Approximating functions using deep neural networks

Muhammad Hammad and Sharareh Sayyad

Deep neural networks have been extensively employed to investigate various problems in different research fields. Here, we present some of its applications in approximating various functions in $\mathbb{R}$ and $\mathbb{R}^2$. We further discuss the success and limitations of our implemented neural network models in learning convex and non-convex functions.

CONTENTS

I. INTRODUCTION

In recent years, we have witnessed a surge of interest in developing and employing various deep neural network algorithms to address numerous scientific problems. One of the most common ones is approximating functions, in which either their closed form is known or their describing differential equation is given apriori. Among a large variety of functions, convex functions, because of their well-established mathematical and optimization behaviors, were among the first targets to be approximated by neural networks. Approaches like convex neural networks [3], input convex neural networks [1], and groupmax neural networks [12] exemplify the deep learning approaches to learn convex functions.

While learning convex functions in various dimensions is an interesting problem, stepping beyond the set of convex functions and approximating any function is a more exciting goal. However, this task is challenging as the nonconvex functions may exhibit numerous local optima, and optimization on this landscape can be computationally expensive. Here, the convergence to global optima is not guaranteed. Nevertheless, one would expect that adapting neural network models to tackle such problems is worthwhile. This is mainly because, in general, the architecture and the nonlinear activation functions in neural networks allow their models to inherit nonconvex behavior. Thus, under well-designed architectures and well-selected hyperparameters, deep learning models should be able to approximate any function.

The performance of neural network models is highly correlated with the values of their hyperparameters, such as learning rates and their architectures, e.g., the width and depth of neural networks [2]. In addition, exploiting a descriptive loss function and a powerful optimizer has also been shown to play a critical role in obtaining reliable and accurate outputs from training neural network models.

This report focuses on employing feed-forward neural networks, also known as multilayer perceptrons (MLP), to approximate various functions in $\mathbb{R}$ and $\mathbb{R}^2$. Our study evolves around the proposed algorithms to learn functions in Refs. [5, 8, 9]. As the primary attention of these two papers is dedicated to functions in $\mathbb{R}$, we further developed MLP models based on the physics-informed neural networks [11] to explore learning functions in $\mathbb{R}^2$. We report the outcome of our implemented codes on various problems to study the role of hyperparameters and architecture in approximating various functions.

The structure of this report is as follows. Sec. II covers brief details on the ideas and mathematics behind different implemented methods in this package. Sec. III then provides some information on how to use various modules and functions in our package. We continue by pointing out the limitations and subtle considerations regarding our code in Sec. IV. We present our results in Sec. V and conclude in Sec. VI.

II. DETAILS OF ALGORITHMS AND THEIR IMPLEMENTATIONS

This section concisely details multiple algorithms and methods explored in this study. It comprises Algorithm 1 presented in [9], the Imamoto-Tang's algorithm in Ref.[8], the Axon algorithm in Ref. [5], and the Physics-informed neural network [6, 11].

A. Axon algorithm

The Axon algorithm proposed in Ref. [5] is based on the greedy learning method. The algorithms have been shown to successfully approximate various functions. It exhibits exponential decay behavior that is in agreement with theoretical predictions on the loss behavior of such methods. This algorithm has been further demonstrated to perform much better than the vanilla neural networks with L2 loss functions.

B. Physics-informed neural network

Physics-informed neural networks (PINN) [4, 11] are one of the deep learning approaches for finding the solutions of (partial) differential equations using certain forms of loss functions. To be more precise, the differential equation, the boundary/initial conditions, the available exact solutions, and possible target properties of the functions form the total loss function. Here, for practical reasons, instead of the common RELU activation functions, tanh is usually used as the non-linearity. Aside from these subtleties, the rest of the PINNs are similar to other MLP models.

In the following subsection, we present the set of differential equations we studied using PINNs.

1. Ordinary differential equations in $\mathbb{R}$

In Ref. [5], where the Axon algorithm is introduced, the authors explore an ordinary differential equation (ODE). This ODE reads

$$-\varepsilon^2 u''(x) + u(x) = 1, \tag{1}$$

with ε being a constant. The initial conditions are $u(0) = u(1) = 0$ and the exact solution casts

$$u(x) = A \exp\left(\frac{x}{\varepsilon}\right) + B \exp\left(-\frac{x}{\varepsilon}\right) + 1, \tag{2}$$

$$A = \frac{1 - \exp\left(\frac{1}{\varepsilon}\right)}{\exp\left(\frac{2}{\varepsilon}\right) - 1}, \tag{3}$$

$$B = \frac{\exp\left(\frac{1}{\varepsilon}\right) - \exp\left(\frac{2}{\varepsilon}\right)}{\exp\left(\frac{2}{\varepsilon}\right) - 1}. \tag{4}$$

We also study this ODE by PINNs. All implementations for this equation can be found in `src_nna.pins.ode_1d.parabolic_ode_1d`. This directory consists of two files. All modules and classes are stored in the `parabolic_ode_1d.py` file, and the tests are presented in `parabolic_ode_1d.ipynb`.

The second ODE, which we explore by PINNs, is the 1-dimensional diffusion equation given by

$$\varepsilon^2 u''(x) + u(x) = 1, \tag{5}$$

where ε is a constant, and the exact solution reads

$$u(x) = 1 - \cos\left(\frac{x}{a}\right) - \sin\left(\frac{x}{a}\right) \tan\left(\frac{1}{2a}\right). \tag{6}$$

Here, $x \in [0, a]$ and $u(0) = u(a) = 0$. Our PINNs implementations for this equation are in `src_nna.pinns.ode_1d.diffusion_eq`. This directory consists of two files. All modules and classes are stored in the `pdiff_eq_1d.py` file, and the tests are presented in `diff_eq_1d.ipynb`.

2. Partial differential equations in $\mathbb{R}^2$

A generic partial differential equation (PDE) in $\mathbb{R}^2$ are given by

$$A\frac{\partial^2 u(x,y)}{\partial x^2} + B\frac{\partial^2 u(x,y)}{\partial x \partial y} + C\frac{\partial^2 u(x,y)}{\partial y^2} + D\left(x, y, u(x,y), \frac{\partial u(x,y)}{\partial x}, \frac{\partial u(x,y)}{\partial y}\right) = 0, \tag{7}$$

where A, B, and C are constants. The above partial equation is called elliptic when $B^2 - 4AC \leq 0$; it is a parabolic PDE when $B^2 - 4AC = 0$, and we would refer to them as hyperbolic PDEs when $B^2 - 4AC \geq 0$.

For each of these three types of PDEs, we solve one PINN. The MLP model and the early stopping codes, which are used for solving our PDEs, are presented in `src_nna.pinns.nn_pinns`. A collection of equations and associated analytical solutions [10] that are explored in our project. Here is a brief overview of these PDEs.

a. Elliptic PDE. To be more precise, for the elliptic PDE, we study

$$\frac{\partial^2}{\partial^2 x} u(x,y) + \frac{\partial^2}{\partial^2 y} u(x,y) = 0. \tag{8}$$

We solve this equation in the (x-y) Cartesian coordinate with two sets of boundary conditions.

Solution based on the first boundary condition: The boundary conditions read $u(0,y) = u(1,y) = u(x,0) = 0$ and $u(x,1) = \phi(x)$. The exact solution then yields

$$u(x,y) = \sum_{k=1}^{\infty} c_k \sin(k\pi x) \sin(k\pi y), \tag{9}$$

$$c_k = \frac{g_k}{\sinh(k\pi)}, \tag{10}$$

$$g_k = 2\int_0^1 \phi(x)\sin(k\pi x)dx. \tag{11}$$

The implementation of this PDE is stored in `src_nna.pinns.elliptic_pde`. Different classes and modules are implemented in `elliptic_pde_2d.py`. The performance analysis for various $g(x)$ functions and hyperparameters are collected in `elliptic_pde_2d.ipynb`.

