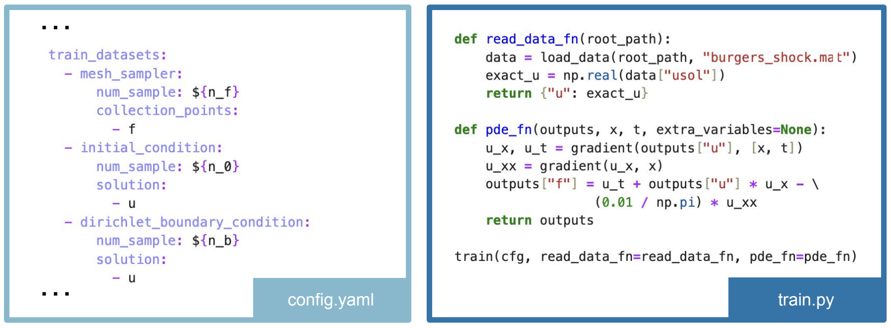
[20] Majid Rasht-Behesht, Christian Huber, Khemraj Shukla, and George Em Karniadakis. Physics-informed neural networks (pinns) for wave propagation and full waveform inversions. *Journal of Geophysical Research: Solid Earth*, 127, 2021.

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

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

[23] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. Enabling fast differentially private sgd via just-in-time compilation and vectorization. In *Neural Information Processing Systems*, 2020.

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



```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
...
```

```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)
```

Figure 4: An example of a basic config file and custom functions for the continuous forward Burgers' equation. Users are required to set up a config file and function for the PINNs-TF2 package to process the data and determine the PDE.

A Appendix

This supplementary document expands on the primary paper in the following ways:

1. Additional information about the PINNs-TF2 Workflow (supplements **Section 2.2**).
2. Presents deeper insights into relative errors for evaluation and training for every problem (supplements **Section 3**).
3. Provides detailed conversations about the examples implemented using our package, including the associated errors and speed-ups (complements **Section 3**).

B PINNs-Torch Workflow

Users should establish a config file interpreted by Hydra, which in turn triggers the relevant classes. They must also delineate functions for fetching data and defining the PDE. An illustration of this setup, using a file and custom functions, can be seen in Figure 4. This package leverages these user definitions to solve the PDE.

C Errors

Error Notations. We use Err as a unified symbol representing either mean squared error (MSE) or sum squared error (SSE). Specifically, Err_0 is the error in initial conditions, Err_b in boundary conditions, Err_c at collection points, Err_s in sampled solutions, and Err^i at time step i.

Relative Errors. We measure errors between predicted and exact solutions using the ℓ_2 norm:

$$\frac{||u_{\text{pred}} - u_{\text{target}}||_2}{||u_{\text{target}}||_2} \tag{1}$$

For variables in the inverse problem, we use:

$$\frac{|\lambda_{\text{pred}} - \lambda_{\text{target}}|}{|\lambda_{\text{target}}|} \tag{2}$$

D Examples

In this section, we summarize examples from our main paper. For detailed insights on the first 8 examples, refer to [17] and for the 3D Navier-Stokes equation in section 3.2, see [18, 19].

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	$u(0, x) = 2\text{sech}(x)$, $v(0, x) = 2\text{sech}(x)$
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 after 60,000 iterations.

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
TF2	0.024	0.106	0.068	0.066	2.62
AMP	0.022	0.103	0.062	0.063	2.11
JIT	0.024	0.110	0.069	0.068	**22.90**
JIT + AMP	0.023	0.109	0.065	0.066	3.38

Continuous Forward Schrodinger Equation. For the nonlinear Schrodinger equation given by:

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

with $x \in [-5, 5]$, $t \in [0, \pi/2]$, and $h(t, x)$ as the complex solution, we partition $h(t, x)$ into its real part u and imaginary part v. Thus, our complex-valued neural network representation is $[u(t, x), v(t, x)]$. The setup is detailed in Table 2.

Prediction discrepancies are gauged against the test data using the relative ℓ_2-norm. Table 3 showcases errors for $h(t, x)$, $u(t, x)$, and $v(t, x)$, plus the average error as mentioned in the primary study.

Continuous Inverse Navier-Stokes Equation. Given the 2D nonlinear Navier-Stokes equation:

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

Table 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 after 250,000 iterations.

