Journal of Computational and Applied Mathematics 490 (2027) 117947

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Reusable weights initialization framework of neural networks for function approximation

Xinwen Hu [a], Yunqing Huang [b], Nianyu Yi [a,c], Peimeng Yin [d,*]

[a] School of Mathematics and Computational Science, Xiangtan University, Xiangtan, 411105, PR China
[b] National Center for Applied Mathematics in Hunan, Key Laboratory of Intelligent Computing & Information Processing of Ministry of Education, Xiangtan University, Xiangtan, 411105, Hunan, PR China
[c] Hunan Key Laboratory for Computation and Simulation in Science and Engineering, Xiangtan University, Xiangtan, 411105, PR China
[d] Department of Mathematical Sciences, The University of Texas at El Paso, El Paso, Texas, 79968, USA

ARTICLE INFO

2020 MSC:
68T07
41A46
65D10

Keywords:
Function approximation
Weights initialization
Neural networks
Progressive training
Generalization
Least squares projection

ABSTRACT

Neural network-based function approximation plays a pivotal role in the advancement of scientific computing and machine learning. Nevertheless, training such neural models is confronted with inherent challenges: (i) each target function typically necessitates training a new model entirely from scratch, resulting in non-trivial redundant computational overhead; (ii) trained models frequently exhibit inadequate generalization capabilities beyond the training domain, thus restricting their utility for unseen target scenarios; and (iii) model performance is highly sensitive to architectural configurations and hyperparameter selections, which often demands extensive hyperparameter tuning. To address these challenges, we propose a reusable initialization framework based on basis function pretraining. Within this framework, basis neural networks are first pretrained to approximate polynomial families on a predefined reference domain. The learned parameter configurations of these pretrained basis networks are then utilized to initialize networks for approximating more complex target functions. To further enhance adaptability across arbitrary target domains, we introduce a domain mapping mechanism that transforms inputs from the target domain to the reference domain, thereby preserving structural compatibility with the pretrained models and facilitating cross-domain knowledge transfer. Extensive numerical experiments on one- and two-dimensional benchmark test cases demonstrate substantial improvements in both generalization performance and model transferability. These findings highlight the promise of initialization-based strategies in enabling highly scalable and modular neural network-based function approximation, paving the way for the development of more efficient and robust neural models in scientific computing and machine learning applications. The full code is made publicly available on Gitee.[1]

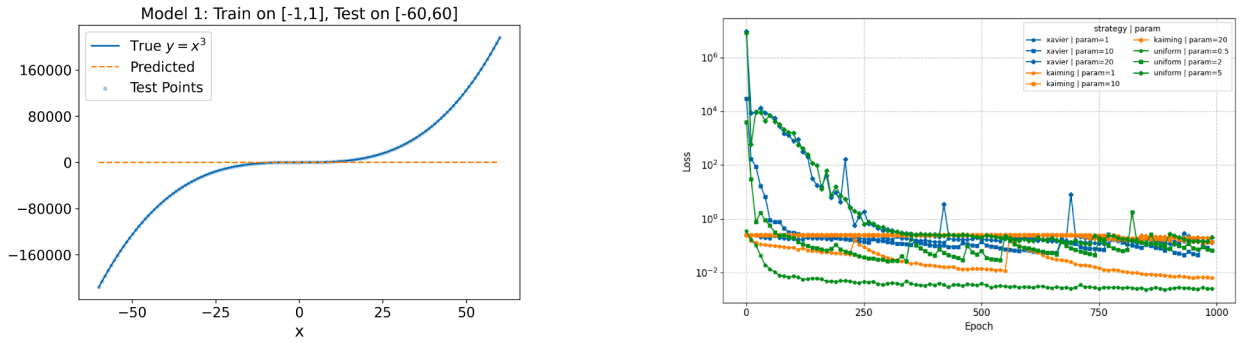
1. Introduction

Function approximation occupies a central position in computational mathematics, computer science, and engineering, underpinning theoretical advancements across these disciplines while enabling a broad spectrum of practical applications. For numerous complex systems governed by differential or integral equations, analytical solutions are often unattainable. In such scenarios, func-

* Corresponding author.
E-mail addresses: xinwen618@gmail.com (X. Hu), huangyq@xtu.edu.cn (Y. Huang), yinianyu@xtu.edu.cn (N. Yi), pyin@utep.edu (P. Yin).

[1] Gitee: https://gitee.com/AIxinwen/nn4poly.

https://doi.org/10.1016/j.cam.2026.117947
Received 24 January 2026; Received in revised form 29 June 2026
Available online 14 July 2026



(a) extrapolation performance (b) training loss

Fig. 1. The performance of extrapolation of neural network approximation and the training loss with difference weights initialization.

tion approximation offers a tractable pathway: by constructing numerical surrogates, it facilitates efficient computation and in-depth analysis of these otherwise intractable systems. These approximation techniques serve as a foundational pillar across diverse fields, including signal processing, numerical analysis [1], control theory, and uncertainty quantification (UQ)–a critical area in modern scientific computing [2].

With the rapid advancement of machine learning, function approximation has evolved into a critical component of data-driven modeling. Modern approaches, especially those based on neural networks and support vector machines, have enabled remarkable advancements in fields including computer vision, speech recognition, and natural language processing [3]. Among these, feedforward neural networks (FNNs) have attracted particular attention due to their universal approximation property: any continuous function defined on a compact domain can be approximated arbitrarily well by an FNN with sufficiently wide hidden layers [4,5].

Despite their strong expressive power, the practical use of neural networks for function approximation confronted with inherent challenges that impede their widespread adoption. First, networks employing classical activation functions (e.g., Sigmoid, Tanh) often suffer from vanishing or exploding gradients during training, particularly in deep architectures [6]. Second, the optimization landscape of neural networks is inherently non-convex, characterized by numerous local minima and saddle points that impede convergence dynamics and undermine training stability [7]. Third, neural networks exhibit inherent susceptibility to overfitting, especially when training data are limited or noisy, which may lead to poor generalization performance [8]. Finally, the approximation accuracy is highly sensitive to the sampling strategy used to generate training data. For instance, uniform sampling may fail to capture intricate function behavior in sparsely sampled regions, whereas adaptive or hybrid sampling strategies can provide more effective coverage of critical features.

To illustrate some main challenges, we introduce the mean squared error (MSE), the coefficient of determination (commonly denoted by R^2), and the relative L^2 error as quantitative evaluation metrics. The MSE is defined as

$$\mathrm{MSE} = \frac{1}{n}\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2,$$

the coefficient of determination (R^2) as

$$R^2 = 1 - -\frac{\|\mathbf{y} - -\hat{\mathbf{y}}\|_2^2}{\|\mathbf{y} - -\bar{y}\mathbf{1}\|_2^2},$$

and the relative L^2 error as

$$\mathrm{ReL}^2 = \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|_2}{\|\mathbf{y}\|_2},$$

where $\mathbf{y} = (y_1, y_2, \ldots, y_n)^\top$ denotes the observed values, $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n)^\top$ the predicted values, $\bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$ the sample mean, and $\mathbf{1} \in \mathbb{R}^n$ the vector of all ones. Here, R^2 is a statistical measure that quantifies the proportion of the total variance in $\mathbf{y}$ that is explained by the model's predictions $\hat{\mathbf{y}}$. It reflects how well the model captures the variability in the data, with $R^2 \in [0, 1]$ and $R^2 = 1$ indicating a perfect fit. The relative L^2 error provides a scale-invariant measure of approximation accuracy, making it particularly suitable for comparing performance across functions with different magnitudes.

To further emphasize the third challenge–generalization, we consider a standard feedforward neural network with architecture $[1, 512, 1]$ to approximate the polynomial function $f(x) = x^3$ on the interval $[-1, 1]$. When the trained model is evaluated on the much larger interval $[-60, 60]$, its extrapolation performance deteriorates dramatically, as illustrated in Fig. 1(a). Quantitative results on $[-60, 60]$ further confirm the poor generalization capability of the network, yielding only $R^2 = 2.44 \times 10^{-3} \ll 1$ and $\mathrm{MSE} = 6.96 \times 10^9$. Both metrics of an extremely small $R^2 \ll 1$ and an extremely large MSE clearly imply that the model fails to capture the underlying functional behavior outside the training domain. This example highlights a fundamental limitation of neural network-based approximation: the lack of reliable generalization beyond the region where training data are available.

To illustrate the last challenge–the impact of different initialization strategies, we trained a fully connected neural network with architecture $[2, 64, 64, 64, 1]$ to approximate the two dimensional function $f(x, y) = \sin(\pi x)\sin(4\pi y)$. Three distinct weight initialization

Table 1
Approximation errors for $f(x, y) = \sin(\pi x)\sin(4\pi y)$ under different initialization strategies.

