Machine learning for numerical methods

Pengzhan Jin

2023.06.13

Outline

Initialization by PINN/Deep Ritz Method

Consider

$$\begin{cases} \mathcal{D}(u) = f & \text{in } \Omega, \\ \mathcal{B}(u) = g & \text{on } \partial\Omega, \end{cases} \tag{1}$$

where $\mathcal{D}$ is a differential operator and $\mathcal{B}$ is a boundary operator.

Phase I: deep learning-based PDE solvers

Phase II: traditional iterative methods

Initialization by PINN/Deep Ritz Method

Least Square Method (PINN): A DNN $\phi(\boldsymbol{x};\boldsymbol{\theta}^*)$ is constructed to approximate the solution $u(\boldsymbol{x})$ for $\boldsymbol{x} \in \Omega$ via minimizing the square loss

$$\begin{aligned} \boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \\ \mathcal{L}(\boldsymbol{\theta}) &:= \mathbb{E}_{\boldsymbol{x}\in\Omega}\left[|\mathcal{D}\phi(\boldsymbol{x};\boldsymbol{\theta}) - f(\boldsymbol{x})|^2\right] + \gamma\mathbb{E}_{\boldsymbol{x}\in\partial\Omega}\left[|\mathcal{B}\phi(\boldsymbol{x};\boldsymbol{\theta}) - g(\boldsymbol{x})|^2\right], \end{aligned} \tag{2}$$

with a positive parameter γ.

Initialization by PINN/Deep Ritz Method

Variational Methods (Deep Ritz Method): (1) is solved via a variational minimization

$$u^* = \arg\min_{u \in H} J(u), \tag{3}$$

where the Hilbert space H is an admissible space, and $J(u)$ is a nonlinear functional over H. Then, the solution space H is parametrized via DNNs, i.e., $H \approx \{\phi(\boldsymbol{x}; \boldsymbol{\theta})\}_{\boldsymbol{\theta}}$. After parametrization, (3) is approximated by the following problem:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\phi(\boldsymbol{x}; \boldsymbol{\theta})). \tag{4}$$

Initialization by PINN/Deep Ritz Method

Example:

Consider the following semilinear elliptic equation

$$\begin{cases} -\Delta u + f(u) = 0 & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases} \tag{5}$$

where Ω is a bounded convex polygon in $\mathbb{R}^d$ $(d = 1, 2)$, and $f(u)$ is a sufficiently smooth function.

Let $V = H_0^1(\Omega)$. Then the variational formulation of problem (5) is to find $u \in V$ such that

$$a(u, \chi) := (\nabla u, \nabla \chi) + (f(u), \chi) = 0, \quad \chi \in V. \tag{6}$$

Initialization by PINN/Deep Ritz Method

We next consider the finite element method for solving problem (6). Let $\mathcal{T}_h$ be a quasi-uniform and shape regular triangulation of Ω into K. Write $h_K = \text{diam}(K)$ and $h := \max_{K\in\mathcal{T}_h} h_K$. Introduce the Courant element space by

$$V_h = \{\phi \in C(\bar{\Omega}) : \phi|_K \in \mathbb{P}_1(K) \text{ for all } K \in \mathcal{T}_h\} \cap V, \tag{7}$$

where $\mathbb{P}_1(K)$ denotes the function space consisting of all linear polynomials over K. Then the finite element method is to find $u^h \in V_h$ such that

$$a(u^h, \chi) = 0, \quad \chi \in V_h. \tag{8}$$

Initialization by PINN/Deep Ritz Method

Algorithm 1 A hybrid Newton's method for semilinear problems

Input: the target accuracy ϵ, the maximum number of iterations $N_{\max}$, the approximate solution in a form of a DNN u^{DL} in Phase I of Int-Deep.
Output: $u^h = u^h_{k+1}$.

