COMPARING PINNS ACROSS FRAMEWORKS: JAX, TENSORFLOW, AND PYTORCH

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

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

ABSTRACT

Physics-Informed Neural Networks (PINNs) have become a pivotal technology for adhering to physical laws and solving nonlinear partial differential equations (PDEs). Enhancing the performance of PINN implementations can significantly quicken the pace of simulations and foster the creation of innovative methodologies. This paper presents 'PINNs-JAX', an innovative implementation that utilizes the JAX framework to leverage the distinct capabilities of XLA compilers. This approach aims to improve computational efficiency and flexibility within PINN applications. We conduct a comprehensive comparison of PINNs-JAX against traditional PINN implementations in widely-used frameworks such as TensorFlow V1, TensorFlow V2, and PyTorch, evaluating performance across a variety of six different examples. These include continuous, discrete, forward, and inverse problems. Our findings indicate that PINNs implemented with JAX outperform in simpler examples, yet TensorFlow V2 presents potential benefits for tackling large-scale challenges, as exemplified by the 3D-Navier Stokes case. To support collaborative development and further research, we have made the source code available to the public at: `https://github.com/rezaakb/pinns-jax`.

1 INTRODUCTION

Physics-informed neural Networks (PINNs) are emerging as a powerful approach for supervised learning that ensures solutions adhere to the laws of physics, especially nonlinear partial differential equations (PDEs) (Raissi et al., 2019), and have been applied across various domains (Haghighat et al., 2021; Rasht-Behesht et al., 2021; Mohammadian et al., 2022).

This paper introduces PINNs-JAX, a novel package based on the JAX framework, known for its just-in-time compilation of Python functions into XLA-optimized kernels (Bradbury et al., 2018). Our study conducts a thorough comparison of JAX with other frameworks to identify the most suitable platform for implementing PINNs. We compare the performance and capabilities of PINNs-JAX with those implemented in TensorFlow V1, PINNs-TF2 within TensorFlow V2 (Bafghi & Raissi, 2023a), and PINNs-Torch in PyTorch (Bafghi & Raissi, 2023b). This evaluation is aimed at providing researchers with a comprehensive comparison to guide their choice of the optimal framework for their specific applications.

Building on prior JAX implementations of Physics-informed Neural Networks (PINNs) (Stanziola et al., 2021; Wang et al., 2023; Sung et al., 2022) and drawing lessons from previous packages (McClenny et al., 2021; Hennigh et al., 2020; Lu et al., 2021), PINNs-JAX leverages XLA and JIT compilers for high-performance numerical computing and the construction of static computational graphs, aiming to enhance PINNs' computational efficiency. Our package is demonstrated through six examples, showcasing significant speed improvements compared to TensorFlow V1 implementations and performance across various frameworks. Furthermore, applying our approach to a large-scale real-world problem sheds light on how batch sizes and the number of trainable parameters influence the performance of the package, suggesting that TensorFlow V2 may be the preferable option for large-scale challenges.

2 PINNs-JAX Package

In this section, we provide a succinct overview of the problem setup utilized in PINNs and outline the functionality of our package by detailing the workflow of our package and the use of the XLA compiler in JAX.

2.1 Problem Setup

We adopt the problem framework from (Raissi et al., 2019), focusing on parametric and nonlinear PDEs of the form:

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

where $u(t,x)$ is the sought solution within domain $\Omega \subset \mathbb{R}^D$, and $\mathcal{N}[.;\lambda]$ is a nonlinear operator dictated by parameter λ. The study explores two main challenges: forward problems, which involve deducing the system's hidden state $u(t,x)$ for given λ (Wang et al., 2022; Raissi et al., 2016), and inverse problems, focused on identifying the parameters λ that best match the observed data (Raissi et al., 2017; Raissi & Karniadakis, 2017; Rudy et al., 2016). The research develops algorithms for both continuous and discrete time models, using new approximators for the former and Runge-Kutta methods for the latter, detailed in (Raissi et al., 2019).

2.2 Implementation

PINNs-JAX Workflow. Our package enhances the PINNs framework from (Bafghi & Raissi, 2023b) for solving PDE-related forward and inverse problems using Hydra (Yadan, 2019), adding support for new boundary conditions, such as inlet, outlet, upper, and lower walls in 2D space and features like prediction saving. In summary, it processes configuration files to set up domains, sampling, and neural network configurations. The process involves reading user-defined PDEs and configurations, compiling conditions, and capturing a computational graph with the XLA compiler for efficient training. For optimization, we employ Optax, a flexible gradient processing and optimization library for JAX, developed by (DeepMind et al., 2020).