Strategy	param	ReL^2	MSE
Xavier	1	7.27×10^{-1}	1.31×10^{-1}
Xavier	10	4.81×10^{-1}	5.76×10^{-2}
Xavier	20	7.01×10^{-1}	1.22×10^{-1}
Kaiming	1	1.55×10^{-1}	5.97×10^{-3}
Kaiming	10	7.53×10^{-1}	1.41×10^{-1}
Kaiming	20	8.90×10^{-1}	1.97×10^{-1}
uniform	0.5	1.32×10^{-1}	4.36×10^{-3}
uniform	2	4.84×10^{-1}	5.83×10^{-2}
uniform	5	8.03×10^{-1}	1.60×10^{-1}

strategies were considered: Xavier, Kaiming, and uniform initialization. For each strategy, we evaluated the network's performance across a range of values of a tunable hyperparameter. As shown in Table 1 and Fig. 1(b), the choice of initialization strategy has a significant impact on both the convergence of the network and the final approximation error. In Table 1, the column labeled "param" refers to the strategy-specific hyperparameter passed during initialization: for Xavier initialization, it corresponds to the gain factor; for Kaiming initialization, it specifies the negative slope of the Leaky ReLU nonlinearity; and for uniform initialization, it defines the bound of the uniform distribution. Substantial differences are observed across initialization methods. Moreover, even within the same initialization method, performance varies markedly with the choice of the parameter. For example, Kaiming initialization results in ReL^2 errors of 1.58×10^{-1}, 7.53×10^{-1}, and 8.90×10^{-1} for param of 1, 10, and 20, respectively, highlighting a strong sensitivity to this setting. These results indicate that initialization is not merely an implementation detail but a critical factor affecting both numerical stability and approximation accuracy.

To tackle the two main challenges outlined above–poor generalization and sensitivity to initialization, we introduce a modular neural approximation framework that integrates parameter reuse, domain normalization, and basis function pretraining. The central idea is to pretrain a collection of basis neural networks to approximate multivariate polynomial basis functions on a reference domain in $\mathbb{R}^d$, such as the hypercube $[-1, 1]^d$. These pretrained modules then serve as reusable and composable components, which can be either directly assembled or fine-tuned to approximate more complex target functions defined over multidimensional input spaces.

To achieve generalization to arbitrary domains $\Omega \subset \mathbb{R}^d$, we integrate a domain normalization module that systematically transforms inputs from Ω to the canonical reference hypercube $[-1, 1]^d$ while preserving the intrinsic structural properties of the target function. This transformation facilitates the seamless integration of pretrained basis networks into downstream approximation tasks, thereby enhancing model transferability and enabling more reliable extrapolation beyond the training distribution.

We systematically evaluate the proposed method on a comprehensive suite of benchmark functions in both one-dimensional and two-dimensional settings. Additionally, we analyze the influence of key architectural factors–including network depth, width, and activation function selection–on approximation accuracy and generalization performance. Experimental results demonstrate significant improvements in both approximation precision and cross-domain transferability, highlighting the method's potential as a scalable, general-purpose neural approximation approach.

The proposed method boasts several distinct key advantages:

- Enhanced generalization capability: By systematically mapping inputs to a canonical reference domain, the method effectively mitigates extrapolation biases and bolsters robustness against out-of-distribution samples, addressing a core limitation of conventional neural approximation approaches.
- Efficiency via parameter reuse: Basis networks are pretrained once and leveraged across multiple downstream approximation tasks. This parameter reuse paradigm eliminates redundant training overhead and substantially accelerates convergence in subsequent task-specific optimization processes.
- Strong modularity and extensibility: The framework inherently supports plug-and-play composition of pretrained basis modules. These modules can either be frozen as fixed feature extractors to preserve learned structural priors or fine-tuned to enhance model expressivity for complex target functions. Furthermore, the combination coefficients of these basis modules offer dual flexibility: they can be computed analytically (e.g., via the least squares method) for interpretable linear composition or learned in an end-to-end manner to capture nonlinear task characteristics.

The rest of this paper is organized as follows. Section 2 introduces the adopted function approximation method and model architecture, and explains how data mapping improves generalization, together with the strategy of pretraining and weights reuse for efficient network construction. Section 3 explains the influence of network depth, neuron count, and activation functions on the expressive power of basis functions, and presents inference results for one- and two-dimensional domains. Section 4 explains numerical experiments, applying the proposed initialization strategy to representative one- and two-dimensional functions with corresponding illustrations and error metrics. Section 5 concludes the paper and outlines future research directions.

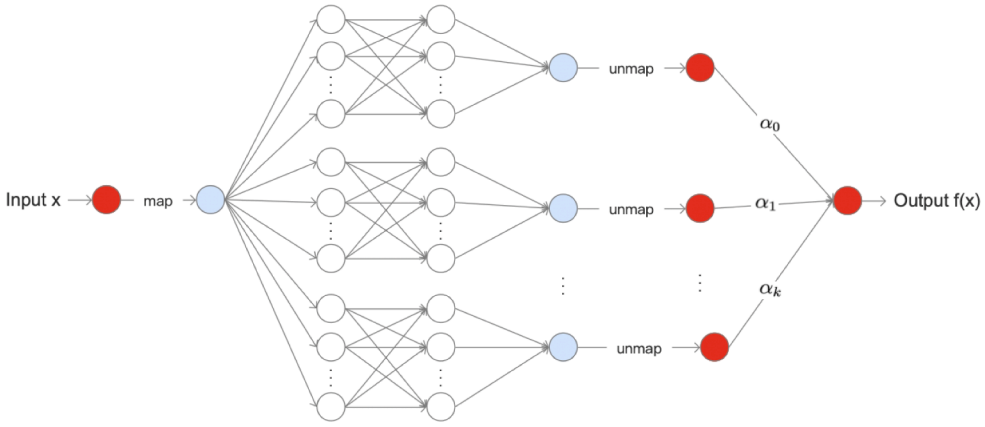


Fig. 2. Workflow of the reusable weights initialization framework for function approximation.

2. Reusable weights initialization for efficient function approximation

We denote the space of continuous functions on a compact domain $\Omega \subset \mathbb{R}^d$ by $C(\Omega)$. Let $f \in C(\Omega)$ be a continuous target function defined over a d-dimensional domain $\Omega = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$. Here, we generalize the problem from one-dimensional space to arbitrary dimension $d \geq 1$; the one-dimensional case corresponds to $d = 1$. We aim to construct an approximation $\mathcal{P}f$ using a set of pretrained basis functions $\hat{\varphi}_{\mathbf{k}}$ by

$$\mathcal{P}f(\mathbf{x}) = \sum_{\mathbf{k} \in \mathcal{K}} \alpha_{\mathbf{k}} \varphi_{\mathbf{k}}(\mathbf{x}), \tag{2.1}$$

where:

- $\mathcal{K} \subset \mathbb{N}_0^d$ is a finite multi-index set specifying the collection of basis functions used in the least-squares procedure (e.g., total degree or hyperbolic cross truncation);
- $\mathcal{P}$ represents a projection operator induced by the multivariate basis functions;
- $\varphi_{\mathbf{k}}(\mathbf{x}) = \hat{\varphi}_{\mathbf{k}}(\mathcal{T}^{-1}(\mathbf{x}))$, where $\mathcal{T} : [-1, 1]^d \to \Omega$ is a component-wise domain transformation mapping the reference hypercube $[-1, 1]^d$ to Ω;
- The coefficients $\alpha_{\mathbf{k}}$ may be obtained either through a multivariate least-squares fit or by fine-tuning them as trainable parameters in a neural network framework.

Instead of training a new neural network from scratch for each target function, an approach that often suffers from slow convergence and limited generalization, we propose a reusable weight initialization strategy based on pretrained monomial basis networks. Specifically, we first construct a library of compact neural networks, each trained from scratch to approximate a single monomial basis function on a fixed reference domain. These pretrained monomial modules are then repurposed to initialize the hidden layers of a general-purpose function approximator. The approximator's architecture is assembled by combining the frozen or fine-tuned internal representations of these basis networks, while the final linear combination layer, whose coefficients determine how the basis functions are blended, can either be computed analytically or learned end-to-end during joint training.

This initialization scheme ensures that the approximator starts from a structured, functionally meaningful prior, leading to faster convergence, reduced overfitting, and improved generalization, especially over large or varying input domains. The overall workflow is illustrated in Fig. 2, consisting of two conceptual phases:

- Construction of a library of pre-trained basis neural networks on a reference domain;
- Deployment of these networks' weights to initialize the hidden layers of the target approximation network.

In this schematic, blue and white neurons together form a neural network trained on a reference domain to approximate canonical basis functions; the learned hidden-layer weights are retained as reusable features. The transformations between blue and red neurons represent input mapping and output unmapping operations that adapt the network to physical domains. These domain-aware mappings, combined with the pretrained internal representation, enable enhanced generalization and structural flexibility when approximating target functions. To complement the schematic workflow illustrated in Fig. 2, Algorithm 1 formalizes the procedural details of the framework.

2.1. Pretraining basis neural networks

To simplify the presentation, we fix the reference domain as $[-1, 1]^d$. To approximate functions within this domain, we construct a set of basis neural networks that serve as accurate surrogates for the polynomial basis functions $\{\varphi_{\mathbf{k}}\}_{\mathbf{k} \in \mathcal{K}_{\text{lib}}}$. Specifically, we define

Algorithm 1 Reusable weights initialization framework in multi-dimensional space.