Solution based on the second boundary condition: The boundary conditions for $0 \le x \le L_x$ and $0 \le y \le L_y$ read $u(0,y) = u(L_x,y) = u(x,0) = t_1$ and $u(x,L_y) = t_m \sin(\pi x)$. The analytical solution then becomes

$$u(x,y) = t_1 + (t_m - t_1)\sum_{n=1}^{\infty} c_n \frac{\sin\left(k\pi\frac{x}{L_x}\right)\sinh\left(k\pi\frac{y}{L_y}\right)}{\sinh\left(k\pi\frac{L_y}{L_x}\right)}, \tag{12}$$

$$c_n = \frac{1+(-1)^{n+1}}{n}. \tag{13}$$

We stored the implementation of solving this PDE using PINNs in `src_nna.pinns.pde_2d`. The main modules and functions can be found in `heat_eq_2d.py`, and various experimentations are saved in `heat_eq_2d.ipynb`.

b. Parabolic PDE. For the parabolic PDE, we explore

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial t^2}, \quad 0 \le x < 1, \tag{14}$$

where α acquires a constant value. The initial and boundary conditions are set as $u(0,t) = u(1,t) = 0$ and $u(x,0) = \phi(x)$. The analytical solution for this PDE reads

$$u(x,t) = \sum_{n=1}^{\infty} A_n e^{-(n\pi\alpha)^2 t}\sin(n\pi x), \tag{15}$$

$$A_n = 2\int_0^1 \phi(x)\sin(n\pi x)dx. \tag{16}$$

The PINN solution for this PDE is presented in `src_nna.pinns.parabolic_pde`. The main implementation is given in `parabolic_eq_1p1.py`. Further, analysis and plots are gathered in `parabolic_eq_1p1.ipynb`.

c. Hyperbolic PDE. As an example of Hyperbolic PDEs, we study

$$\frac{\partial^2 u}{\partial^2 t} = \alpha^2 \frac{\partial^2 u}{\partial^2 x}, \tag{17}$$

with α being a constant. The boundary and initial conditions read $u(t,0) = u(t,1) = 0$, $u(0,x) = \phi(x)$ and $\partial u(0,x)\partial t = \psi(x)$. The analytical solutions then cast

$$u(t,x) = \sum_{n=1}^{\infty}\left[A_n\cos(n\pi\alpha t)\sin(n\pi x) + B_n\sin(n\pi\alpha t)\sin(n\pi x)\right], \tag{18}$$

$$A_n = 2\int_0^1 \phi(x)\sin(n\pi x)dx, \tag{19}$$

$$B_n = \frac{2}{\pi\alpha}\int_0^1 \psi(x)\sin(n\pi x)dx. \tag{20}$$

The PINNs codes for solving and exploring Eq. (17) can be found in `src_nna.pinns.hyperbolic_pde`. The main classes and modules are stored in `hyperbolic_pde_1p1.py` and the Jupyter notebook `hyperbolic_pde_1p1.ipynb` holds numerous figures on the performance of our PINNs.

C. Optimal approximtion

Here, we address a fundamental question in function approximation: given a convex univariate function and a fixed number of linear segments, what is the minimal possible approximation error?

We explore this question through the implementation and comparison of two key algorithms: Algorithm 1, proposed by [9], and the [8].

Our experiments focused on three convex functions: e^x, x^2, and x^3. We compared the performance of Algorithm 1 and the Imamoto-Tang algorithm for approximating these functions using 2, 3, 5, and 10 linear segments. The objective is to identify which algorithm minimizes the approximation error, particularly in cases where the number of segments is constrained.

1. *Algorithm 1:Optimal Piecewise Linear Approximation*

Algorithm 1, proposed by [9], is an iterative method for finding the optimal piecewise linear approximation of convex univariate functions. The algorithm's core principle is to adjust segment endpoints to equalize approximation errors, aiming to minimize the overall approximation error. This error is bounded by an equation that demonstrates the algorithm's effectiveness in reducing error as the number of segments increases.

The optimal approximation error $\Delta(f_n^*)$ is bounded as follows:

$$\frac{(b-a)^2 \cdot \min\limits_{x\in[a,b]} f''(x)}{16} \cdot \frac{1}{n^2} \leq \Delta(f_n^*) \leq \frac{(b-a)^2 \cdot \max\limits_{x\in[a,b]} f''(x)}{16} \cdot \frac{1}{n^2} \tag{21}$$

This iterative process ensures that the approximation error decreases at a rate proportional to $\frac{1}{n^2}$, where n is the number of linear segments used in the approximation.

The algorithm follows these key steps:

1. Initialize sub-intervals evenly across the domain.
2. For each sub-interval, compute the optimal linear approximation.
3. Iteratively adjust the interior endpoints of segments to equalize approximation errors across all segments.
4. Repeat step 3 until the difference between maximum and minimum approximation errors is below a specified threshold.

Our implementation leverages `NumPy` for efficient numerical computations and uses `SciPy`'s `optimize` utility for optimization tasks. The core functions of the implementation are:

- `optimal_approximation_errors`: Calculates optimal approximation errors for given segment boundaries.
- `optimal_segment_param`: Computes optimal parameters for each linear segment.
- `find_optimal_boundaries`: Objective function for optimization.
- `optimal_approximation_algorithm`: Main function implementing the iterative process.

The algorithm is guaranteed to converge and provides both upper and lower bounds for the approximation error [9].

2. *Imamoto-Tang's Algorithm*

The algorithm in Ref. [8] is designed to find optimal piecewise linear approximations of convex functions. This classical approach offers an efficient method for approximating complex convex functions using a series of connected linear segments.

The algorithm iteratively refines a set of pivot points across the function's domain to minimize approximation errors. It determines optimal linear segments at these pivot points, resulting in a piecewise linear approximation that closely follows the original function.

Our implementation of this algorithm consists of the following key steps:

1. Initialize pivot points evenly spaced across the domain $[a, b]$.
2. For each pivot point, determine the optimal linear segment using:

- Function value $f(c)$.
- Slope $f'(c)$.

3. Compute segment endpoints by finding intersections of adjacent line segments.
4. Calculate approximation errors at these endpoints.
5. Update pivot point locations using a heuristic estimate.
6. Repeat steps 2-5 until the difference between maximum and minimum approximation errors is below a specified threshold ϵ.

Our Python implementation includes two main components:

- `PW_approx` class: Represents the piecewise linear approximation.
- `imamoto_optimal_approximation` function: Implements the core algorithm.

3. PWL Search Algorithm

The Imamoto-Tang algorithm, presented below, is an iterative method for finding the optimal piecewise linear (PWL) approximation of convex functions. It is designed to minimize the maximum approximation error across all segments while maintaining computational efficiency.

Algorithm 1: Imamoto-Tang Algorithm

Input: Function $f(x)$, number of segments N, boundary points α_0 and α_N
Output: Optimal piecewise linear approximation $g(x)$
Set $j \leftarrow 0$;
Set $\Delta \leftarrow 1$;
for $i = 0$ **to** $N - 1$ **do**
$t_{i,0} \leftarrow \alpha_0 + (i/N + 1/2)(\alpha_N - \alpha_0)$;
end
repeat
for $i = 1$ **to** $N - 1$ **do**
$\alpha_{i,j} \leftarrow \frac{f(t_{i,j}) - f(t_{i-1,j}) + f'(t_{i,j})t_{i,j} - f'(t_{i-1,j})t_{i-1,j}}{f'(t_{i,j}) - f'(t_{i-1,j})}$;
end
$\alpha_{0,j} \leftarrow \alpha_0$, $\alpha_{N,j} \leftarrow \alpha_N$;
for $i = 0$ **to** N **do**
$\varepsilon_{i,j} \leftarrow g_{i,j}(\alpha_{i,j}) - f(\alpha_{i,j})$;
where $g_{i,j}(x) = f'(t_{i,j})(x - t_{i,j}) + f(t_{i,j})$;
end
if $\max|\varepsilon_{i,j}| - \min|\varepsilon_{i,j}| < \delta$ **then**
$\varepsilon \leftarrow \frac{1}{2} \cdot \max / \min\{\varepsilon_{i,j}\}$;
$g(x) \leftarrow f'(t_i)(x - t_i) + f(t_i) - \varepsilon$;
break;
else
if $\max|\varepsilon_{i,j}| > \max|\varepsilon_{i,j-1}|$ **then**
$j \leftarrow j - 1$;
$\Delta \leftarrow \Delta/2$;
else
for $i = 0$ **to** $N - 1$ **do**
$d_{i,j} \leftarrow \Delta \cdot \frac{\varepsilon_{i+1,j} - \varepsilon_{i,j}}{\varepsilon_{i+1,j} + \varepsilon_{i,j}} \cdot \left(\frac{1}{\alpha_{i+1,j} - t_{i,j}} + \frac{1}{t_{i,j} - \alpha_{i,j}}\right)$;
end
for $i = 0$ **to** $N - 1$ **do**
$t_{i,j+1} \leftarrow t_{i,j} + d_{i,j}$;
end
$j \leftarrow j + 1$;
end
end
until *convergence*;

This algorithm demonstrates the iterative nature of the Imamoto-Tang method. It begins by initializing parameters and segment endpoints, then enters a loop that refines the approximation until convergence. Key steps include computing $\alpha_{i,j}$ values for each segment, calculating error values $\varepsilon_{i,j}$, and adjusting segment endpoints based on these errors. The algorithm employs a dynamic adjustment process, reducing the step size Δ when necessary to fine-tune the approximation.

The convergence criterion, $\max|\varepsilon_{i,j}| - \min|\varepsilon_{i,j}| < \delta$, ensures that the algorithm terminates when the difference between maximum and minimum errors falls below a specified threshold. While this method often yields effective results in practice, it's important to note that it lacks a formal proof of convergence. Nevertheless, this algorithm provides a practical approach to function approximation, balancing accuracy with computational efficiency.

In our experiments, we set the convergence threshold δ to 10^{-4}, matching the tolerance used for Algorithm 1 to ensure a fair comparison.

Key features of the Imamoto-Tang algorithm include:

- Generally faster computation times compared to Algorithm 1.
- Often achieves slightly lower approximation errors.

III. USAGE

To facilitate using our package to approximate various implemented functions, we provide a concise summary of how to call various modules/functions to get the code running. We emphasize that a small description of each function is given inside the function, which should ease understanding and employ our package.

A. Physics-informed neural networks

In all ".py" files in directories gathered in `src_nna.pinns`, one can find `exact_solution`, which computes the analytical solutions for the specific differential equation. This ".py" consists of other modules to calculate `loss` functions and `train` the network. To train the network, one can simply call `run_training`.