Initialization: Let $u^h_0 = I_h u^{DL}$, $k = 0$, and $e_k = 1$;
while $e_k > \epsilon$ and $k < N_{\max}$ **do**
Find $v^h_k \in V_h$ such that

$$(\nabla v^h_k, \nabla \chi) + (f'(u^h_k) v^h_k, \chi) = -(\nabla u^h_k, \nabla \chi) - (f(u^h_k), \chi), \quad \chi \in V_h.$$

Let $u^h_{k+1} = u^h_k + v^h_k$.
$e_{k+1} = \|u^h_{k+1} - u^h_k\|_0 / \|u^h_k\|_0$, $k = k + 1$.
end while

Initialization by PINN/Deep Ritz Method

Numerical example:

Consider the following semilinear elliptic equations

$$\begin{cases} -\Delta u - (u-1)^3 + (u+2)^2 = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases}$$

where $\Omega = (0,1)^d$ with $d = 1$ or 2. We choose f such that the problem has an exact solution $u(x) = 3\sin(2\pi x)$ for $d = 1$ and $u(x) = 3\sin(2\pi x)\sin(2\pi y)$ for $d = 2$.

Initialization by PINN/Deep Ritz Method

Table: The performance of Newton's method with different initial guesses u_0. ω stands for a Gaussian random noise with mean zero and unit variance.

u_0	#K	$\|e_0^h\|_{0,\infty,h}$	$\|e^h\|_{0,\infty,h}$	u_0	#K	$\|e_0^h\|_{0,\infty,h}$	$\|e^h\|_{0,\infty,h}$
1	5	4.00e+0	1.95e+0	ω	6	5.68e+0	3.05e+0
2	5	5.00e+0	1.95e+0	$1+\omega$	5	6.33e+0	1.95e+0
5	15	8.00e+0	1.28e+5	$-1+\omega$	15	6.95e+0	2.65e+6
-1	15	4.00e+0	9.50e+4	$u+\omega$	6	4.03e+0	1.82e-5
-2	12	5.00e+0	4.66e+0	$u+2.5\times\omega$	15	8.89e+0	2.10e+5
-5	15	8.00e+0	6.45e+4	u^{DL}	5	2.97e-1	1.82e-5