Require: Target function $f(\mathbf{x})$, target domain $\Omega = [a_1, b_1] \times \cdots \times [a_d, b_d]$, reference domain $[-1, 1]^d$, pretrained multivariate basis networks $\{\hat{\varphi}_{\mathbf{k}}\}_{\mathbf{k}\in\mathcal{K}_{\text{lib}}}$, domain transformation $\mathcal{T} : [-1, 1]^d \to \Omega$

Ensure: Approximated function $\mathcal{P}f(\mathbf{x})$

Offline: Library Construction

1: **for** each multi-index $\mathbf{k} \in \mathcal{K}_{\text{lib}}$ **do**

2: Train $\hat{\varphi}_{\mathbf{k}}$ on the reference domain $[-1, 1]^d$ to approximate the $\mathbf{k}$th multivariate basis function

3: Store trained parameters $\theta_{\mathbf{k}}$ to formulate the reusable library V_{NN}

4: **end for**

Online: Target Approximation

5: Map input $\mathbf{x} \in \Omega$ to reference domain: $\hat{\mathbf{x}} = \mathcal{T}^{-1}(\mathbf{x})$

6: Evaluate each pretrained basis network at $\hat{\mathbf{x}}$ to obtain $\hat{\varphi}_{\mathbf{k}}(\hat{\mathbf{x}})$

7: Construct the approximator using the selected subset $\mathcal{K} \subseteq \mathcal{K}_{\text{lib}}$:

$$\mathcal{P}f(\mathbf{x}) = \sum_{\mathbf{k}\in\mathcal{K}} \alpha_{\mathbf{k}}\, \hat{\varphi}_{\mathbf{k}}(\mathcal{T}^{-1}(\mathbf{x}))$$

8: Determine coefficients $\{\alpha_{\mathbf{k}}\}_{\mathbf{k}\in\mathcal{K}}$ either by solving a multivariate least-squares problem using samples $\{(\mathbf{x}^{(i)}, f(\mathbf{x}^{(i)}))\}_{i=1}^{N}$, or by fine-tuning them as trainable weights in a neural network with fixed basis functions

9: **return** $\mathcal{P}f(x)$

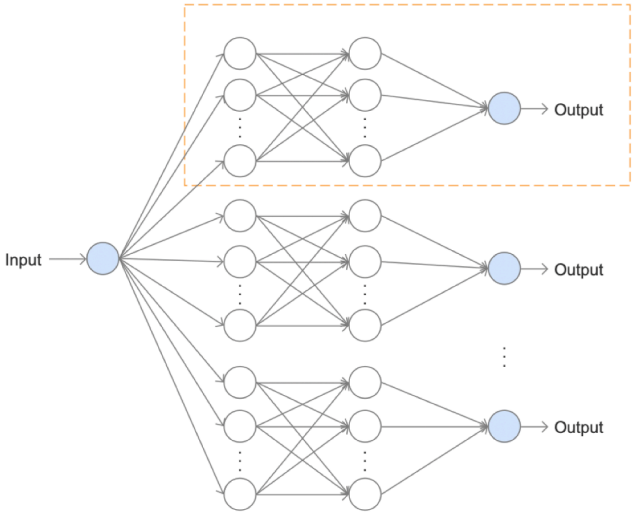


Fig. 3. Modular construction of basis neural networks.

the neural representation space

$$V_{\text{NN}} = \left\{\hat{\varphi}_{\mathbf{k}}(\hat{\mathbf{x}}; \theta_{\mathbf{k}}) \,\middle|\, \mathbf{k} \in \mathcal{K}_{\text{lib}}\right\},$$

where $\mathcal{K}_{\text{lib}} \subset \mathbb{N}_0^d$ is a finite multi-index set, and each $\hat{\varphi}_{\mathbf{k}} \approx \varphi_{\mathbf{k}}$ is a neural network approximation of the multivariate polynomial basis function, parameterized by trainable weights $\theta_{\mathbf{k}}$. The basis networks are trained by minimizing the empirical loss

$$L(\theta_{\mathbf{k}}) = \frac{1}{N}\sum_{i=1}^{N} \left|\hat{\varphi}_{\mathbf{k}}(\hat{\mathbf{x}}_i; \theta_{\mathbf{k}}) - \varphi_{\mathbf{k}}(\hat{\mathbf{x}}_i)\right|^2,$$

where $\{\hat{\mathbf{x}}_i\}_{i=1}^{N}$ are sampling points in the reference domain $[-1, 1]^d$.

Once trained, the resulting basis neural networks form a reusable library of basis function modules. As illustrated in Fig. 3, the boxed orange module denotes a neural approximation to a specific basis function φ_k. These modules can be reused and made up for downstream tasks. The weights of these modules may either be fixed to preserve their pre-learned behavior or fine-tuned to accommodate task-specific adaptations.

To efficiently approximate structured families of multivariate basis functions–such as tensor-product polynomials $\varphi_{\mathbf{k}}(\mathbf{x}) = \prod_{j=1}^{d} x_j^{k_j}$ with multi-index $\mathbf{k} = (k_1, \ldots, k_d) \in \mathbb{N}_0^d$, we employ a progressive initialization strategy. Rather than training each neural surrogate independently from scratch, the parameters obtained from a previously trained model are reused to initialize the training of a higher-complexity basis function. This sequential transfer of knowledge exploits the inductive bias between related basis functions, yielding

substantial improvements in training efficiency and stability. Similar inductive biases have been validated in studies on transfer learning [9,10]. The procedure is outlined in Algorithm 2.

Algorithm 2 Progressive initialization in multi-dimensional space.

Require: Multivariate basis family $\{\varphi_{\mathbf{k}}(\mathbf{x})\}_{\mathbf{k}\in\mathcal{K}_{\text{lib}}}$, where $\mathcal{K}_{\text{lib}} \subset \mathbb{N}_0^d$ is ordered by increasing total degree $|\mathbf{k}| = k_1 + \cdots + k_d$
Ensure: Trained models $\{\hat{\varphi}_{\mathbf{k}}\}_{\mathbf{k}\in\mathcal{K}_{\text{lib}}}$ with parameters $\{\theta_{\mathbf{k}}\}_{\mathbf{k}\in\mathcal{K}_{\text{lib}}}$
1: Let $\mathbf{k}^{(0)}$ be the multi-index of minimal degree (e.g., $\mathbf{0}$)
2: Initialize $\theta^{(0)}_{\mathbf{k}^{(0)}}$ randomly
3: Train model $\hat{\varphi}_{\mathbf{k}^{(0)}}(\hat{\mathbf{x}};\theta_{\mathbf{k}^{(0)}}) \approx \varphi_{\mathbf{k}^{(0)}}(\hat{\mathbf{x}})$ using $\theta^{(0)}_{\mathbf{k}^{(0)}}$
4: **for** each subsequent $\mathbf{k} \in \mathcal{K}_{\text{lib}} \setminus \{\mathbf{k}^{(0)}\}$ in order of non-decreasing $|\mathbf{k}|$ **do**
5: Select a predecessor $\mathbf{k}'$ with $|\mathbf{k}'| = |\mathbf{k}| - 1$ (e.g., by decrementing one component of $\mathbf{k}$)
6: $\theta^{(0)}_{\mathbf{k}} \leftarrow \theta_{\mathbf{k}'}$
7: Initialize model $\hat{\varphi}_{\mathbf{k}}$ with $\theta^{(0)}_{\mathbf{k}}$
8: Train $\hat{\varphi}_{\mathbf{k}}(\hat{\mathbf{x}};\theta_{\mathbf{k}}) \approx \varphi_{\mathbf{k}}(\hat{\mathbf{x}})$
9: **end for**
10: **return** $\{\theta_{\mathbf{k}}\}_{\mathbf{k}\in\mathcal{K}_{\text{lib}}}$

2.2. Constructing approximation via basis neural networks and data mapping

To extend the applicability of pretrained basis neural networks–originally trained on the reference domain $[-1,1]^d$ with $d \in \{1,2,3\}$–to arbitrary computational domains in $\mathbb{R}^d$, we generalize the data mapping strategy to a vectorized formulation that consistently preserves learned representations across dimensions. This approach enables seamless reuse of networks trained on canonical monomial bases for general input intervals or spatial regions.

We begin by defining a component-wise bidirectional mapping between the physical domain point $\mathbf{x} = (x_1, x_2 \dots, x_d)^T \in \mathbb{R}^d$ and its normalized counterpart $\hat{\mathbf{x}} \in [-1,1]^d$:

$$\begin{cases} \hat{\mathbf{x}} = T(\mathbf{x}) & \text{(Forward mapping)}, \\ \mathbf{x} = T^{-1}(\hat{\mathbf{x}}) & \text{(Inverse mapping)}. \end{cases}$$

Forward mapping is performed through logarithmic scaling in dimensions. For each component x_i, we compute an integer scale exponent

$$s_i = \left\lceil \log_{10}\left(\lfloor |x_i| + 1 \rfloor\right) \right\rceil, \quad i = 1, \dots, d,$$

and define the scaling vector $\sigma = (10^{s_1}, \dots, 10^{s_d})^T$. The normalized input is then given by element-wise division:

$$\mathbf{x} = T(\mathbf{x}) = \mathbf{x} \oslash \sigma = \left(\frac{x_1}{10^{s_1}}, \dots, \frac{x_d}{10^{s_d}}\right)^T,$$

where the operator $\oslash$ denotes Hadamard (component-wise) division. Critically, exponents $\{s_i\}_{i=1}^d$ are computed once per input vector during the forward pass and remain fixed throughout the inference.