Inside the `loss` modules, various loss contributions are evaluated, including the physical loss originating from the differential equation, the MSE loss with the analytical solution, and the MSE loss for the boundary/initial conditions. These loss terms will be weighted and summed together by `lambda` parameters. The selected values of these weights work fine for the provided parameters. However, these values should be adjusted to deploy our implementation on new functions. One can do so by first setting `verbose=True` and `plotting=True` and checking how each component of the loss function evolves as a function of iteration steps.

In cases where enforcing the boundary conditions is strict, especially in $\mathbb{R}^2$, choosing a large boundary point number in `n_data_per_bc` is helpful. To reduce the computational effort, we also randomly selected `Nc` points inside the $\mathbb{R}^2$ regions of our interest. To see the distribution of internal and boundary points, set `plotting=True`. Choosing `Nc`>500 usually suffices for the selected set of values in the package. However, we emphasize that having a reasonable balance between `Nc` and `n_data_per_bc` is recommended for getting similar scales of loss contributions.

The MLP models consist of various numbers of hidden layers set by `num_layers` and different hidden units adjusted by `num_hidden`. Selecting a correct combination of these two parameters expedites the convergence and improves the final accuracy of the approximation. In the ".ipynb" files, we explored the impact of these two parameters on the performance of PINNs.

We have provided various functions to be used as boundary conditions (ϕ, ψ). Examples of implemented functions are sin, cos, x^2, $\sqrt{x}$, x and constant c functions. It is also possible to pass different inline functions to our code instead of the provided functions.

B. Optimal approximation

This section outlines the usage instructions for both Algorithm 1 and the Imamoto-Tang algorithm.

1. Algorithm 1: Optimal Piecewise Linear Approximation

To use Algorithm 1 for optimal piecewise linear approximation:

1. Import the algorithm module from the project's source:
   - `import src_nna.optimal_approx.algorithm_1 as alg1`
2. Select a target function from the available options ($e^{\hat{x}}$, $\hat{x2}$, or $\hat{x3}$):
   - `target_fn = alg1.exp` (for $e^{\hat{x}}$)
   - `target_fn = alg1.x_sq` (for $\hat{x2}$)
   - `target_fn = alg1.x_cube` (for $\hat{x3}$)
3. Define the domain boundaries $[a, b]$ for the approximation:
   - `a, b = 0, 1` (domain $[a, b]$)
   - `n_segments = 10`
4. Specify the desired number of segments for the piecewise linear approximation:
   - `n_segments = 10` (number of segments)
5. Call the optimal approximation algorithm with the chosen parameters:
   - `g, optimal_approx_errors = alg1.optimal_approximation_algorithm( target_fn, a, b, n_segments)`
6. The algorithm returns two main outputs:
   - `g`: The optimal piecewise linear approximation function.
   - `optimal_approx_errors`: An array of approximation errors for each segment.
7. Optionally, calculate the mean approximation error if needed:
   - `mean_delta_S = np.mean(optimal_approx_errors)`
8. Optionally, use the provided plotting function to visualize the results, comparing the original function with its approximation:
   - `plot_final(x, y_f, y_g, label="Function Label")`

2. Imamoto-Tang Algorithm

To use the Imamoto-Tang algorithm:

1. Define the target function f_x and its derivative f'_x.
2. Specify the domain $[\alpha_0, \alpha_N]$ and number of segments N.
3. Call `immamoto_optimal_approximation(f_x, f_prime_x, alpha_0, alpha_N, N)`.
4. The returned g object can be used to evaluate the approximation.

This approach allows for easy implementation and analysis of the optimal piecewise linear approximation for various functions within the specified domain.

IV. LIMITATIONS AND CONSIDERATIONS

In this section, we present further considerations and restrictions on the applicability extent of our code to better assess the presented algorithms and our implementations.

A. Physics-informed neural networks

In most vanilla neural network models, well-selecting hyperparameters such as the model's learning rates, depth, and width are critical in obtaining reasonable results. Such care should also be taken into account when training PINNs. However, the main challenge in training PINNs is finding a suitable set of weights such that the optimizer, set to be `Adam`, can navigate the landscape and approach global optima. The total loss in PINNs is a combination of various contributions; each of them has its own order of magnitude and might not be comparable. As a result, one-scale optimizers, such as Adam, may encounter difficulties and quickly be trapped in local minima that are far from global ones. This is the difficulty we frequently encounter in approximating functions in $\mathbb{R}^2$. The shortcomings of vanilla PINNs in exploring high-dimensional functions and multi-scale loss functions have been discussed in the literature; see Ref. [7].

B. Optimal approximation

Compared to the original Imamoto-Tang algorithm, our implementation shows promising results in terms of convergence:

- **Convergence Achievement:** Our algorithm successfully converged for all tested functions, including x^3, across various segment counts (2, 3, 5, and 10).
- **Performance Consistency:** The Mean ΔS values in our implementation demonstrate robust performance across different function complexities.
- **Computational Efficiency:** Our implementation shows competitive execution times, particularly for higher segment counts.
- **Theoretical Bounds:** The inclusion of theoretical upper and lower bounds in our results provides additional context for the algorithm's performance, demonstrating that our implementation operates within expected error margins.

These results address the convergence challenges noted in previous implementations, particularly for complex functions like x^3 at higher segment counts. Our findings suggest a reliable approximation method that may be applicable to a broad range of convex functions. Further research and testing with a wider variety of functions could provide additional insights into the algorithm's capabilities and limitations.

	Function	n	Mean ΔS	Theoretical Upper Bound	Theoretical Lower Bound	Time (Alg1) [s]	Mean ε	Time (Imatomotang) [s]
0	e(x)	2	0.02635	0.0425	0.0156	0.019	0.05270	0.000
1	e(x)	3	0.01170	0.0189	0.0069	0.019	0.02340	0.001
2	e(x)	5	0.00421	0.0068	0.0025	0.075	0.00842	0.183
3	e(x)	10	0.00107	0.0017	0.0006	0.311	0.00210	0.026
4	x**2	2	0.12500	0.1250	0.1250	0.037	0.25000	0.000
5	x**2	3	0.05556	0.0556	0.0556	0.024	0.11111	0.001
6	x**2	5	0.02000	0.0200	0.0200	0.082	0.04000	0.901
7	x**2	10	0.00500	0.0050	0.0050	0.156	0.01000	0.021
8	x**3	2	0.04486	0.0938	0.0000	0.021	0.08973	0.000
9	x**3	3	0.01946	0.0417	0.0000	0.065	0.03892	0.001
10	x**3	5	0.00687	0.0150	0.0000	0.242	0.01374	0.003
11	x**3	10	0.00188	0.0038	0.0000	0.362	0.00338	0.025

Table I. Comparison of Mean ΔS, Theoretical Bounds, and Computation Times for Algorithm 1 and the Imamoto-Tang Algorithm

1. *Implementation of ReLU Network*

We implemented a ReLU neural network inspired by the architecture described in the original research paper. The network consists of one hidden layer, with the number of neurons varying depending

on the complexity of the function being approximated. We employed Kaiming initialization to initialize the weights, and stochastic gradient descent (SGD) with momentum to optimize the model.

We tested the ReLU network on three functions: i) $f(x) = e^x$, ii) $f(x) = x^2$, and iii) $f(x) = x^3$. To assess the network's performance across varying levels of complexity, we used 2, 3, 5, and 10 segments for each function approximation.

2. Discussion and Comparative Analysis

Our implementation of the ReLU network for function approximation yielded results that differed from the original paper. Below, we summarize our findings:

a. Approximation Behavior.—

1. **Exponential Function ($f(x) = e^x$):**
   - 2 and 5 segments: Close approximation in the lower half, minor deviations in the upper half
   - 3 and 10 segments: Significant deviation, especially in the upper half
2. **Quadratic Function ($f(x) = x^2$):**
   - Accuracy improved with more segments; 10-segment approximation nearly matched true function
   - Small deviations at domain extremes, particularly with fewer segments
3. **Cubic Function ($f(x) = x^3$):**
   - Good performance even with 2 segments, improving further with more segments

b. Key Differences from Original Paper.—

- **Approximation Accuracy:** Varied across functions and segment counts, unlike consistently high accuracy reported in the paper
- **Function-Specific Performance:** Exceptional performance on $f(x) = x^3$ across all segment counts, contrasting with the paper's findings
- **Segment Count Impact:** Non-linear relationship between segment count and accuracy, differing from the paper's consistent improvement with higher segment counts

While our implementation followed the general principles outlined in the original paper, the results show significant variations. These differences highlight:

- Sensitivity of neural networks to implementation details
- Importance of precise replication of all model aspects, including hyperparameters and training procedures, to achieve comparable results

V. RESULTS

After presenting the mathematical foundations and implementation limits and how to use our package, we continue the report by briefly discussing some of our results on approximating various functions in $\mathbb{R}$ and $\mathbb{R}^2$.