XLA Compiler. The JAX system functions as a just-in-time (JIT) compiler, generating code for subroutines that are pure and statically composed—meaning a function is pure if it has no side effects and is statically composed if it can be represented as a static data dependency graph based on a set of primitive functions. It achieves this through high-level tracing in conjunction with the XLA compiler infrastructure (Demeure et al., 2023). JAX builds upon the tracing library used by Autograd (Maclaurin, 2016), which is designed for self-closure and thus recognizes its operations as primitives. Additionally, JAX incorporates Numpy's (van der Walt et al., 2011) numerical functions as part of its primitives, enabling it to generate code for Python functions that use Numpy syntax. This includes supporting arbitrary-order forward and reverse-mode automatic differentiation (Frostig et al., 2018).

3 Experiments

In this section, we explore the experiments designed to assess the performance of PINNs-JAX across different scenarios, with a particular focus on how varying batch sizes impact its effectiveness. Additionally, we benchmark its performance against other popular frameworks like TensorFlow and PyTorch, aiming to identify the most efficient approach for implementing PINNs. Throughout these experiments, the Adam optimizer was exclusively utilized.

Hardware Setup. All experiments were performed using a single NVIDIA Quadro RTX 8000 GPU to guarantee consistency and facilitate reproducibility.

Speed-up Metric. Following (Bafghi & Raissi, 2023a;b), we calculate the median iteration time for different scenarios and compare it to that of the original TensorFlow V1 (TF1) implementations. Speed-up is measured by dividing TF1 times by the times for each scenario.

Table 1: Comparison of average speed-ups across six examples discussed in Section 3.1 using various libraries to TensorFlow V1. The table indicates that JAX outperforms compared to other libraries.

	PyTorch	TensorFlow V2	JAX
Avg. speed-up w.r.t. TensorFlow V1	5.43	18.12	**23.68**

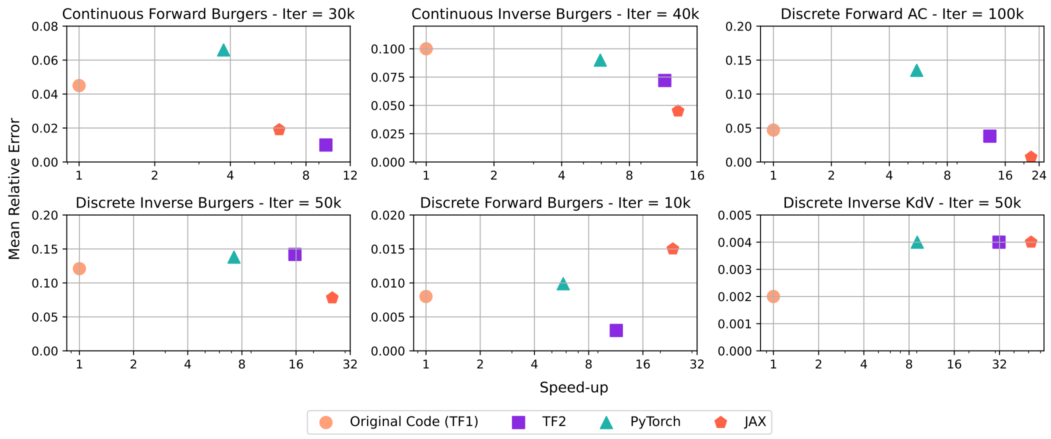

Figure 1: Each subplot represents a distinct problem, with the specific iteration count displayed at the top. The logarithmic x-axis illustrates the speed-up relative to TensorFlow V1, while the y-axis measures the mean relative error. The plot underscores that JAX enhances speed without introducing substantial error in all experiments, with the exception of Continuous Forward Burgers. Additionally, it showcases the capability of the XLA compiler, utilized by both JAX and TensorFlow, to accelerate PINNs more effectively than PyTorch.

Frameworks. Our experiments utilize optimal configurations for TensorFlow V2 and PyTorch as described in (Bafghi & Raissi, 2023a;b). Specifically, TensorFlow V2 models employ the XLA compilers (Sabne, 2020), whereas PyTorch models make use of CUDA Graph (Ramarao, 2022) and TorchScript (DeVito, 2022) for enhanced performance. Additionally, TensorFlow V1 does not utilize any acceleration technologies, and the original setup from the referenced work was adopted. In this paper, we exclude configurations utilizing Mixed Precision and those not employing JIT compilers due to their poor performance in PINNs (Bafghi & Raissi, 2023a;b).