To illustrate the improved generalization ability of neural networks with this mapping, we revisit the approximation of the function $y = x^3$ with structure $[1, 512, 1]$ over $x \in [-60, 60]$, compared to Fig. 1(a). After applying the transformation $x \mapsto \hat{x}$, a neural network originally trained on $[-1, 1]$ is repurposed to approximate the transformed function $\hat{y} = \hat{x}^3$. The inference results on $x \in [-60, 60]$ are presented in Fig. 4, where the inverse mapping restores predictions in the original domain. Quantitative evaluation yields MSE = 2.87×10^1 and $R^2 = 0.999812$.

Assume the neural network has been pretrained on the reference domain to approximate the canonical monomial basis functions of multi-index degree $\mathbf{k} = (k_1, \dots, k_d)^T$:

$$\hat{\varphi}(\hat{\mathbf{x}}) = \prod_{i=1}^{d} \hat{x}_i^{k_i}, \tag{2.2}$$

Given this representation, the corresponding physical-space basis function $\varphi(\mathbf{x}) = \prod_{i=1}^d x_i^{k_i}$ is recovered via inverse scaling. Substituting $x_i = 10^{s_i}\hat{x}_i$ yields

$$\varphi(\mathbf{x}) = \prod_{i=1}^{d} (10^{s_i}\hat{x}_i)^{k_i} = \left(\prod_{i=1}^{d} 10^{k_i s_i}\right) \cdot \left(\prod_{i=1}^{d} \hat{x}_i^{k_i}\right) = 10^{\mathbf{k}^\top \mathbf{s}} \cdot \hat{\varphi}(\hat{\mathbf{x}}), \tag{2.3}$$

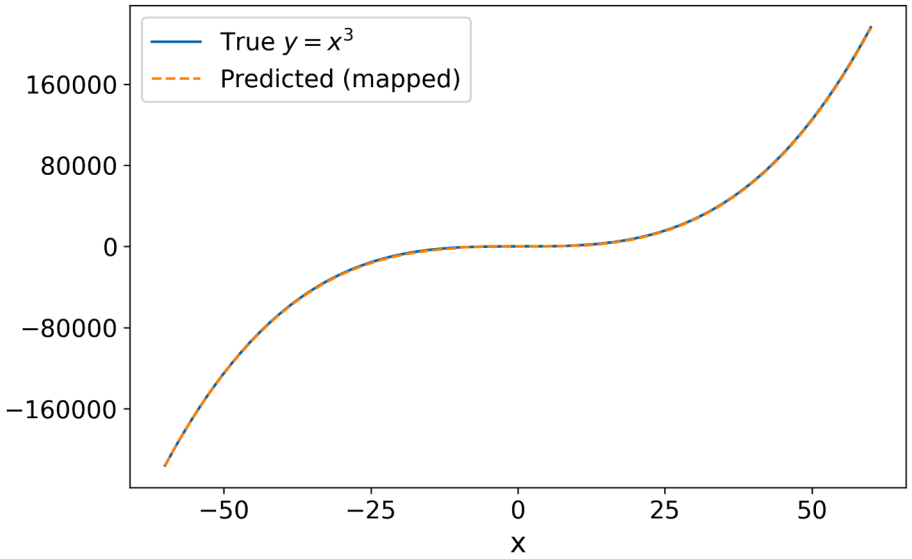


Fig. 4. Model inference of $y = x^3$ over $[-60, 60]$ after inverse mapping.

where $\mathbf{s} = (s_1, \ldots, s_d)^\top$ encodes the original scaling exponents derived from $\mathbf{x}$.

Eqs. (2.2) and (2.3) constitute a dimension-agnostic framework: they apply uniformly to one-, two-, and three-dimensional settings and naturally extend to any linear combination of polynomial basis functions by superposition. The proposed vectorized mapping thus enables a consistent deployment of pretrained networks in the reference-domain across arbitrary spatial scales and domains in $\mathbb{R}^d$, while preserving exact recovery of the polynomial structure under the prescribed scaling law.

This weights-reuse framework enables neural models trained on the reference domain to be seamlessly extended to arbitrary real-valued intervals without retraining from scratch. In downstream applications–such as functional regression or spectral projection–the reused basis networks provide efficient parameter initialization and retain generalization capability.

3. Optimal basis neural networks

In function approximation problems, polynomials are commonly employed as basis functions. However, different polynomial bases exhibit varying levels of complexity and nonlinearity. In practical applications, hardware constraints and computational resources further necessitate careful selection of the neural network architecture.

The problem can be reformulated as follows: Given GPU/CPU resource constraints and training time limits, our goal is to optimize key hyperparameters of neural networks, hyperparameters–such as depth, width, and activation functions–to balance precision with efficient inference [11,12]. Specifically, to maximize approximation accuracy under limited computational resources, we investigate the following architectural hyperparameters:

- Network depth (number of hidden layers): The number of layers in a neural network determines the model's depth. Excessively deep networks may lead to prolonged training times and an increased risk of overfitting, yet they are often necessary when approximating complex functions. Common choices for the number of layers typically include $1, 2, 4$, or 8 layers.
- Width (neurons per layer): The number of neurons per layer affects the network's representational capacity. More neurons enable the approximation of more complex functions, but they also increase computational and storage costs. Typically, the number of neurons per layer ranges from 16 to 1024.
- Activation functions: Choosing appropriate activation functions can enhance the network's nonlinear representation capability while mitigating the risks of gradient vanishing or explosion. Common activation functions available for selection include ReLU, Tanh, GELU, Mish, and ELU.

The search space for architecture selection depends on target function complexity, dataset size and available hardware resources.

3.1. Network architecture

Without loss of generality, we take the function $y = x^3$ as a representative example to examine how the number of neurons and hidden layers influences the approximation performance. In all experiments, we employ the Mish activation function, which, while not necessarily optimal in terms of raw performance, provides consistently high accuracy and helps minimize confounding effects from architectural variations.

- **Training Data**: Uniformly sampled from the interval $[-1, 1]$.

Table 2
Approximation $y = x^3$ using single hidden layer neural networks with progressively increasing neurons.

Net	MSE	R^2
[1, 8, 1]	8.57×10^{-7}	0.999989
[1, 16, 1]	5.84×10^{-8}	0.999999
[1, 32, 1]	4.23×10^{-8}	0.999999
[1, 64, 1]	1.15×10^{-7}	0.999999
[1, 128, 1]	8.08×10^{-8}	0.999999
[1, 256, 1]	8.76×10^{-9}	1.000000

Table 3
Neural networks with increasing hidden layers (fixed 8 neurons per layer) for approximating $y = x^3$.

Net	MSE	R^2
[1, 8, 1]	8.57×10^{-7}	0.999989
[1, 8, 8, 1]	5.39×10^{-8}	0.999999
[1, 8, 8, 8, 8, 1]	4.48×10^{-8}	0.999999

Table 4
Approximating $y = x^3$ using neural networks with increasing hidden layers and fixed width (128 neurons per hidden layer).

Net	MSE	R^2
[1, 128, 1]	8.08×10^{-8}	0.999999
[1, 128, 128, 1]	1.50×10^{-8}	1.000000
[1, 128, 128, 128, 128, 1]	3.16×10^{-7}	0.999996

- **Test Data**: Uniformly sampled from the broader interval $[-20, 20]$ to evaluate out-of-distribution generalization.

We first consider the single hidden-layer neural network, and the corresponding results are reported in Table 2. we can see clearly: i) as the number of neurons increases ($8 \rightarrow 32$), the MSE decreases from 8.57×10^{-7} to 4.23×10^{-8}, demonstrating that expanding model capacity enhances fitting accuracy on training data; ii) with 64 neurons, the MSE exhibits minor fluctuations and the smallest MSE is achieved with 256 neurons.

Next, we examine the impact of network depth on function approximation performance. As shown in Table 3, when fixing the width at 8 neurons per hidden layer, increasing the number of layers progressively enhances the network's expressive power via nonlinear compositional operations, thereby compensating for the limited capacity of single-layer architectures. In contrast, for wider networks with 128 neurons per hidden layer (Table 4), the best performance is achieved with two hidden layers, while further increasing depth to four layers leads to performance degradation. This underscores the necessity of balanced width–depth scaling, in line with the compound scaling principle of EfficientNet [13].

For a comprehensive evaluation, we systematically varied both the number of neurons and the depth of the network while keeping the data sampling protocol fixed: training samples were drawn from $[-1, 1]$, whereas the test set was extended to $[-20, 20]$ to rigorously assess generalization beyond the training distribution. This experimental design enables a controlled numerical investigation of the interplay between network width and depth.