A. Axon Algorithm

The original implementation of this algorithm was available on Github. We have stored this implementation in our package in the directory `src_nna.axon_github`. We have realized small incompatibilities in the original code and the installed packages on our system. We fixed these issues in codes saved in `src_nna.axon_approx.src`. The original vanilla neural networks were prepared using Pytorch. We further experiment with reimplementing this code in TensorFlow and Jax to see whether the internal

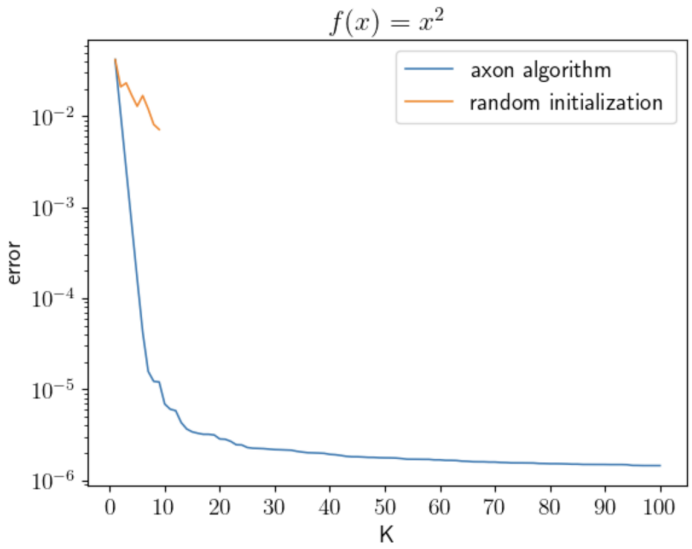

Figure 1. Comparison of the error approximation for learning x^2 function obtained from the Axon algorithm and vanilla neural network with random initialization. The neural network for random initialization is implemented using the Jax package.

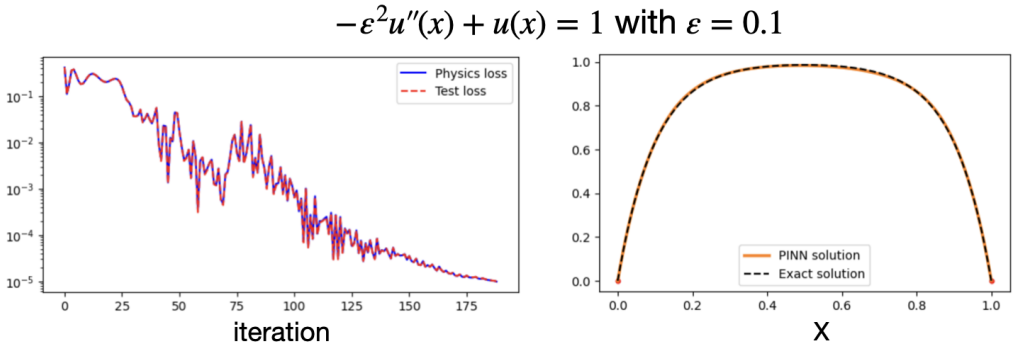

Figure 2. Solution of Eq. (1) with PINNs. The model consists of 6 hidden layers and 32 hidden units. The learning rate is set to 0.01. Dashed black lines depict analytical solutions.

optimizers within these packages can perform better in approximating functions. We have collected these codes by `axon_model_(jax/tensorflow/torch).py`. The overall conclusion was similar to what was stated in Ref. [5]. However, using our code implemented by Jax, we witness an initial trend in lowering the loss function; see Figure 1. However, this trend did not persist in reducing the loss values to an acceptable value. We thus merely rely on the results performed in the original paper as our implementations did not reach better conclusions.

Further tests on the performance of random initializations and function approximations using our Jax-, TensorFlow-, and Pytoch-based codes can be found in `tests/axon_approx` directory.

B. Physics-informed neural networks

1. *Ordinary differential equations in* $\mathbb{R}$

The solution to Eq. (1) for $\varepsilon = 0.1$ is plotted in Fig.2; the notebook containing these results is in`src_nna.pinns.ode_1d.parabolic_ode_1d` under the name `parabolic_ode_1d.ipynb`. This solution is obtained for a network with a learning rate equal to 0.01 and with (hidden layers, hidden units) = (6, 32). The Axon algorithm has solved a similar equation, and the result with a loss value near 10^{-4} is shown in Fig.3. Comparing the obtained error and number of iterations, we see that PINNs reach the

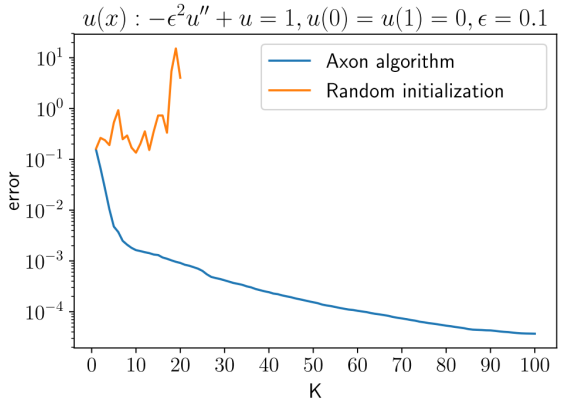

Figure 3. Adapted from Fig. 10(a) in Ref. [5].

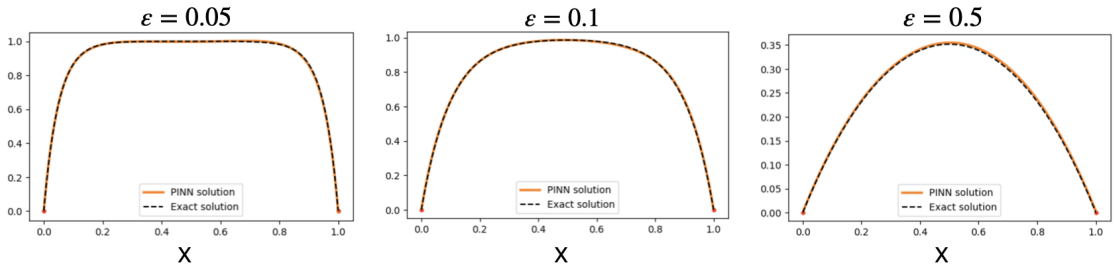

Figure 4. Solution of Eq. (1) with PINNs for three different values of $\varepsilon \in \{0.05, 0.1, 0.5\}$. The model consists of 4 hidden layers and 16 hidden units. Dashed black lines depict analytical solutions.

loss value around 10^{-4} in nearly 100 iterations, similar to the Axon results. Comparing the loss values for PINNs and Axon methods shows that the loss function in Axon is monotonically decreasing as opposed to the PINNs result. This can be attributed to i) external averaging of more than 20 different experiments in obtaining the values of error in the Axon algorithm and ii) the presence of multiple loss contributions in the loss functions of PINNs, which might not all converge as iteration steps increase.

To further explore how well our network approximates $u(x)$ for various values of ε, we present Fig. 4. It is clear that for all values of $\varepsilon \in \{0.05, 0.1, 0.5\}$ both boundary conditions (red points) and analytical solutions (dashed lines) are well-approximated by PINNs; see orange lines. Our numerical experiments reveal the PINNs should contain more parameters, larger depth, and width to approximate $u(x)$ for larger values of ε. It is clear that upon increasing the parameters of the model, more points on the x-axis should be provided to prevent overfitting. We implement early stopping in our code to terminate the code when the loss stagnates for over `patience=100` iterations. We also introduce a scheduler for adjusting the learning rate during training.

Hence, we conclude that the performance of PINNs is comparable to the Axon algorithm in terms of iteration steps needed to reach the converged solutions. The flexibility of PINNs allows us to approximate the function upon varying the parameter ε.

The PINNs solution to the second ODE in our project, given in Eq. (5), is shown in Figs.5 and 6; the notebook containing these plots are saved in `src_nna.pinns.ode_1d.diffusion_eq` under the name `diff_eq_1d.ipynb`.

Figure 5 exhibits the values of loss functions as a function of iteration steps in the top row and displays the approximated function satisfying Eq. (5) with $\varepsilon = 0.1$ in the bottom row. Dashed black lines depict exact solutions. The numbers of layers and hidden units are, respectively, set to 4 and 32 in the left column and to 6 and 16 in the right column. Both models approximate the function with loss values below 10^{-2}, and the approximate functions (solid orange line) agree with the exact solutions. The wider model with 32 hidden units reaches loss values below 10^{-3} during training iteration. This may reflect that aside from the importance of depth, which is commonly believed to be important in learning steps, the width of the network also plays some role in our cases. The approximate function is not a convex/concave function beyond the scope of most concave-based learning schemes.