Initialization by PINN/Deep Ritz Method

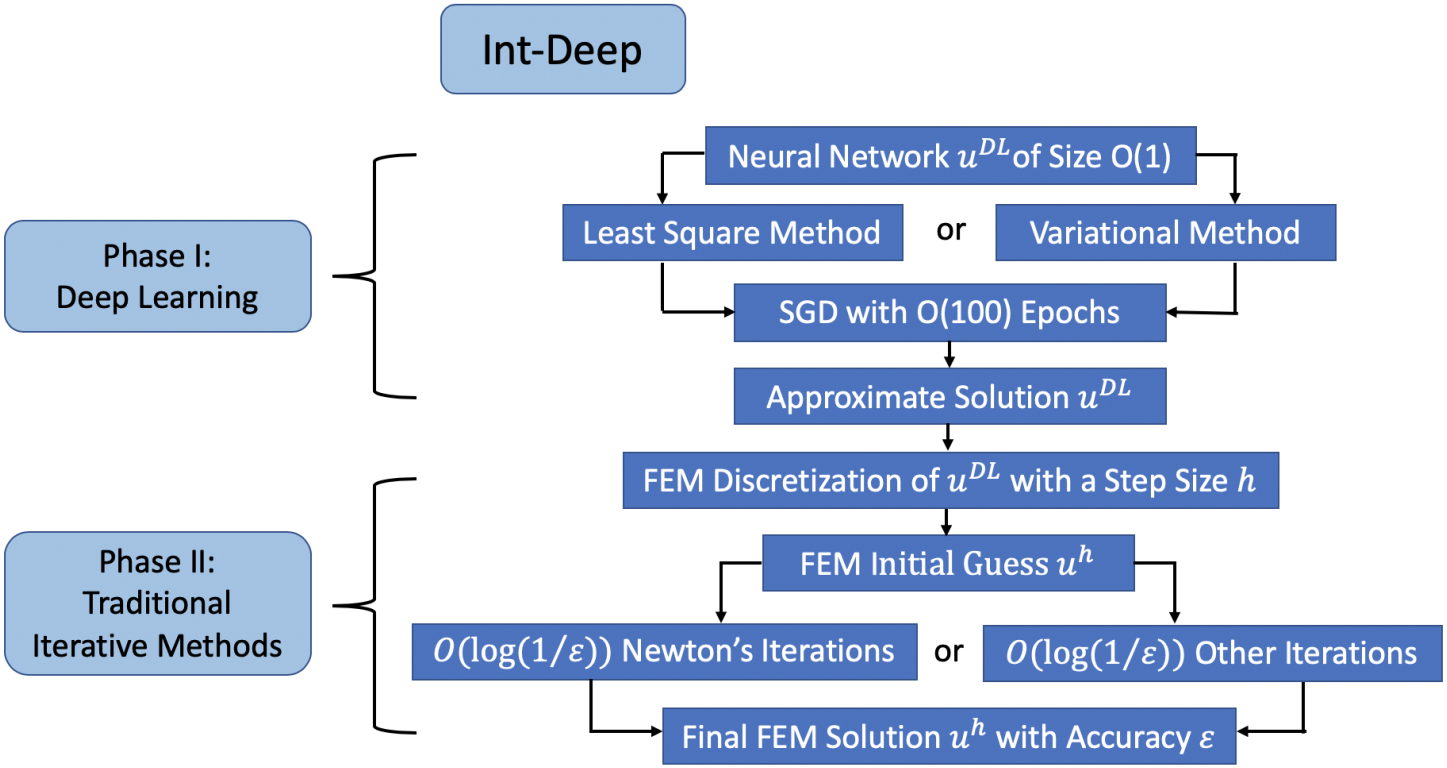


Figure: Computational flow of Int-Deep.

Initialization by PINN/Deep Ritz Method

Reference

J. Huang, H. Wang, H. Yang, Int-deep: A deep learning initialized iterative method for nonlinear problems, Journal of Computational Physics 419 (2020) 109675.

Initialization by operator regression

We consider the following linear differential equation

$$\mathcal{L}_{\mathbf{x}}(u; k) = f, \mathbf{x} \in \Omega \tag{9a}$$

$$\mathcal{B}_{\mathbf{x}}(u) = g, \mathbf{x} \in \partial\Omega \tag{9b}$$

where $\mathcal{L}_{\mathbf{x}}$ is the differential operator, $\mathcal{B}_{\mathbf{x}}$ is the boundary operator, $k = k(\mathbf{x})$ parameterizes $\mathcal{L}_{\mathbf{x}}$, $f = f(\mathbf{x})$ and $g = g(\mathbf{x})$ are the right-hand-side forcing terms, and $u = u(\mathbf{x})$ is the solution. Assuming that g is a known fixed function, Eq. 9 defines a family of differential equations parameterized by f and k.

Initialization by operator regression

We first need to train a DeepONet offline. This DeepONet approximates the solution operator $\mathcal{G}$ defined by

$$\mathcal{G} : k, f \mapsto u \text{ s.t. Eq. 9 holds,} \tag{10}$$

where we have assumed the uniqueness of the solution u. DeepONet receives $k(x)$ and $f(x)$ in the form of discrete evaluations on $n_{\mathrm{D}} + 1$ uniform points, respectively. We denote this discretization associated with DeepONet as $\Omega^{h_{\mathrm{D}}}$.

For a new given (k, f), we can predict an approximate solution $\tilde{u} = \mathit{DeepONet}(k, f)$ as an initial guess for traditional iterative methods.