The results in Table 5 reveal marked performance differences across network architectures. A single hidden layer network with configuration $[1, 8, 1]$ attains an MSE of 8.57×10^{-7}, substantially higher than that of deeper models, highlighting the critical role of depth in enhancing representational capacity. Expanding to a two-hidden-layer architecture $[1, 8, 8, 1]$ lowers the MSE to 5.39×10^{-8}, showing that additional depth–even without increasing width–can significantly strengthen nonlinear approximation capabilities.

Performance continues to improve with progressively wider architectures: the $[1, 8, 16, 32, 64, 1]$ network attains an MSE of 6.37×10^{-9}, and the $[1, 8, 64, 128, 256, 1]$ variant further reduces it to 2.1×10^{-9}. These results demonstrate that gradually increasing the number of neurons across layers effectively expands model capacity while maintaining training stability. Such architectures enable hierarchical feature learning, where lower layers capture low-frequency patterns and higher layers extract fine-grained details, consistent with established hierarchical representation theory [14]. The consistent improvements across these systematically scaled architectures provide strong empirical evidence for the joint importance of depth and progressive width scaling in neural network design.

In terms of network width, increasing the number of neurons per layer enhances representational power and reduces fitting error, particularly in shallow networks. For instance, widening a single hidden layer from 8 to 256 neurons significantly reduces the MSE. Notably, good performance is also achieved by architectures that combine depth with progressively increasing layer widths, such

Table 5
Neural networks with varied hidden layers and node counts for approximating $y = x^3$.

Net	MSE	R^2
[1, 8,1]	8.57×10^{-7}	0.999989
[1, 8, 8, 1]	5.39×10^{-8}	0.999999
[1, 8, 16, 1]	3.34×10^{-7}	0.999996
[1, 8, 32, 1]	1.36×10^{-7}	0.999998
[1, 8, 64, 1]	1.81×10^{-7}	0.999998
[1, 8, 128, 1]	7.26×10^{-7}	0.999991
[1, 8, 256, 1]	9.63×10^{-9}	1.000000
[1, 8, 8, 8, 8, 1]	4.48×10^{-8}	0.999999
[1, 8, 16, 32, 64, 1]	6.37×10^{-9}	1.000000
[1, 8, 16, 32, 128, 1]	1.30×10^{-8}	1.000000
[1, 8, 16, 32, 256, 1]	8.74×10^{-9}	1.000000
[1, 8, 16, 64, 128, 1]	3.90×10^{-8}	1.000000
[1, 8, 16, 64, 256, 1]	5.06×10^{-9}	1.000000
[1, 8, 16, 128, 256, 1]	1.74×10^{-8}	1.000000
[1, 8, 32, 64, 128, 1]	2.94×10^{-9}	1.000000
[1, 8, 32, 64, 256, 1]	2.98×10^{-8}	1.000000
[1, 8, 32, 128, 256, 1]	3.80×10^{-8}	1.000000
[1, 8, 64, 128, 256, 1]	2.10×10^{-9}	1.000000

as $[1, 8, 16, 32, 64, 1]$ or $[1, 8, 64, 128, 256, 1]$. These structures not only achieve extremely low training error but also exhibit strong generalization.

Based on these experimental findings, and considering the dual requirements of accuracy and computational efficiency for basis function approximation within the training domain, all subsequent experiments employ a standardized architecture with a single hidden layer containing 1024 neurons. This configuration provides sufficient expressive power to capture the complexity of polynomial functions while remaining computationally tractable across different basis functions and training settings. In addition, the domain mapping module introduced in this work ensures robust out-of-domain generalization of the learned basis functions.

3.2. Activation functions

The activation function plays a pivotal role in both the design and training of neural networks. These nonlinear transformations not only determine a model's expressive power but also directly impact computational efficiency during training, particularly when processing large-scale datasets or complex architectures. With the advancement of deep learning, an expanding repertoire of activation functions has emerged, each demonstrating distinct effectiveness and efficiency across varied tasks.

Under controlled experimental conditions (macOS 15.3.1, Python 3.11.9), systematic measurements of forward and backward propagation times were conducted over N iterations to evaluate computational efficiency for the following activation functions: ReLU, Sigmoid, Tanh, Mish, GELU, CELU, and SELU. The evaluation was performed within a strictly controlled regression setting using a single-layer fully connected network with 1024 hidden cells, mapping a scalar input to a scalar output. All models training employed the Adam optimizer with consistent hyperparameters–including learning rate, batch size, and number of epochs–across all configurations. Model performance was assessed via mean squared error (MSE) on two independent test sets comprising 10,000 and 100,000 samples, respectively, to evaluate both basic fitting capacity and generalization stability. This design isolates the effect of the activation function by holding all other architectural and optimization variables constant, thereby enabling a principled comparison of both computational cost and functional expressivity.

- ReLU [15]:

$$\mathrm{ReLU}(x) = \max(0, x)$$

- Sigmoid [16]:

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x};$$

- Tanh [17]:

$$\mathrm{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}};$$

- Mish [18]:

$$\mathrm{Mish}(x) = x \cdot \tan(\ln(1 + e^x));$$

- GELU [19]:

$$\mathrm{GELU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{2}\pi(x + 0.047715x^2)\right)\right);$$

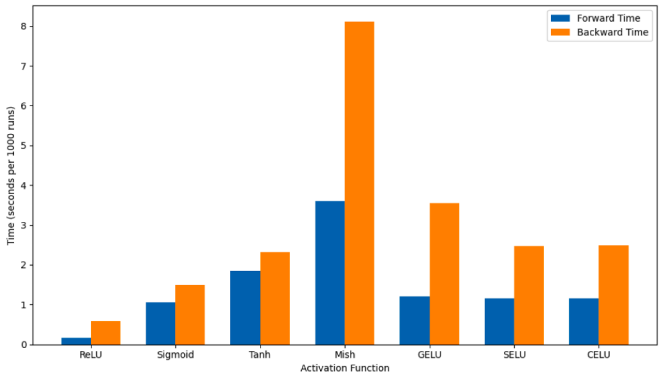


Fig. 5. Comparative time cost of 1000 computations with different activation functions.

Table 6
Comparative analysis of generalization error metrics using different activation functions in neural networks approximating $y = x^3$.

Function	MSE	R^2
GELU	1.40×10^{-6}	0.999994
CELU	1.60×10^{-6}	0.999994
SELU	1.71×10^{-5}	0.999933
Mish	8.08×10^{-8}	1.000000
Tanh	8.51×10^{-6}	0.999967
ReLU	9.87×10^{-7}	0.999996
Sigmoid	3.43×10^{-4}	0.998654

- SELU [20]:

$$\text{SELU}(x) = \begin{cases} \lambda x & x > 0, \\ \lambda\alpha(e^x - 1) & x \leq 0; \end{cases}$$

- CELU [21]:

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha(e^{x/a} - 1)).$$

As evidenced in Fig. 5, ReLU demonstrates superior computational speed owing to its simple thresholding operation that eliminates the need for exponential or complex mathematical computations. In contrast, Sigmoid and Tanh functions exhibit significantly longer propagation times - approximately $2-3$ times slower than ReLU [22] - due to their inherent dependence on exponential function evaluations.

The most computationally intensive activations prove to be Mish and GELU, where compound nonlinearities and approximation requirements substantially increase processing overhead. Specifically, GELU's reliance on error function approximations and Mish's composite nonlinear operations result in backward propagation times that are $4-5$ times greater than those of ReLU. These empirical findings quantitatively validate the critical trade-off between activation function expressivity and computational efficiency in deep learning systems.

Similarly, using polynomial fitting of $y = x^3$ from one-dimensional space as an example, we present evaluation metrics (Table 6) for networks employing different activation functions while maintaining fixed architecture and training hyperparameters:

As a fundamental component of neural networks, activation functions critically determine a model's nonlinear representational capacity. Empirical studies reveal substantial performance variations among different activation functions through diverse tasks.

The Mish activation function effectively combines ReLU's nonlinear characteristics with Tanh's smoothness, while its distinctive non-monotonic property enables superior capture of higher-order nonlinear patterns in polynomial approximation tasks. While ReLU offers notable advantages in training efficiency and mitigation of vanishing gradient problems, its application in certain function fitting scenarios may lead to suboptimal local minima or real-domain discontinuities, potentially compromising approximation accuracy. GELU employs Gaussian error function-based smoothing of negative values to maintain nonlinearity while enhancing gradient stability, whereas CELU utilizes exponential transformations for negative inputs at the cost of requiring additional parameter optimization (e.g. coefficient tuning).

In summary, our comprehensive evaluation of commonly used activation functions reveals a clear trade-off between computational efficiency and approximation accuracy. While ReLU remains the fastest option due to its minimalist structure, it lacks the expressive smoothness required in higher-order function approximation. Mish demonstrates superior fitting performance but suffers from prohibitively high computational cost, particularly during backpropagation.