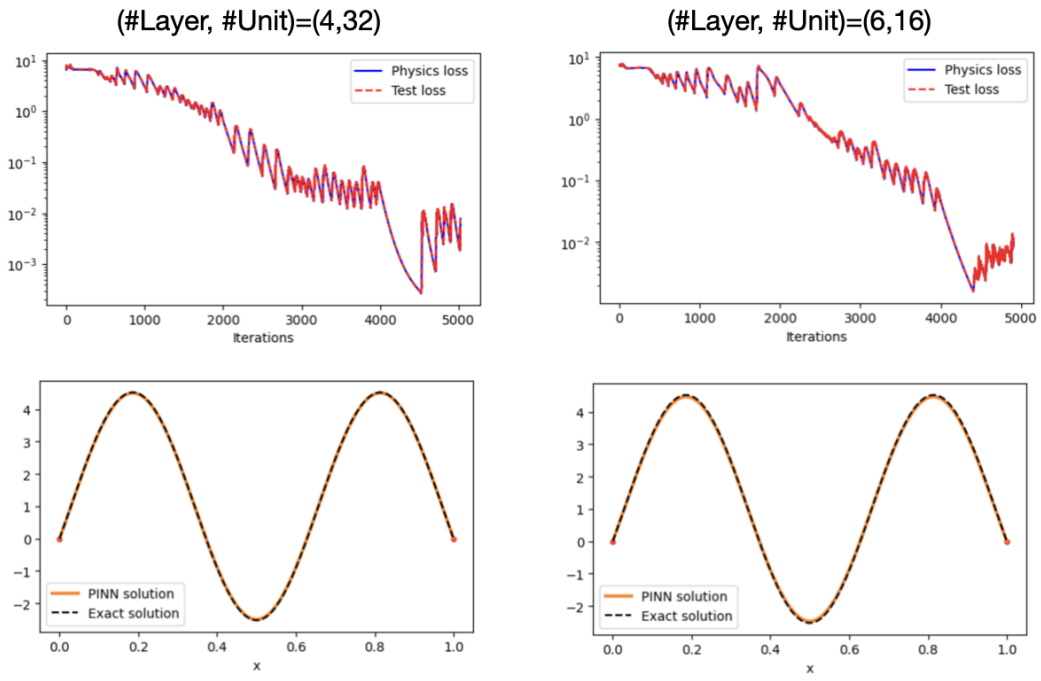

Figure 5. Solution of Eq. (5) using PINNs at $\varepsilon = 0.1$. Dashed black lines depict analytical solutions.

Adjusting the ε results in varying the oscillatory behavior of $u(x)$ as can be seen from the bottom row of Fig. 6 for $\varepsilon = 0.05$(left) and 0.2(right) columns. The model is built from 4 hidden layers and 32 hidden units, and the learning rate is selected to attain the best convergence around 10^{-5}. Similar to other PINN simulations, the loss function fluctuates in both cases; see the top row. However, the loss values vary smoother when learning rates are smaller. This is not unexpected as lower learning rates learning rate allows the model to explore the landscape of the loss function in a smoother manner, with smaller steps, preventing rapid changes in the loss function.

Based on these results, we conclude that approximating functions in $\mathbb{R}$ independent of their convexity behavior is achievable by proper choice of loss functions and hyperparameters of the neural network model.

2. *Partial differential equations in* $\mathbb{R}^2$

Let us now explore the performance of PINNs in approximating functions in $\mathbb{R}^2$.

a. Elliptic PDE.— We solve Eq. (8) using the first type of its boundary conditions, see Sec. II B 2 a, with PINNs. Figure 8 exhibits the loss function, the exact solution, the approximate solution, and the difference between the analytical and approximate solutions from left to right. The PINN captures the overall behavior of the analytical functions with the total loss values being below 10^{-2}. Comparing the exact and learned solutions (middle two columns) shows that the most challenging parts of $u(x, y)$ to be approximated by PINNs are around the corners where, by boundary conditions, abrupt changes are imposed. Near those regions when $\phi = 3$ (top row), the function rapidly varies from zero on the $x = 0, x = 1$ boundaries to three on the $y = 1$ boundary. In the bottom row, the imposed boundary condition is asymmetric in x, which also reflects that the major discrepancies between the approximated and exact solutions reside near $x = 1$ and $y = 1$ region. Approximating other functions as boundary functions is also feasible; however, the inaccuracies near the top corners of the x-y plane usually persist. Further details on experimenting with other functions can be found in the Jupyter notebook `elliptic_pde_2d.ipynb` in the directory `src_nna.pinns.elliptic_pde`.

Next, we present our results for approximating the solution of Eq. (8) when imposing the second type of boundary conditions see Sec. II B 2 a. Note that the major difference between these two boundary conditions is assigning a nonzero temperature to the three different edges and a sinus function to the top edge at $y = 1$ in the second boundary condition. Figure 8 present the PINNs simulation

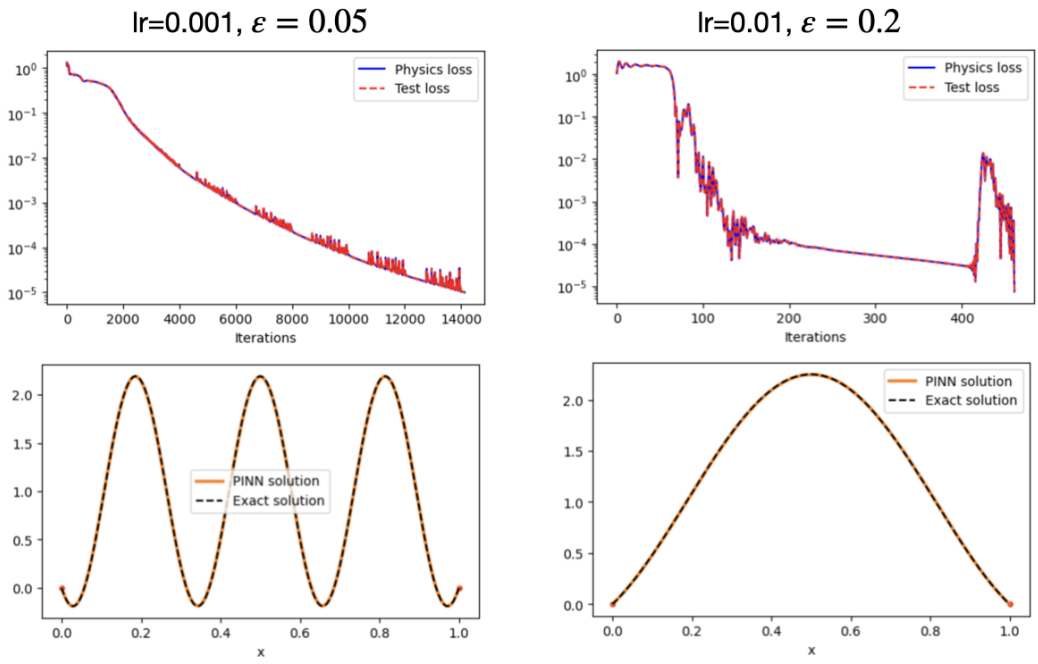

Figure 6. Solution of Eq. (5) using PINNs for $(\varepsilon, \text{learning rate}) = (0.05, 0.001)$[left column] and $(0.2, 0.01)$[right column]}. The model contains four hidden layers and 32 hidden units. Dashed black lines depict analytical solutions.

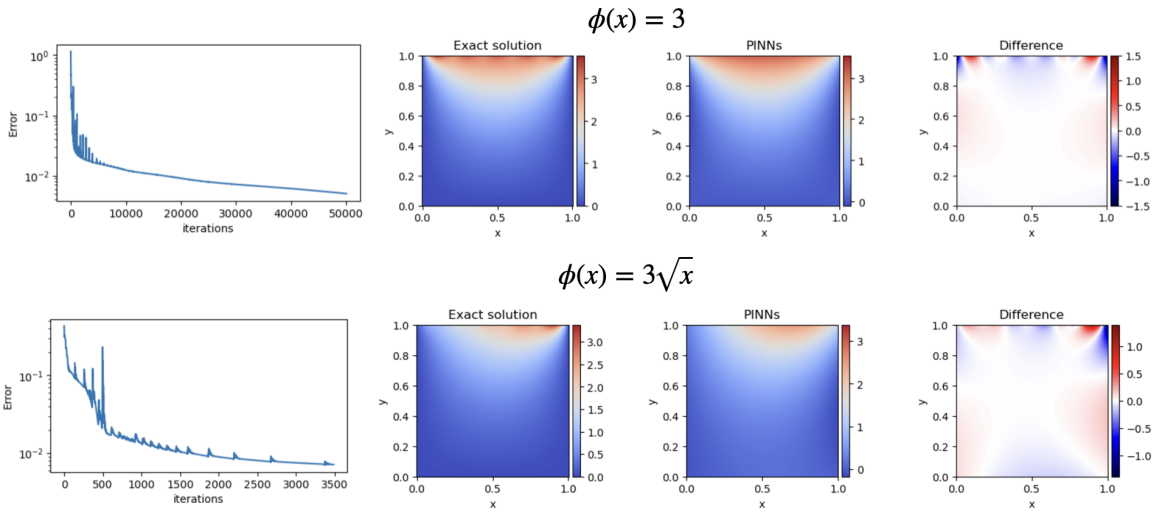

Figure 7. Solving Eq. (8) using PINNs by imposing the first type of boundary conditions inSec. II B 2 a. The model has six hidden layers and 16 hidden units. The boundary function is $\phi = 3$(top row) and $3\sqrt{x}$ (bottom row). The learning rate is 5×10^{-3}.