Hybrid Iterative Numerical Transferable Solver (HINTS)

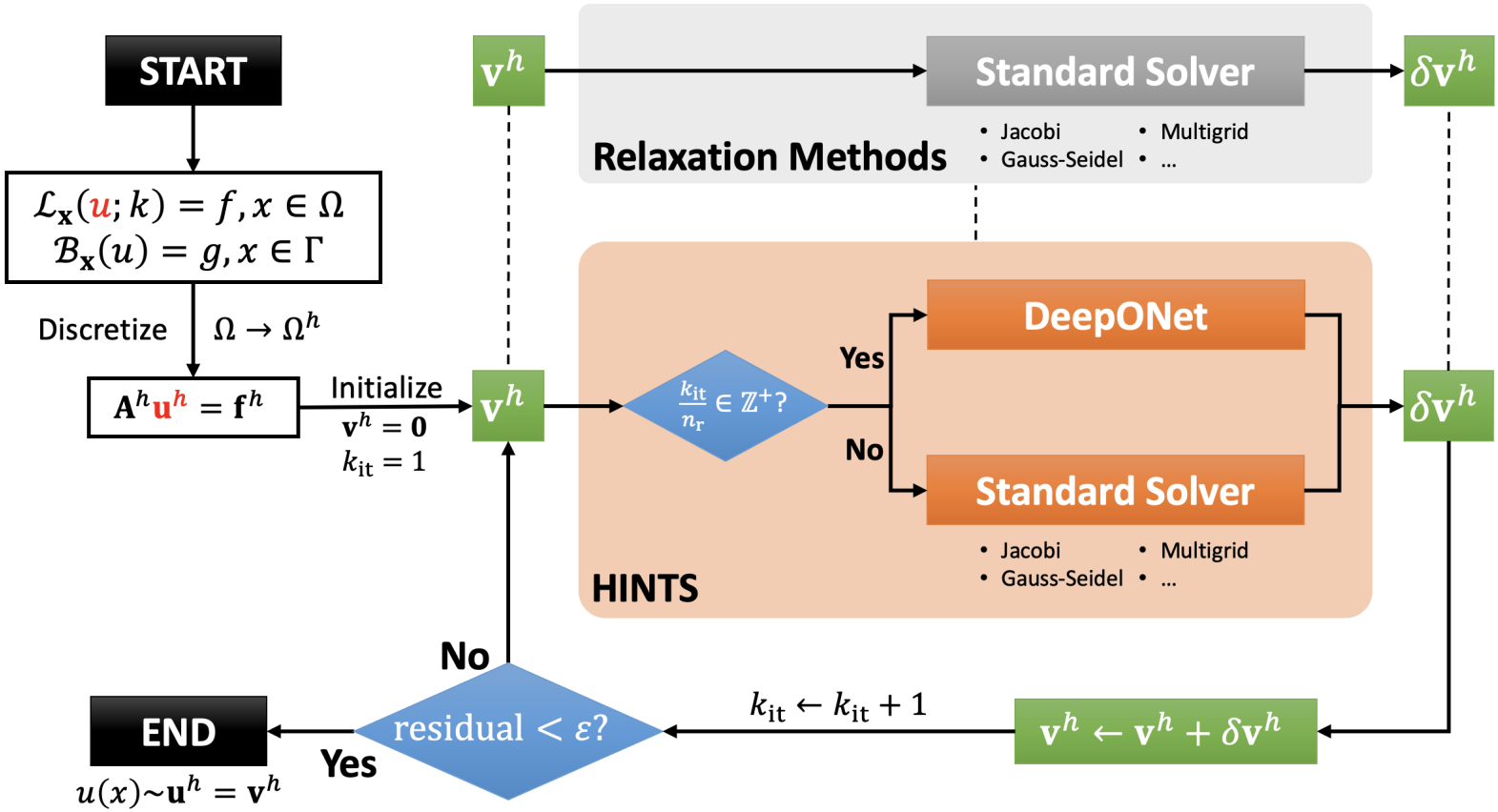


Figure: **Overview of the Hybrid Iterative Numerical Transferable Solver (HINTS).**

Hybrid Iterative Numerical Transferable Solver (HINTS)

Algorithm S3 HINTS-Jacobi

function HINTS_JACOBI($k(\mathbf{x}), f(\mathbf{x}), \text{PDE}$)