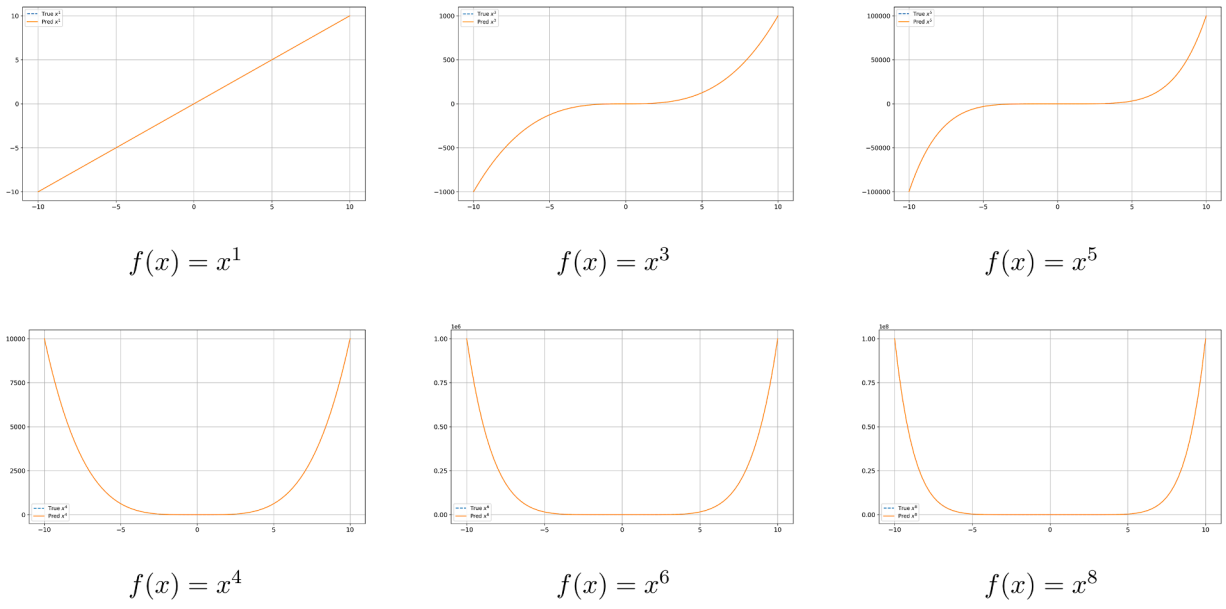


Fig. 6. Optimal neural network representation of one-dimensional polynomial basis functions.

Considering both performance and efficiency, GELU emerges as the most balanced choice. It offers excellent generalization capability comparable to Mish with significantly lower computational overhead, especially when compared to other smooth, non-monotonic activations. Its incorporation of the Gaussian error function contributes to improved gradient stability and smoother approximation across nonlinear regions, making it well-suited for the function approximation tasks considered in this study.

Therefore, in all subsequent experiments and network implementations throughout this work, GELU is adopted as the default activation function, providing an optimal compromise between accuracy and computational cost.

3.3. Performance of basis neural networks

This section investigates the capability of neural networks to represent polynomial basis functions in both one and two dimensions, with a particular focus on out-of-distribution generalization. In one dimension case, we employ a single-hidden-layer neural network to approximate polynomial basis functions of many degrees. The model is trained on data sampled from the interval $[-1, 1]$ and tested on an extended domain $[-10, 10]$. As shown in Fig. 6, the predicted values (orange curves) closely match the theoretical polynomials (blue curves), demonstrating the network's strong extrapolation ability. Notably, the generalization performance is influenced by the complexity of the target function. For higher-degree polynomials, improving model capacity–such as by increasing the number of hidden units or layers–or employing strategies like regularization or piecewise training may enhance extrapolation.

Extending this analysis to two dimension, we consider polynomial basis functions of the form:

$$f(x_1, x_2) = \sum_{i+j \le k} c_{ij} x_1^i x_2^j,$$

where $x_1, x_2 \in \mathbb{R}$, $i, j \ge 0$ are integers, and $c_{ij} \in \mathbb{R}$ are the coefficients.

Each monomial term $x_1^i x_2^j$ is modeled individually using a single neural network. Fig. 7 displays the predicted outputs for all terms with $i + j = 5$. Similar to one dimension case, the network is trained on $[-1, 1]^2$ and evaluated on the broader domain $[-10, 10]^2$, confirming the network's ability to generalize polynomial structure to unseen regions. Polynomial approximation examples for various polynomial bases can be found in the publicly available code repository accompanying this paper.

4. Numerical experiments

4.1. Function approximation in one dimension

This section applies the proposed initialization strategy combined with the least squares method to a variety of one-dimensional functions.

Test Case 1. As shown in Table 7, the models consistently achieve extremely small MSE and R^2 values very close to 1.0, indicating near-perfect approximation accuracy. These results demonstrate that the proposed approach provides highly reliable approximations across smooth, oscillatory, polynomial, and composite function classes.

This conclusion is further supported by the visualization results in Fig. 8, which illustrates the fitting performance of the six functions in six respective plots. In each plot, the blue line represents the true function, while the red line represents the least squares

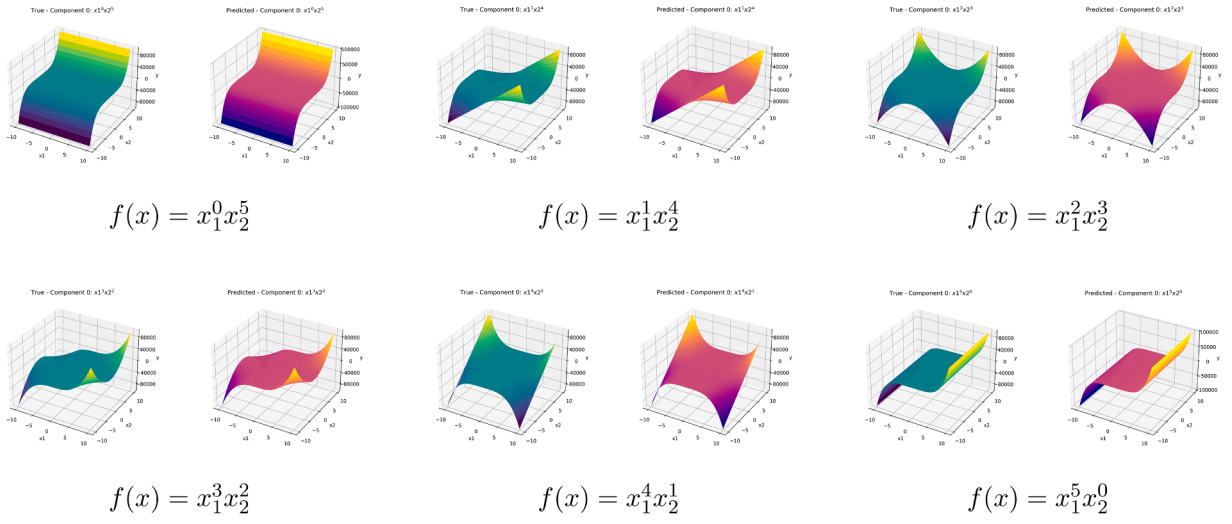


Fig. 7. Optimal neural network representation of two-dimensional polynomial basis functions.

Table 7
Metric results of numerical experiments in one dimension.

functions	max degree	domain	MSE	R^2
$f_1(x) = \exp(\sin(x))$	6	$[-1, 1]$	4.17×10^{-8}	1.000000
$f_2(x) = \ln(1 + x^2)$	8	$[-1, 1]$	1.05×10^{-8}	1.000000
$f_3(x) = \sin(\exp(x))$	8	$[-1, 1]$	1.64×10^{-8}	1.000000
$f_4(x) = \cos(x)$	8	$[4, 9]$	3.65×10^{-8}	1.000000
$f_5(x) = x^2$	4	$[-1, 9]$	8.39×10^{-6}	1.000000
$f_6(x) = \begin{cases} x^3 & \text{if } x < -1, \\ \sin\left(\frac{\pi x}{2}\right) & \text{if } -1 \le x < 1, \\ x & \text{otherwise.} \end{cases}$	12	$[-6, 4]$	5.38×10^{-3}	0.999998

Table 8
Comparison of different approximation methods for the Runge function after 4000 training epochs.

Method	Time (s)	MSE	R^2	Relative L^2 Error
Analytic Basis + LS	2.02×10^{-4}	1.60×10^{-3}	9.80×10^{-1}	1.01×10^{-1}
Random Initialized Basis	12.39	6.33×10^{-3}	9.22×10^{-1}	2.02×10^{-1}
Pretrained Basis (Proposed)	12.53	2.43×10^{-6}	$9.999\,70 \times 10^{-1}$	3.96×10^{-3}

fitting result. These plots are intended to visually convey the degree of agreement between the true function (True) and the fitted function (LS Fit). In nearly all plots, the red and blue lines completely overlap and are visually indistinguishable, further validating the high precision revealed by the metrics in the table. Even for more complex, piecewise-defined functions, the fitting performance remains strong, demonstrating the robustness of the two-stage fitting approach.

Test Case 2. To further evaluate the effectiveness of the proposed reusable initialization framework and provide a direct comparison with representative function approximation approaches, we consider the classical Runge function

$$f(x) = \frac{1}{1 + 25x^2},$$

which is well known for its sharp peak near the origin and therefore serves as a challenging benchmark for both polynomial and neural approximation methods. Three representative approaches are considered:

- **Analytic Basis + LS**: the classical polynomial basis combined with least-squares fitting;
- **Random Initialized Basis**: the proposed neural architecture trained from random initialization;
- **Pretrained Basis (Proposed)**: the same neural architecture initialized using the proposed reusable pretrained basis and subsequently fine-tuned.