3.1 Evaluation of Various Frameworks

We assess the performance of multiple frameworks across various scenarios, including the Discrete Forward Allen–Cahn (AC) Equation, and Discrete Inverse Korteweg-de Vries (KdV) Equation. Additionally, we explore Burgers' Equation in all configurations: continuous, discrete, forward, and inverse. Each example utilizes static batches. For detailed insights about the examples, refer to (Bafghi & Raissi, 2023a; Raissi et al., 2019). Our evaluations cover TensorFlow V2, PyTorch, and JAX, comparing them against a baseline in TensorFlow V1.

Figure 1 illustrates that employing JAX and TensorFlow V2 results in a significant speed enhancement over TensorFlow V1 and PyTorch when using a single GPU, without compromising accuracy. The most substantial speed-up recorded is 51.94, achieved in the KdV example. This performance boost may be attributed to JAX's functional programming nature, eliminating the need to compute the second derivative and allowing for direct calculation of the third derivative of the output. Although JAX outperformed TensorFlow V2 in most cases, the speed-up observed in the Continuous Forward Burgers example was lower with TensorFlow V2, highlighting the impact of batch size, a topic to be explored in the subsequent section. Table 1 presents the average speed-up of our implemented examples compared to TensorFlow V2 and PyTorch, showing that JAX, on average, achieves a speed-up of 23.68 in the examples mentioned.

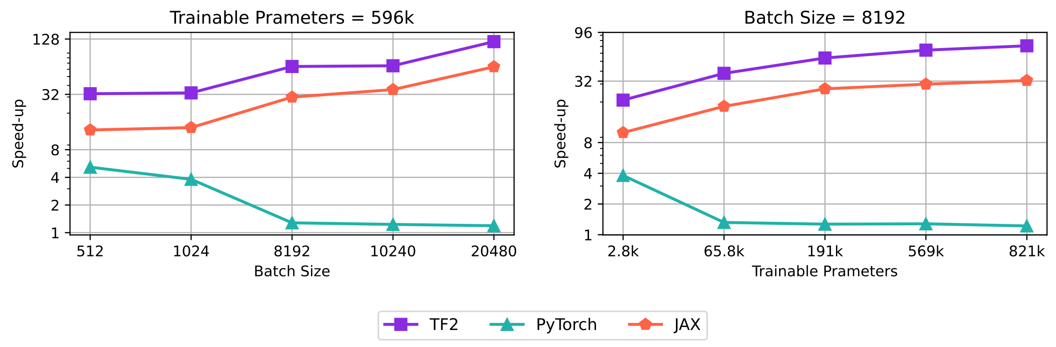

Figure 2: The left plot illustrates the speed-up achieved by various frameworks as batch sizes increase. The right plot demonstrates the performance of these libraries across varying numbers of trainable parameters, achieved by adjusting the depth of the neural network's layers. The findings indicate that TensorFlow V2, utilizing XLA compilers, surpasses the performance of other frameworks. However, JAX displays a competitive speed-up relative to TensorFlow V2.

3.2 Assessing the Impact of Batch Size and Number Trainable Parameters

In this subsection, we analyze the effects of batch size variations and the number of trainable parameters on model efficiency across different frameworks. Our study is centered on simulating three-dimensional physiological blood flow using a realistic intracranial aneurysm (ICA) model, governed by the 3D Navier-Stokes equation. The dataset comprises 29 million data points spanning spatial and temporal domains for five different solutions, employing dynamic batches and random sampling in each iteration. For further information on the example, we refer readers to (Raissi et al., 2018; 2020).

We evaluate speed-up metrics across various configurations, starting by modifying only the batch size while keeping all other factors constant in a model with 596k trainable parameters. The left plot in Figure 2 indicates that while JAX's performance improves with increasing batch sizes, it consistently falls short of TensorFlow V2, demonstrating TensorFlow V2's superior capability in managing large-scale problems. This enhanced performance with TensorFlow V2 and JAX with respect to PyTorch could be attributed to the XLA compiler's ability to optimize memory usage in computational graphs, for instance, through operation fusion, thereby accommodating larger batch sizes. In a subsequent experiment, with a fixed batch size of 8192, we varied the number of trainable parameters by altering the depth of the neural network's layers. Here, while performance generally decreased across PyTorch, the efficiency of XLA compilation exhibited an increase, as depicted in the right plot of Figure 2. This discussion highlights the XLA compiler's benefits, particularly for scenarios involving large batch sizes and numerous trainable parameters, and suggests that TensorFlow V2 is the more suitable library for large-scale computational tasks.