$\mathbf{A}^h, \mathbf{f}^h = \text{DISCRETIZE}(k(\mathbf{x}), f(\mathbf{x}), \text{PDE})$ ▷ discretization; to solve $\mathbf{v}^h = (\mathbf{A}^h)^{-1}\mathbf{f}^h$

$\mathbf{v}^h \leftarrow \mathbf{0}^h$ ▷ initial guess of the solution

$k_{\text{it}} \leftarrow 1$

while $k_{\text{it}} \leq n_{\text{it}}$ and not converged **do**

$\mathbf{r}^h \leftarrow \mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h$

if k_{it} mod $n_{\text{r}} = 0$ **then** ▷ condition for invoking DeepONet

$r(\mathbf{x}) = \text{REVERSE\_DISCRETIZE}(\mathbf{r}^h)$ ▷ see Section S3E

$\delta\mathbf{v}^h\ (= \delta v(\mathbf{x})) \leftarrow \text{DEEPONET}(k(\mathbf{x}), r(\mathbf{x}))$

$\mathbf{v}^h \leftarrow \mathbf{v} + \delta\mathbf{v}^h$

else

$\mathbf{v}^h \leftarrow \text{DAMPED\_JACOBI}(\mathbf{A}^h, \mathbf{f}^h, \mathbf{v}^h)$ ▷ see Alg. S1

(or equivalently: $\mathbf{v}^h \leftarrow \mathbf{v}^h + \text{DAMPED\_JACOBI}(\mathbf{A}^h, \mathbf{r}^h, \mathbf{0}^h)$)

end if

$k_{\text{it}} \leftarrow k_{\text{it}} + 1$

end while

return $\mathbf{v}^h\ (= v(\mathbf{x}))$

end function

Hybrid Iterative Numerical Transferable Solver (HINTS)

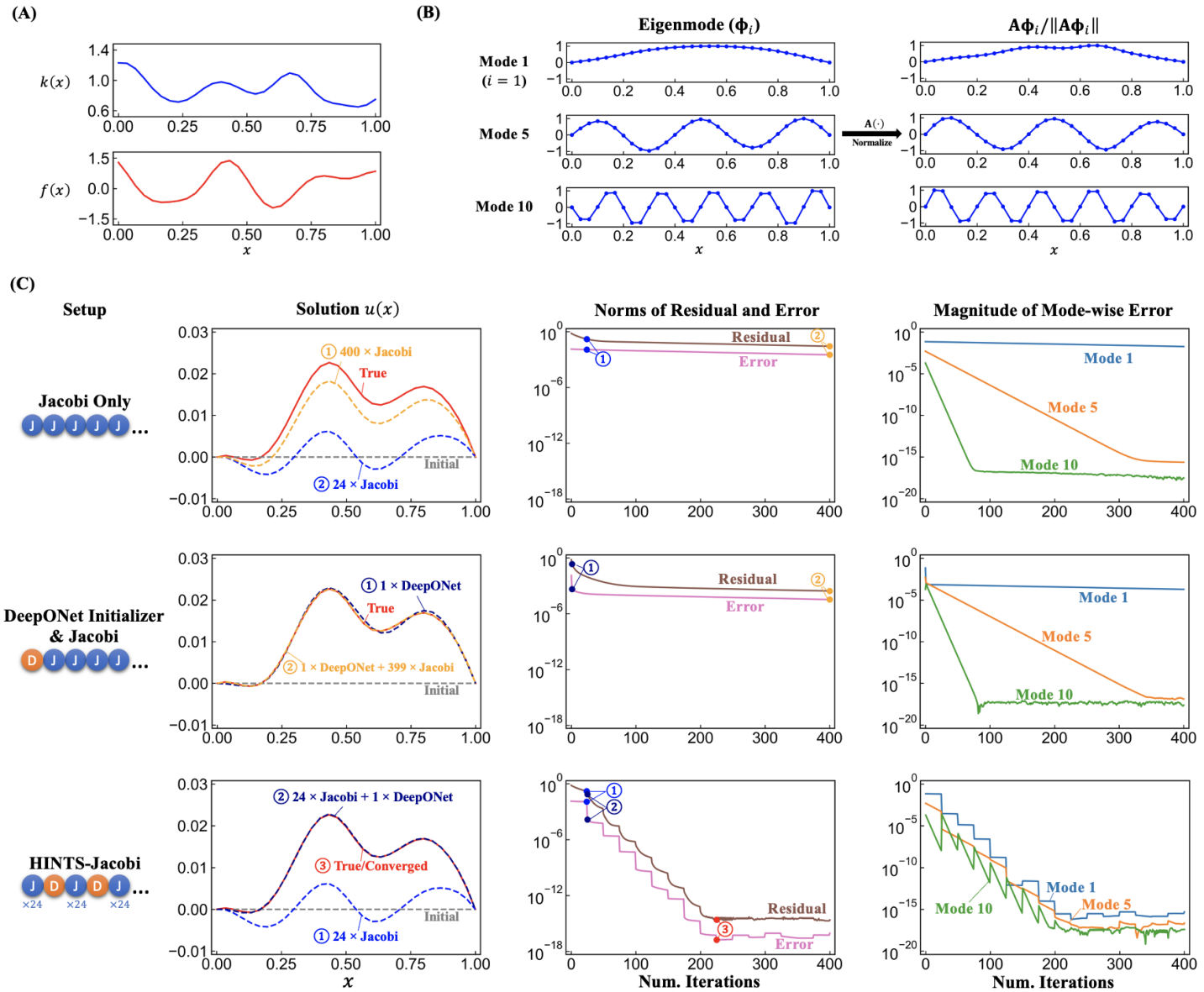
Consider Poisson equation:

$$\nabla \cdot \Big(k(\mathbf{x}) \nabla u(\mathbf{x}) \Big) + f(\mathbf{x}) = 0, \quad \mathbf{x} \in \Omega \subset \mathbb{R}^d \tag{11a}$$

$$u(\mathbf{x}) = 0, \quad \mathbf{x} \in \partial\Omega; \tag{11b}$$

As a prototypical example, we consider the Poisson equation in one dimension ($d = 1$), defined in $\Omega = (0, 1)$. The goal of HINTS-Jacobi is to solve this equation with arbitrary $k(x)$ and $f(x)$. We first train a DeepONet with paired data $[k(x), f(x)]$ (generated by a Gaussian random field) and corresponding $u(x)$. After training, we employ HINTS-Jacobi to solve for new instances of $k(x)$ and $f(x)$.

Hybrid Iterative Numerical Transferable Solver (HINTS)



Hybrid Iterative Numerical Transferable Solver (HINTS)

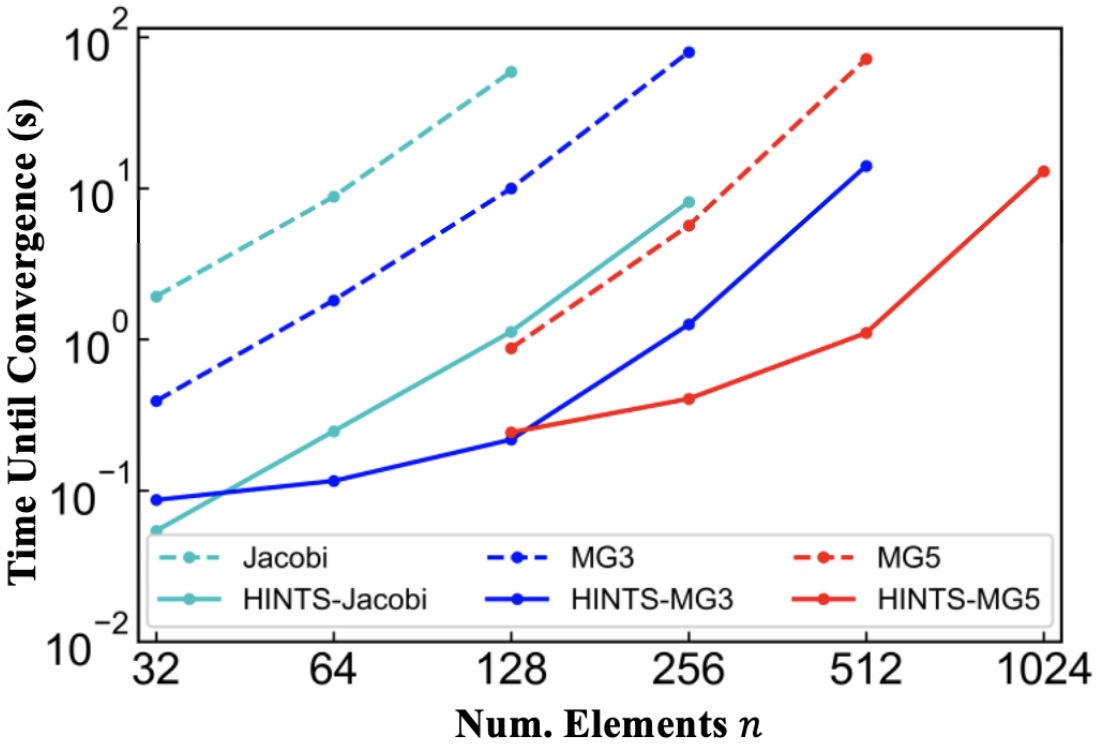


Figure: **Computational Cost of Different Methods for 1D Poisson Equation.** All multigrid (MG) methods use damped Jacobi relaxation. MG3 and HINTS-MG3 use three grid levels, and MG5 and HINTS-MG5 use five grid levels.

Improvement for HINTS

$$C(K,Y)=\begin{cases} C(K_1\times\cdots\times K_n\times K_0), & (\text{standard NN})\\ C(K_1\times\cdots\times K_n)\hat{\otimes}_\varepsilon C(K_0), & (\text{DeepONet})\\ C(K_1)\hat{\otimes}_\varepsilon\cdots\hat{\otimes}_\varepsilon C(K_n)\hat{\otimes}_\varepsilon C(K_0). & (\text{MIONet}) \end{cases}$$

It is better to use MIONet instead of DeepONet.

For $\mathcal{G}:(k,f)\mapsto u$, we are able to construct MIONet which is nonlinear with respect to k but linear with respect to f. We can also take into account the boundary condition and learn the map $\mathcal{G}:(k,f,g)\mapsto u$ via MIONet.

Hybrid Iterative Numerical Transferable Solver (HINTS)