Unless otherwise specified, all quantitative comparisons are reported after 4000 training epochs, which already clearly reveal the influence of different initialization strategies on the optimization process.

The quantitative results are summarized in Table 8. The analytic polynomial least-squares approximation is computationally inexpensive, requiring only 2.02×10^{-4} seconds, but its approximation accuracy is limited, yielding an MSE of 1.60×10^{-3}, an R^2

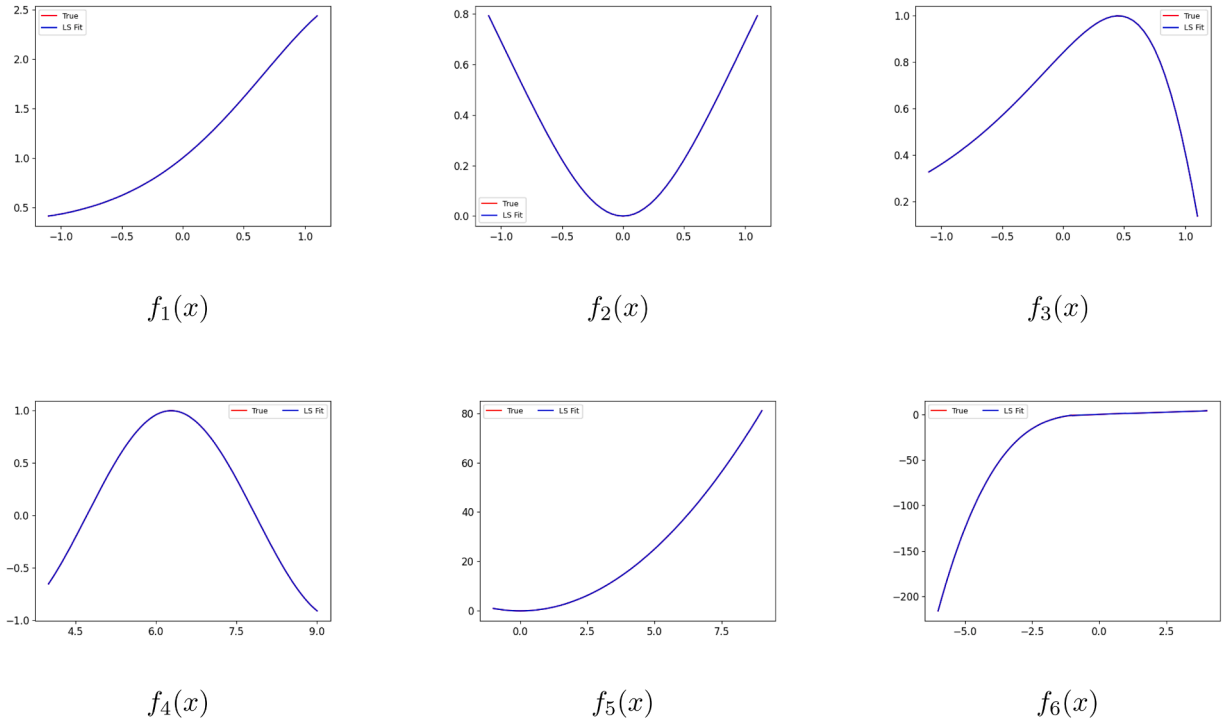
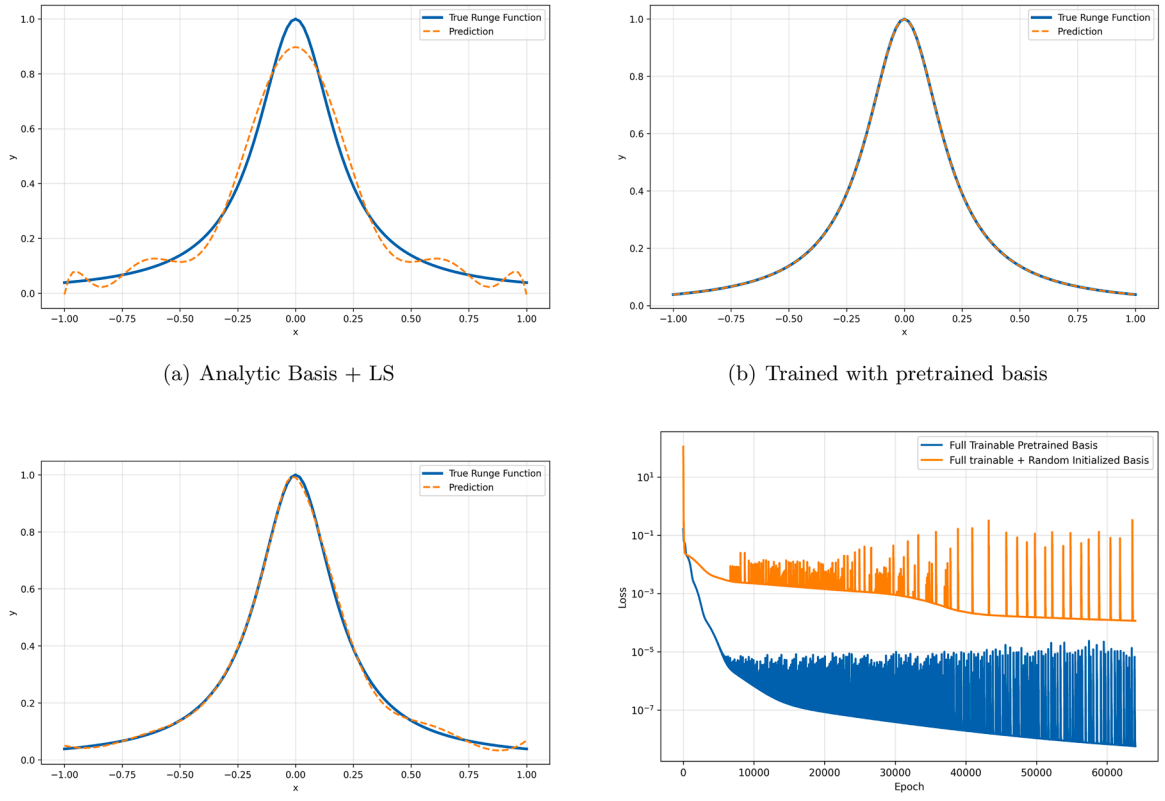


Fig. 8. Visualization of one dimensional function approximation.



(a) Analytic Basis + LS

(b) Trained with pretrained basis

(c) Trained with random initialized basis

(d) Training loss comparison (64000 epochs)

Fig. 9. Comparison of different approximation strategies for the Runge function using polynomials of degree 10.

Table 9
Metric results of numerical experiments in two dimension.

functions	max degree	domain	MSE	R^2
$f_1(\mathbf{x}) = \cos(x_1 + x_2)$	4	$[-1, 1]^2$	2.42×10^{-6}	0.999983
$f_2(\mathbf{x}) = \sin(\exp(x_1))$	6	$[-1, 1]^2$	2.48×10^{-7}	0.999956
$f_3(\mathbf{x}) = 2^{(x_1+x_2)}$	6	$[2, 3]^2$	1.32×10^{-6}	1.000000
$f_4(\mathbf{x}) = x_1^2 + x_2^2$	2	$[-5, 8]^2$	8.60×10^{-5}	1.000000

score of 9.80×10^{-1}, and a relative L^2 error of 1.01×10^{-1}. In contrast, the proposed reusable initialization framework achieves the highest approximation accuracy, with an MSE of 2.43×10^{-6}, an R^2 score of $9.999\,70 \times 10^{-1}$, and a relative L^2 error of only 3.96×10^{-3}. Although the optimization requires approximately 12.5 seconds, the approximation error is reduced by nearly three orders of magnitude compared with the analytic least-squares baseline.

To further isolate the contribution of the proposed initialization strategy, we compare the pretrained model with a randomly initialized neural network having exactly the same architecture, optimizer, and training procedure. Consequently, the only difference between the two neural models is the initialization strategy. As shown in Table 8, both methods require nearly identical computational time (12.53 versus 12.39 seconds), yet their approximation quality differs substantially. The randomly initialized model achieves an MSE of only 6.33×10^{-3}, an R^2 score of 9.22×10^{-1}, and a relative L^2 error of 2.02×10^{-1} after 4000 training epochs. In comparison, the proposed pretrained initialization reduces the MSE by more than three orders of magnitude while simultaneously improving both the R^2 score and the relative L^2 error. The quantitative improvements indicate that the performance gain originates primarily from the proposed reusable basis initialization rather than from the network architecture itself.

To better understand the optimization dynamics, we further extend the training process to 64,000 epochs and record the corresponding loss histories. As shown in Fig. 9, the pretrained model rapidly converges to a low-loss region during the early stage of optimization and subsequently exhibits smooth and stable convergence throughout the remaining iterations. In contrast, although the randomly initialized network also decreases the training loss, its optimization trajectory continues to exhibit pronounced oscillations and frequent loss spikes even after prolonged training. This observation suggests that the reusable pretrained basis provides a substantially better optimization starting point, leading to a smoother optimization landscape and improved optimization stability. Consequently, the superiority of the proposed initialization strategy is already evident after only 4000 epochs, whereas the extended 64000 epochs experiment mainly serves to visualize the long-term convergence behavior rather than to improve the reported quantitative metrics.