in approximating $u(x, y)$ when $t_1 = 0.1$ and $t_2 = 0.1$ in the top row and $t_m = 0.5$ in the bottom row. When t_m is smaller, the loss values relax faster, and the simulation terminates around iteration 600(bottom left panel). Initially, one might attribute this behavior to the complexity of the model as the number of layers in the bottom row is 12 as opposed to the four hidden layers in the top panel. However, this is not the case, as our simulation with 12 hidden layers for the boundary condition of the top panel did not reach loss values below 10^{-3}. In fact, having smaller t_m and the complexity of the model combined results in obtaining better loss functions. Nevertheless, the discrepancies between the exact solutions and the PINNs simulations still vary in the range $\pm t_m/2$ in both cases; see further comparisons in `heat_eq_2d.ipynb` in directory `src_nna.pinns.heat_eq`. This again confirms that

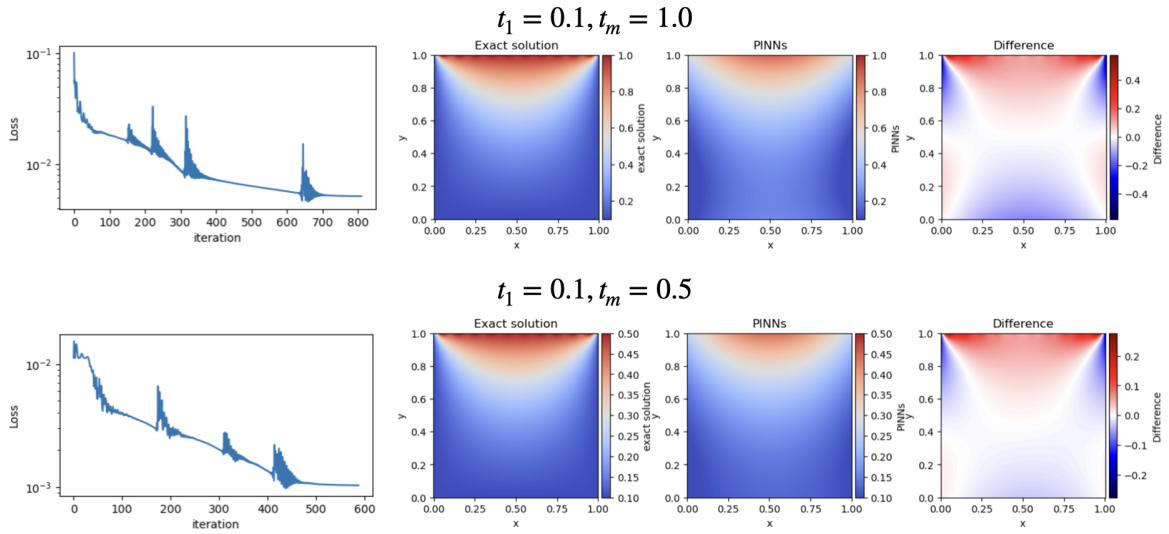

Figure 8. Solving Eq. (8) using PINNs by imposing the first type of boundary conditions inSec. II B 2 a. The model has four hidden layers and 32 hidden units in the top row. It consists of 12 hidden layers with 32 hidden units to obtain the bottom row panels. The boundary function is $t_1 = 0.1, t_m = 1.0$(top row) and $t_1 = 0.1, t_m = 0.5$ (bottom row). The learning rate is 5×10^{-3}.

PINNs experience difficulty in approximating drastic changes in the boundary conditions.

b. Parabolic PDE.— We next examine the performance of PINNs in simulating the solutions of Eq. (14). Figure 9 exemplifies some of our results on approximating the solution of parabolic PDEs.

We first look at results at $\alpha = 0.5$ and with boundary function $\phi(x) = \cos(x)$ (top row). The loss function reduces below 10^{-3}. We observe numerous jumps in the loss function due to the implemented scheduler adjusting the learning rate. The analytical and approximate solutions agree qualitatively to a large extent. However, quantitatively, they differ close to the bottom borders close to $(x, t) = \{(0, 0), (1, 0)\}$. This is similar to what we have seen in the approximate solutions of elliptic PDEs in Sec. V B 2 a.

Reducing the value of α to 0.25 (middle row) modifies the PDE and its solutions, as seen from the middle columns in the middle row. In this case, the difference between the exact and analytical values around the bottom corners of the two-dimensional region is less pronounced; note the difference in the range of color bars in the top and middle rows. Changing the boundary function to $\phi(x) = \sin(x)$ introduces asymmetricity in the solution of PDE; see bottom row. PINNs also capture this behavior. As a result of this asymmetricity, the corner where the most inconsistency between the analytical and PINNs solutions is evident is located around $(x, t) = (1, 1)$.

We note that in all cases, the loss values become less than 10^{-3}. However, as this loss function is not merely the mean-square error (L2 loss) between the analytical and exact solutions, it is not straightforward to translate not fully converged loss values to the accuracies of function approximations. The presented results and other related experiments on exploring solutions of parabolic PDEs are stored in `parabolic_eq_1p1.ipynb` in the directory `src_nna.pinns.parabolic_pde`.

c. Hyperbolic PDE.— Let us now discuss the performance of PINNs in learning the solution of the hyperbolic PDE given in Eq. (17). Figure 10 exhibits two of our results obtained with boundary functions set to $\phi(x) = \psi(x) = 1$ (top row) and $\phi(x) = x, \psi(x) = x^2$ (bottom row). In both cases, the loss functions converge very poorly, even not reaching 10^{-2}. The analytical and approximated solutions merely agree on their maxima. We see huge discrepancies between the approximated and analytical solutions, as evident from the panels on the right column. We emphasize that further experiments with the complexity of the model and hyperparameters, such as the learning rate, did not improve the outputs. These experiments are available in `hyperbolic_pde_1p1.ipynb` in the `src_nna.pinns.hyperbolic_pde` directory.

The failure of PINNs in reproducing the analytical solution can be attributed to an extra boundary condition on the $\partial_t u(x, t)$, which introduces an extra scale in various terms of the loss function. We expect that employing a multi-scale optimizer such as *MultiAdam* [13] improves the performance of PINNs.

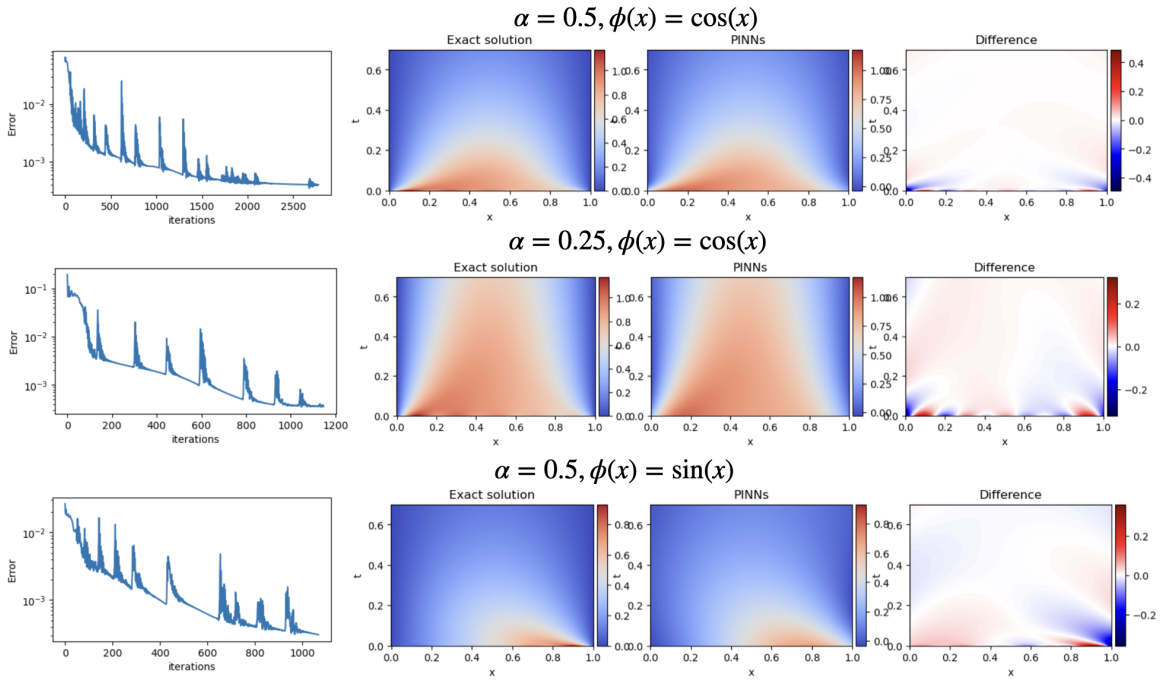

Figure 9. The solution of Eq. (14) approximated by PINNs. The left columns present loss values, the middle columns present approximate solutions from left to right, and the analytical and right columns display differences between the approximate and analytical solutions. The numbers of hidden layers and hidden units are set to four and 32, respectively. The parameter α is 0.5 in the top and bottom rows, and it is $\alpha = 0.25$ in the middle row. The boundary function is $\phi(x) = \cos(x)$ in the first two top rows, and it is set to $\phi(x) = \sin(x)$ in the last row. The learning rate is set to 5×10^{-3} in cases.