4 Conclusions and Limitations

In this study, we investigate the performance of Physics-Informed Neural Networks (PINNs) across various libraries, including PyTorch, TensorFlow, and JAX. We demonstrate that the use of XLA compilers in TensorFlow and JAX significantly enhances speed, highlighting the critical role of compilers in improving efficiency beyond standard TensorFlow V1 implementations. The evaluation of six examples reveals that JAX achieves a higher average speed-up of 23.68 compared to TensorFlow V2 and PyTorch. However, TensorFlow exhibits superior performance for large batch sizes and a higher number of parameters. Furthermore, by applying PINNs to a range of problems, we showcase the adaptability of the PINNs-JAX package to diverse scientific challenges. Additionally, our findings suggest that the choice of library and computational strategies can be crucial in optimizing PINNs for specific applications, reinforcing the need for a nuanced approach to their implementation. This paper aims to serve as a valuable resource for researchers and practitioners implementing PINNs across different libraries, guiding them in selecting appropriate tools and techniques for their computational needs.

References

Reza Akbarian Bafghi and Maziar Raissi. Pinns-tf2: Fast and user-friendly physics-informed neural networks in tensorflow v2. *ArXiv*, abs/2311.03626, 2023a. URL https://api.semanticscholar.org/CorpusID:265043331.

Reza Akbarian Bafghi and Maziar Raissi. PINNs-torch: Enhancing speed and usability of physics-informed neural networks with pytorch. In *The Symbiosis of Deep Learning and Differential Equations III*, 2023b. URL https://openreview.net/forum?id=nl1ZzdHpab.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/google-deepmind.

Nestor Demeure, Theodore Kisner, Reijo Keskitalo, Rollin C. Thomas, Julian Borrill, and Wahid Bhimji. High-level GPU code: a case study examining JAX and openmp. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*, pp. 1105–1113. ACM, 2023. doi: 10.1145/3624062.3624186. URL https://doi.org/10.1145/3624062.3624186.

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

Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018. URL https://api.semanticscholar.org/CorpusID:4625928.

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. URL https://api.semanticscholar.org/CorpusID:232225919.

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. URL https://api.semanticscholar.org/CorpusID:229180787.

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. doi: 10.1137/19M1274067.

Dougal Maclaurin. Modeling, inference and optimization with composable differentiable procedures. 2016. URL https://api.semanticscholar.org/CorpusID:196066990.

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. URL https://api.semanticscholar.org/CorpusID:232417425.

Mostafa Mohammadian, Kyri Baker, and Ferdinando Fioretto. Gradient-enhanced physics-informed neural networks for power systems operational support. *ArXiv*, abs/2206.10579, 2022. URL https://api.semanticscholar.org/CorpusID:249888947.

Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *ArXiv*, abs/1708.00588, 2017. URL `https://api.semanticscholar.org/CorpusID:2680772`.

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. URL `https://api.semanticscholar.org/CorpusID:4501615`.

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. URL `https://api.semanticscholar.org/CorpusID:3863396`.

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. URL `https://api.semanticscholar.org/CorpusID:51983709`.

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. URL `https://api.semanticscholar.org/CorpusID:57379996`.

Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 367:1026 – 1030, 2020. URL `https://api.semanticscholar.org/CorpusID:210985022`.

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

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. URL `https://api.semanticscholar.org/CorpusID:237347132`.

Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3, 2016. URL `https://api.semanticscholar.org/CorpusID:4575852`.

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

Antonio Stanziola, Simon Arridge, Ben T. Cox, and Bradley E. Treeby. A research framework for writing differentiable pde discretizations in jax. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Nicholas Sung, Jian Cheng Wong, Chinchun Ooi, Abhishek Gupta, Pao-Hsiung Chiu, and Yew Soon Ong. Neuroevolution of physics-informed neural nets: Benchmark problems and comparative results. *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, 2022. URL `https://api.semanticscholar.org/CorpusID:260119439`.

Stéfan van der Walt, Enthought Usa S. Chris Colbert, Gaël Varoquaux, and Inria Saclay. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13: 22–30, 2011. URL `https://api.semanticscholar.org/CorpusID:16907816`.

Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality is all you need for training physics-informed neural networks. *arXiv preprint arXiv:2203.07404*, 2022.

Sifan Wang, Shyam Sankaran, Hanwen Wang, and Paris Perdikaris. An expert's guide to training physics-informed neural networks. *arXiv preprint arXiv:2308.08468*, 2023.

Omry Yadan. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL `https://github.com/facebookresearch/hydra`.