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
TF2	0.021	0.023	0.001	0.051	0.024	2.50
AMP	0.028	0.022	0.001	0.050	0.025	2.32
JIT	0.019	0.021	0.001	0.045	0.022	**28.77**
JIT + AMP	0.021	0.026	0.001	0.038	0.022	12.66

where $u(t,x,y)$ and $v(t,x,y)$ are the x and y components of the velocity field, and $p(t,x,y)$ is the pressure, we seek the unknowns $\lambda = (\lambda_1, \lambda_2)$. When required, we integrate the constraints:

$$\begin{aligned} 0 &= u_x + v_y, \\ u &= \psi_y, \\ v &= -\psi_x, \end{aligned} \tag{3}$$

We use a dual-output neural network to approximate $[\psi(t,x,y), p(t,x,y)]$, leading to a physics-informed neural network $[f(t,x,y), g(t,x,y)]$. The setup is detailed in Table 4.

Prediction discrepancies are assessed against a test dataset. Table 5 displays the relative ℓ_2-norm errors for both velocity components and the relative errors for the λ parameters, alongside the average error referenced in the main paper.

Discrete Forward Allen-Cahn Equation. Given the non-linear AC equation:

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

with $x \in [-1,1]$ and $t \in [0,1]$, we adopt Runge–Kutta methods with q stages as described in [17, 5]. The neural network output is:

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

where u^{n+c_j} is data at time $t^n + c_j\Delta t$. The problem setup can be found in Table 6. We extract data from the exact solution at $t_0 = 0.1$ aiming to predict the solution at $t_1 = 0.9$ using a single time-step of $\Delta t = 0.8$. Table 8 shows ℓ_2-norm errors for $u(x)$ at t_1.

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

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	$[u^{n+c_1}(x), \dots, u^{n+c_{q-1}}(x), u^{n+c_q}(x)]$
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*
$t_0 \to t_1$	$0.2 \to 0.8$
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.*

Discrete Inverse Korteweg–de Vries Equation. Given the non-linear KdV equation:

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

we use Runge–Kutta methods with q stages to identify parameters $\lambda = (\lambda_1, \lambda_2)$. The network outputs:

$$[u^{n+c_1}(x), \dots, u^{n+c_{q-1}}(x), u^{n+c_q}(x)]$$

with $u^{n+c_j} = u(t^n + c_j \Delta t, x)$ as data at time $t^n + c_j \Delta t$. Data is sampled at $t^n = 0.2$ and $t^{n+1} = 0.8$. See Table7 for problem details and Table 9 for relative errors of λ_1 and λ_2.

Continuous Forward Burgers' Equation. Given the Burgers' equation:

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

with domain $x \in [-1, 1]$ and $t \in [0, 1]$, and the initial and boundary conditions:

$$u(0,x) = -\sin(\pi x),$$
$$u(t,-1) = 0,$$
$$u(t,1) = 0,$$

we aim to determine the solution $u(t,x)$. Refer to Table 10 for problem details and Table 11 for the relative error of $u(t,x)$.

Table 8: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete forward Allen-Cahn equation after 100,000 iterations.

Method	Relative Error	Mean Relative Error	Speed-up
	$u(t,x)$		
Original Code (TF1)	0.047	0.047	1
TF2	0.040	0.040	1.68
AMP	0.029	**0.029**	1.20
JIT	0.038	0.038	**13.31**
JIT + AMP	0.071	0.071	10.30

Table 9: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete inverse Korteweg–de Vries equation after 50,000 iterations.

Method	Relative Errors		Mean Relative Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.0005	**0.002**	1
TF2	0.002	0.007	0.004	2.72
AMP	0.001	0.007	0.004	2.37
JIT	0.002	0.007	0.004	**31.75**
JIT + AMP	0.001	0.007	0.004	26.09

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	$\text{MSE}_0 + \text{MSE}_b + \text{MSE}_c$

Table 11: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for continuous forward Burgers' equation after 30,000 iterations.

Method	Relative Error	Mean Relative Error	Speed-up
	$u(t,x)$		
Original Code (TF1)	0.045	0.045	1
TF2	0.010	**0.010**	1.12
AMP	0.022	0.022	0.87
JIT	0.013	0.013	**9.60**
JIT + AMP	0.022	0.022	4.73

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 after 40,000 iterations.