Overall, these experiments demonstrate that the proposed reusable initialization framework achieves an excellent balance between approximation accuracy, optimization stability, and computational cost. Compared with the classical polynomial least-squares approximation, it significantly improves prediction accuracy, while under essentially identical training time it substantially outperforms the randomly initialized neural network. The long-term convergence study further confirms that reusable pretrained basis initialization not only improves the final approximation quality but also accelerates convergence and yields considerably more stable optimization throughout the training process.

4.2. Function approximation in two dimension

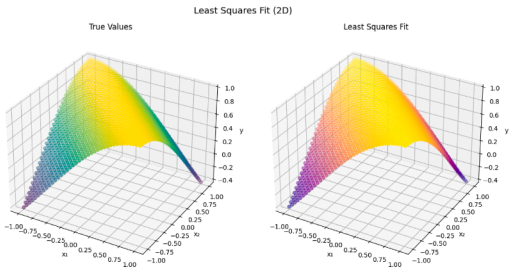
Here, we also employ the novel model initialization strategy introduced in this work in conjunction with the least squares approach to approximate general functions over a two dimensional domain, accompanied by quantitative performance metrics and graphical illustrations. The results presented in Fig. 10 and Table 9.

For relatively smooth and low-order functions such as the quadratic function $f_4(x_1, x_2)$, the neural network achieves nearly perfect reconstruction of the target surface. The predicted outputs closely match the ground truth, which is also reflected in the quantitative metrics with MSE $= 8.60 \times 10^{-7}$ and $R^2 = 1.0$, indicating excellent predictive accuracy.

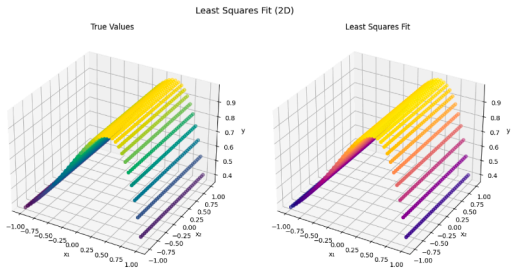
For trigonometric-type nonlinear functions, such as $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$, despite their oscillatory and composite structures, the networks deliver highly accurate results within the standard domain $[-1, 1]^2$. Specifically, the MSE metrics are 2.42×10^{-6} and 2.48×10^{-7}, respectively, with R^2 scores of 0.999983 and 0.999956, demonstrating that the models capture both periodicity and nonlinearity with remarkable precision.

Particularly notable is the performance on the exponential-type function $f_3(x_1, x_2)$, which is defined over a broader domain $[2, 3]^2$ and exhibits rapid growth. Even under these conditions, the network maintains strong predictive capability, achieving MSE $= 1.32 \times 10^{-6}$ and $R^2 = 1.0$. This suggests that the model generalizes effectively even for functions with steep gradients and large output ranges.

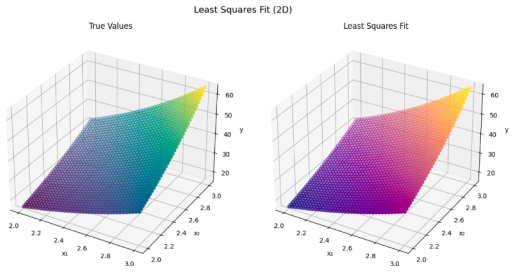
Overall, these results confirm that the proposed weights initialization strategy significantly enhances model performance within standard training regions. Moreover, when combined with the data mapping module, it contributes to improved out-of-domain generalization, highlighting the robustness and adaptability of the method in approximating a variety of function types in two dimensional space.



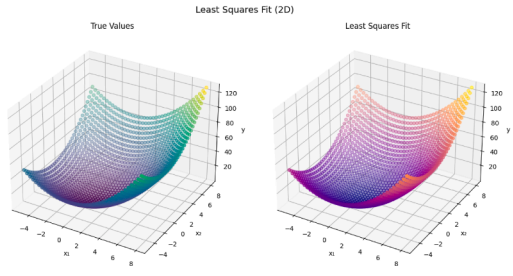
$$f_1(x_1, x_2) = \cos(x_1 + x_2)$$



$$f_2(x_1, x_2) = \sin(\exp(x_1))$$



$$f_3(x_1, x_2) = 2^{(x_1+x_2)}$$



$$f_4(x_1, x_2) = x_1^2 + x_2^2$$

Fig. 10. Visualization of two-dimensional function approximation.

5. Conclusion and future directions

In the present work, we propose a neural approximation framework that leverages reusable weight initialization based on pre-trained basis neural networks. By decoupling the representation of polynomials from the learning of target functions, our approach enables more stable and efficient training while concurrently enhancing approximation accuracy and generalization capability. Experimental evaluations on polynomial basis functions demonstrate that the proposed initialization strategy exhibits favorable performance in terms of both convergence behavior and extrapolation ability.

Several promising research directions emerge from this work. First, an important extension lies in higher-dimensional domains, where the proposed initialization strategy may help mitigate the curse of dimensionality via structured basis design or decomposition techniques. Second, integrating symbolic priors or physics-informed constraints into network architecture and training objectives represents a fruitful avenue–this enhancement can bolster both interpretability and generalization in scientific computing scenarios. Third, the proposed method holds potential for application in operator learning, where the efficient approximation of function families or parametric solutions can be substantially improved through reusable basis initialization. Beyond these, a particularly compelling direction is extending this framework to the numerical solution of partial differential equations (PDEs), where the neural approximation of solution functions–guided by the basis-initialized architecture–may offer a robust and flexible alternative to classical discretization methods.

To support the proposed framework, we have developed a modular software package that facilitates key components of the workflow, including data preparation, neural network training, and function approximation. The package is engineered for extensibility, featuring comprehensive documentation alongside an interactive interface for visualizing and analyzing approximation results. The full source code is openly available at: https://gitee.com/AIxinwen/nn4poly.

Acknowledgment

X. Hu and N. Yi was supported by the National Key R & D Program of China (2024YFA1012600). Y. Huang was supported in part by NSFC Project (12431014). P. Yin's research was supported by the University of Texas at El Paso Startup Award.

Data availability

Data will be made available on request.

References

[1] J. Hao, Y. Huang, N. Yi, P. Yin, Neural network-enhanced hr-adaptive finite element algorithm for parabolic equations, J. Comput. Phys. 565 (2026) 115173.
[2] M.J.D. Powell, Approximation Theory and Methods, Cambridge University Press, 1981.
[3] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[4] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (1989) 359–366.
[5] G. Cybenco, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst. 2 (4) (1989) 303–314.
[6] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, IEEE Trans. Neural Netw. 5 (1994) 157–166.
[7] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 9, 2010, pp. 249–256.
[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958.
[9] N. Houlsby, V. Giurgiu, S. Stojanovic, M.W. Chang, S.G. Kumar, P. Blunsom, Parameter-efficient transfer learning for NLP, 2019. arXiv preprint arXiv:1902.00751.
[10] X. Li, Y. Grandvalet, F. Davoine, Explicit inductive bias for transfer learning with convolutional networks, in: Proceedings of the 35th International Conference on Machine Learning, vol. 80, PMLR, 2018, pp. 2825–2834.
[11] Z. Shen, H. Yang, S. Zhang, Deep network approximation characterized by number of neurons, Commun. Comput. Phys. 28 (5) (2020) 1768–1811.
[12] Z. Lu, H. Pu, F. Wang, Z. Hu, L. Wang, The expressive power of neural networks: a view from the width, (2017). arXiv preprint arXiv:1610.01145v3.
[13] M. Tan, Q.V. Le, EfficientNet: rethinking model scaling for convolutional neural networks, in: Proceedings of the 36th International Conference on Machine Learning (ICML), 2019, pp. 6124–6133.
[14] Y. Bengio, A. Courville, P. Vincent, Representation learning: a review and new perspectives, IEEE Trans. Pattern Anal. Mach. Intell. 35, 2013, 1798–1828.
[15] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 15 (2011) 315–323.
[16] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536.
[17] Y. LeCun, L. Bottou, G.B. Orr, K.-R. Müller, Efficient Backprop, Neural Networks: Tricks of the Trade, 1998, pp. 9–50.
[18] D. Misra, Mish: a self-regularized non-monotonic activation function, (2019). arXiv preprint arXiv:1908.08681.
[19] D. Hendrycks, K. Gimpel, Gaussian error linear units (GELUs), 2016. arXiv preprint arXiv:1606.08415.
[20] G. Klambauer, T. Unterthiner, A. Mayr, S. Hochreiter, Self-normalizing neural networks, Adv. Neural Inf. Process. Syst. 30 (2017) 971–980.
[21] J.T. Barron, Continuously differentiable exponential linear units, 2017. arXiv preprint arXiv:1704.07483.
[22] V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, in: Proceedings of the 27th International Conference on Machine Learning (ICML), 2010.