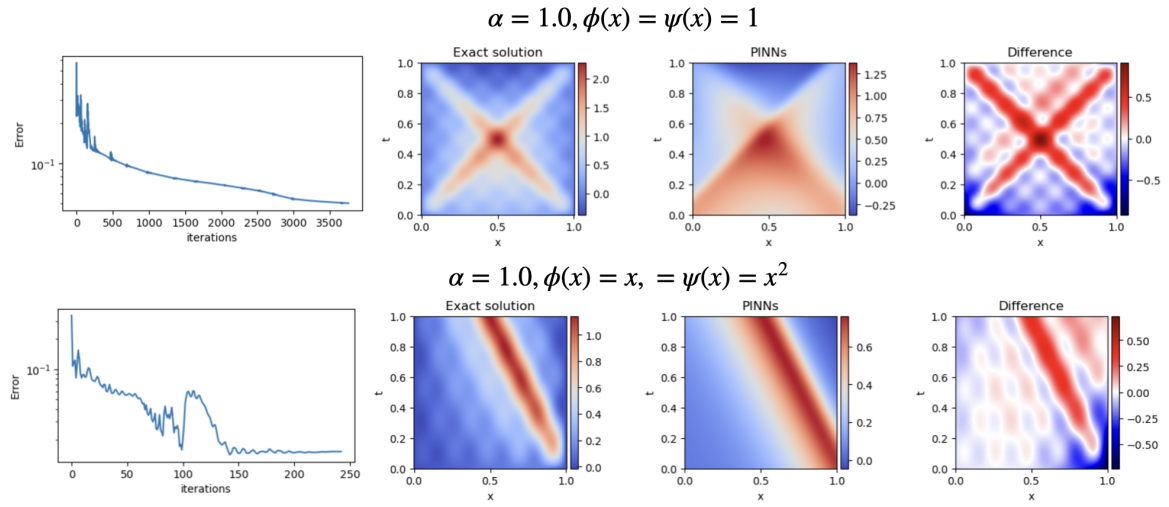

Figure 10. Solution of the hyperbolic PDEs in Eq. (17) obtained by PINNs. Boundary functions are set to $\phi(x) = \psi(x) = 1$ (top row) and $\phi(x) = x, \psi(x) = x^2$ (bottom row). The parameter $\alpha = 1$ in both rows. The number of hidden units is eight, and the number of hidden layers is four in both models. The learning rate is set to 0.05.

C. Optimal approximation

Our study compared the performance of three function approximation methods: Algorithm 1, the Imamoto-Tang algorithm, and a ReLU neural network. We evaluated these methods on three convex functions: exponential (e^x), quadratic (x^2), and cubic (x^3), using 2, 3, 5, and 10 linear segments. Figures 11, 12, and 13 illustrate the performance of each method across all functions and segment

counts.

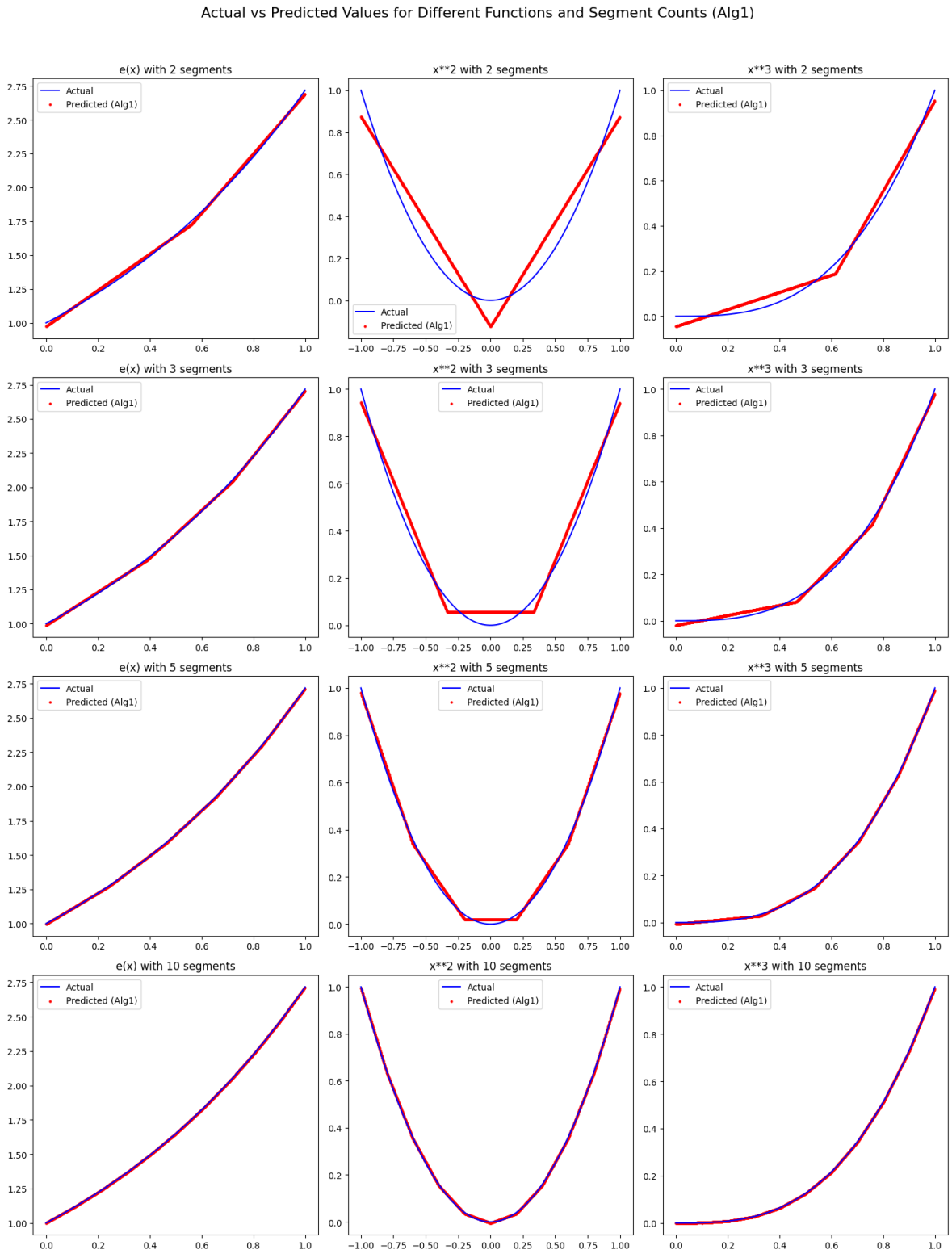

Figure 11. Actual vs Predicted Values for Different Functions and Segment Counts (Algorithm 1)

Our analysis revealed several key findings:

- **Accuracy Improvement:** Both Algorithm 1 and the Imamoto-Tang algorithm demonstrated enhanced accuracy with increasing segment counts across all functions. This improvement was

particularly notable for e^x and x^2, where approximations became visually indistinguishable from the actual functions at higher segment counts.

- **Imamoto-Tang Performance:** The Imamoto-Tang algorithm consistently outperformed Algorithm 1, achieving lower approximation errors and faster computation times, especially for higher segment counts. This advantage was most pronounced for the more complex x^3 function, where it showed superior convergence speed and accuracy.

- **ReLU Network Variability:** The ReLU neural network exhibited function-dependent performance:

  - For e^x, it closely followed the true function in the lower half of the domain for 2 and 5 segments but showed significant divergence for 3 and 10 segments, especially in the upper half.

  - With x^2, it produced generally accurate approximations, improving with increased segment counts, though small deviations were observed at domain extremes for lower segment counts.

  - Notably, for x^3, the ReLU network performed exceptionally well across all segment counts, closely approximating the true function even with only 2 segments and improving further with more segments.

- **Theoretical Bounds:** Both Algorithm 1 and the Imamoto-Tang algorithm consistently operated within the expected error margins as determined by the theoretical bounds, confirming their reliability.

- **Comparative Effectiveness:** While the Imamoto-Tang algorithm provided the most consistent and accurate approximations across all functions and segment counts, the ReLU network showed particular strength in approximating the cubic function, outperforming expectations based on previous research findings.