Method	Relative Errors		Mean Relative Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.196	0.100	1
TF2	0.009	0.158	0.084	1.27
AMP	0.006	0.155	0.081	1.17
JIT	0.003	0.141	**0.072**	**11.49**
JIT + AMP	0.040	0.167	0.104	8.44

Continuous Inverse Burgers' Equation. Considering the equation:

$$u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0,$$

we aim to both predict the solution $u(t,x)$ and determine the unknown parameters $\lambda = (\lambda_1, \lambda_2)$. For the problem configuration, see Table 12. Relative errors for $u(t,x)$, λ_1, and λ_2 are in Table 13.

Discrete Forward Burgers' Equation. For this problem, we use data from $t_1 = 0.1$ to predict solutions at $t_2 = 0.9$ utilizing Runge-Kutta methods with q stages. The equation is:

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

Here, u^n indicates information at time t^n. For more details, consult Table 15 for the setup and Table 14 for relative errors of $u(t,x)$.

Discrete Inverse Burgers' Equation. Similar to its forward counterpart, we utilize Runge-Kutta methods with q stages. The equation here is given by:

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

The goal is to determine λ_1 and λ_2. Data points are taken from $t = 0.1$ to $t = 0.9$. For more details, see Table 18 for the problem setup and Table 17 for relative errors.

Table 14: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete forward Burgers' equation after 10,000 iterations.

Method	Relative Error	Mean Relative Error	Speed-up
	$u(t,x)$		
Original Code (TF1)	0.008	0.008	1
TF2	0.008	0.008	1.45
AMP	0.004	0.004	1.23
JIT	0.003	**0.003**	**11.39**
JIT + AMP	0.003	**0.003**	8.62

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

Discrete Forward Burgers' Equation	
PDE equations	$f^{n+c_j} = u^{n+c_j} u_x^{n+c_j} - (0.01/\pi) u_{xx}^{n+c_j}$
Dirichlet boundary conditions	$u(t,-1) = u(t,1) = 0$
The output of net	$[u^{n+c_1}(x), \dots, u^{n+c_q}(x), u^{n+1}(x)]$
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*
$t_0 \rightarrow t_1$	$0.1 \rightarrow 0.9$
Loss function	$\text{SSE}_s^0 + \text{SSE}_c^0 + \text{SSE}_b^1$

**Same points used for collocation and solutions.*

Table 16: The problem setup for continuous forward 3D Navier Stokes equation.

Continuous Forward 3D NS	
PDE equations	$e_1 = c_t + (uc_x + vc_y + wc_z) - (1.0/\text{Pec})(c_x x + c_y y + c_z z)$ $e_2 = u_t + (uu_x + vu_y + wu_z) + p_x - (1.0/\text{Re})(u_x x + u_y y + u_z z)$ $e_3 = v_t + (uv_x + vv_y + wv_z) + p_y - (1.0/\text{Re})(v_x x + v_y y + v_z z)$ $e_4 = w_t + (uw_x + vw_y + ww_z) + p_z - (1.0/\text{Re})(w_x x + w_y y + w_z z)$ $e_5 = u_x + v_y + w_z$
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	$\text{MSE}_s + \text{MSE}_c$

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

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

Velocity components are given by $\mathbf{u} = (u, v, w)$, and p is the pressure. For the problem setup, refer to Table 16. Adjustments were made in batch sizes, and hidden layers for parameter training.

Table 17: Comparison of different methods in terms of individual errors, mean error, and speed-up factor for discrete inverse Burgers' equation after 50,000 iterations.

Method	Error		Mean Error	Speed-up
	λ_1	λ_2		
Original Code (TF1)	0.003	0.239	**0.121**	1
TF2	0.004	0.280	0.142	1.55
AMP	0.004	0.278	0.141	1.35
JIT	0.004	0.280	0.142	**15.77**
JIT + AMP	0.004	0.278	0.141	11.87

Table 18: The problem setup for discrete inverse Burgers' equation.

Discrete Inverse Burgers' Equation	
PDE equations	$f^{n+c_j} = \lambda_1 u^{n+c_j} u_x^{n+c_j} - \lambda_2 u_{xx}^{n+c_j}$
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.*