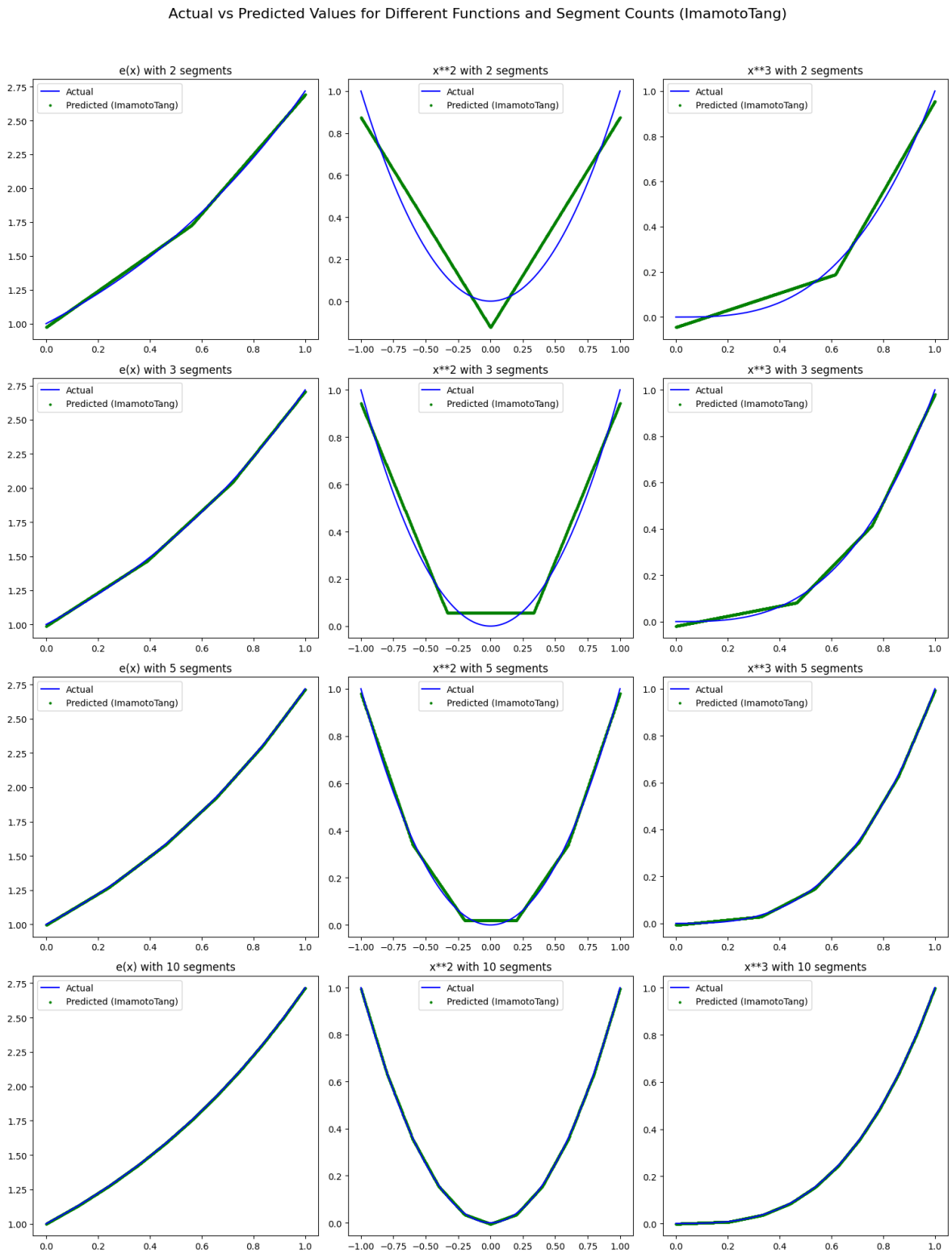

Figure 12. Actual vs Predicted Values for Different Functions and Segment Counts (Imamoto-Tang)

These results highlight the nuanced performance of each approximation method. The Imamoto-Tang algorithm emerged as the most robust approach, offering superior overall performance across different functions and segment counts. The ReLU network, while more variable in its performance, demonstrated significant potential, particularly for complex functions like x^3. Algorithm 1, though generally effective, was consistently outperformed by the Imamoto-Tang algorithm.

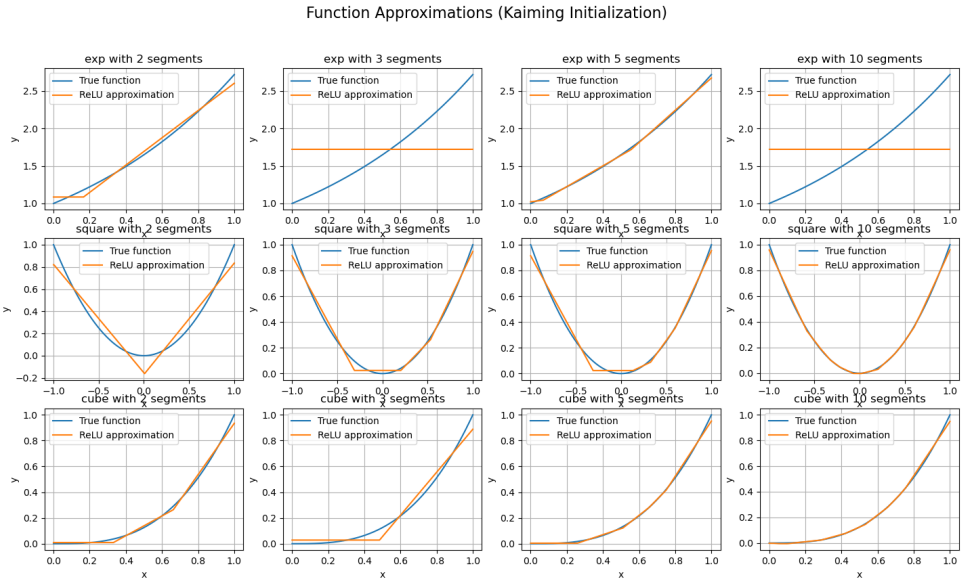

Figure 13. ReLU Network Approximation for Different Functions and Segment Counts

Our findings underscore the importance of selecting the appropriate approximation technique based on the specific function and desired accuracy. They also suggest that neural network approaches, when properly tuned, can offer competitive and sometimes superior performance for certain types of functions, challenging traditional approximation methods.

VI. DISCUSSION AND CONCLUSION

To summarize, we have presented the performance of our PYNNA package in approximating various convex and generic functions in $\mathbb{R}$ and $\mathbb{R}^2$.

Our results based on PINNs in most cases exhibits excellent performance in $\mathbb{R}$. In $\mathbb{R}^2$, the performance of PINNs deteriorates, which we associate with the presence of various scales in the total loss function of the model and employing a single-scale optimizer to optimize the landscape of loss functions. Overall, obtaining loss values smaller than 10^{-3} for functions in $\mathbb{R}^2$ is usually tricky in our PINN models.

Our study comparing Algorithm 1, the Imamoto-Tang algorithm, and a ReLU neural network for function approximation has revealed important insights: i) The Imamoto-Tang algorithm consistently outperformed Algorithm 1, demonstrating superior accuracy and efficiency across all tested functions (e^x, x^2, and x^3) and segment counts. ii) The ReLU neural network showed variable performance but excelled in approximating the cubic function, challenging traditional methods in this specific case. iii) Method selection is crucial and should be based on the target function's characteristics and the desired balance between accuracy and computational efficiency.

These findings highlight the Imamoto-Tang algorithm as a robust, all-purpose solution for function approximation. However, the unexpected strength of the ReLU network in handling complex functions like x^3 suggests potential for neural approaches in specific scenarios.

Future research could explore these methods across a broader range of functions, investigate various neural network architectures, and develop hybrid approaches combining traditional and neural network-based techniques. Extending this comparison to higher-dimensional functions could also provide valuable insights for real-world applications in optimization, control systems, and machine learning.

In summary, while traditional methods like the Imamoto-Tang algorithm offer reliable performance for convex functions, the success of neural network approaches, such as PINNs, may be comparable in various cases. This shows that the field of function approximation remains open for innovation and

further studies.

---

[1] Brandon Amos, Lei Xu, and J Zico Kolter. Input convex neural networks. In *International conference on machine learning*, pages 146–155. PMLR, 2017.

[2] Luis Balderas, Miguel Lastra, and José M Benítez. Optimizing dense feed-forward neural networks. *Neural Networks*, 171:229–241, 2024.

[3] Yoshua Bengio, Nicolas Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte. Convex neural networks. *Advances in neural information processing systems*, 18, 2005.

[4] Salah A. Faroughi, Nikhil Pawar, Celio Fernandes, Maziar Raissi, Subasish Das, Nima K. Kalantari, and Seyed Kourosh Mahjour. Physics-guided, physics-informed, and physics-encoded neural networks in scientific computing. *arXiv preprint*, 2022.

[5] Daria Fokina and Ivan Oseledets. Growing axons: Greedy learning of neural networks with application to function approximation. *Russian Journal of Numerical Analysis and Mathematical Modelling*, 38(1):1–12, 2023.

[6] Zhongkai Hao, Songming Liu, Yichi Zhang, Chengyang Ying, Yao Feng, Hang Su, and Jun Zhu. Physics-informed machine learning: A survey on problems, methods and applications. *arXiv preprint arXiv:2211.08064*, 2022.

[7] Zhongkai Hao, Jiachen Yao, Chang Su, Hang Su, Ziao Wang, Fanzhi Lu, Zeyu Xia, Yichi Zhang, Songming Liu, Lu Lu, and Jun Zhu. Pinnacle: A comprehensive benchmark of physics-informed neural networks for solving pdes. *arXiv:2306.08827*, 2023.

[8] A. Imamoto and B. Tang. Optimal piecewise linear approximation of convex functions. In *Proceedings of the World Congress on Engineering and Computer Science*, pages 1191–1194, 2008.

[9] Bo Liu and Yi Liang. Optimal function approximation with relu neural networks. *Neurocomputing*, 2021.

[10] Analytical solutions can be found in Partial Differential Equations: Analytic Solutions.

[11] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[12] Xavier Warin. The groupmax neural network approximation of convex functions. *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[13] Jiachen Yao, Chang Su, Zhongkai Hao, Songming Liu, Hang Su, and Jun Zhu. Multiadam: Parameter-wise scale-invariant optimizer for multiscale training of physics-informed neural networks. *arXiv:2306.02816*, 2023.