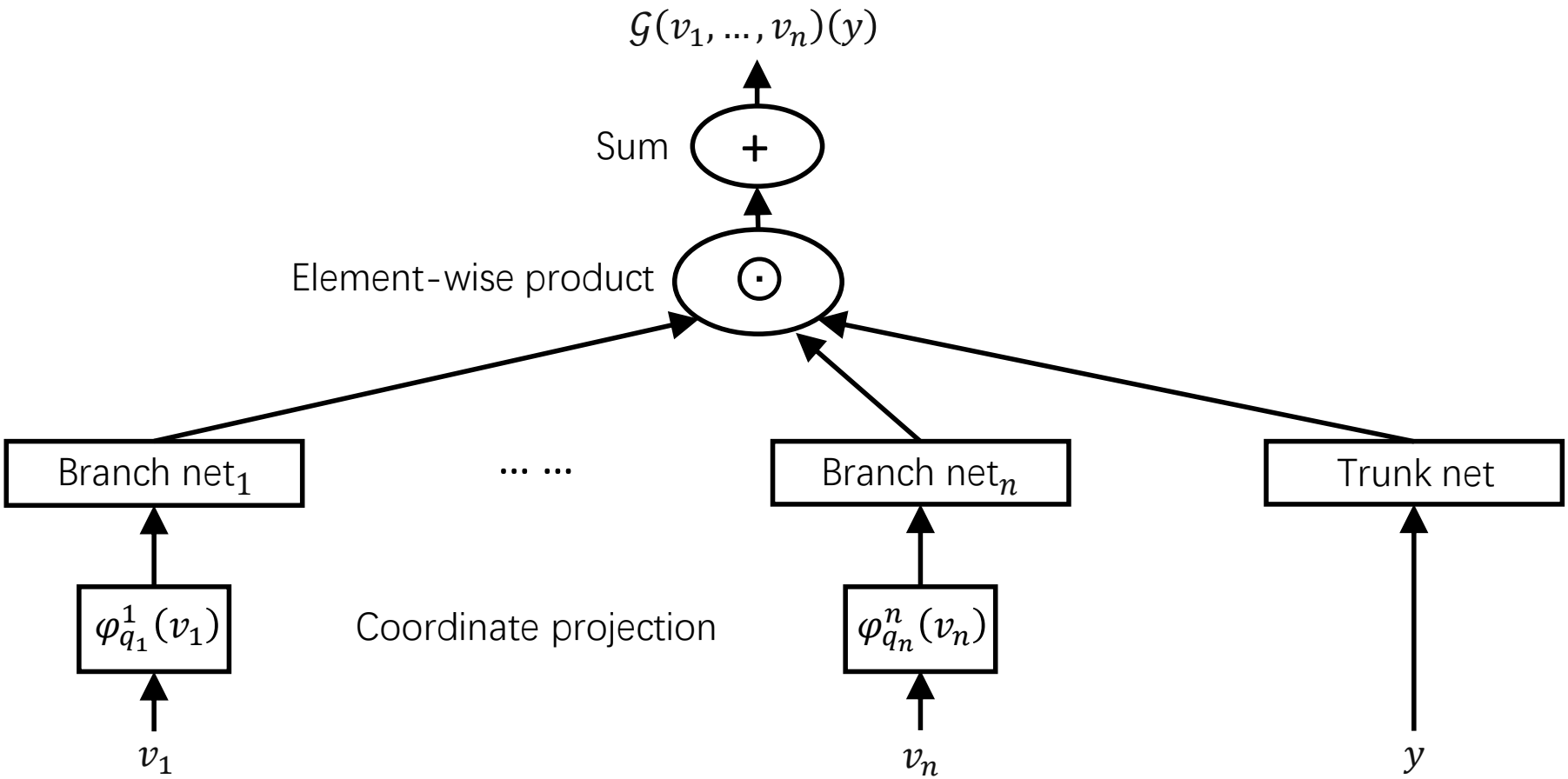


Figure: Architecture of MIONet.

Hybrid Iterative Numerical Transferable Solver (HINTS)

Reference

E. Zhang, A. Kahana, E. Turkel, R. Ranade, J. Pathak, and G. E. Karniadakis. A hybrid iterative numerical transferable solver (hints) for pdes based on deep operator network and relaxation methods. arXiv preprint arXiv:2208.13273, 2022.

Learning ODE integrators

Consider a differential equation $\dot{y}(t) = f(y(t))$, $y(0) = y_0$ on $t \in [0, T]$ can be recast as

$$L(y, F)(t) = y(t) - y_0 - \int_0^t f(y(s))ds = 0,$$

where F comprises the ODE's information related to vector field f and initial condition y_0. Note here that the extrinsic input is (f, y_0) and the output of L is a function of time.

Learning ODE integrators

For a fixed problem type L, a solution operator is a mapping $A : \mathcal{F} \to \mathcal{Y}$, which produces the true solution $y = A(F)$ given a problem instance F, so that $L(y, F) = 0$. Often, we do not have an explicit means to represent A. Thus, for computational purposes we design a numerical algorithm that computes an estimate solution $y \approx \hat{A}(F, h)$, where $h > 0$ denotes the accuracy of approximation. We call $\hat{A} : \mathcal{F} \times \mathbb{R}_+ \to \mathcal{Y}$ an approximate solver, which is consistent if $\lim_{h \to 0} \hat{A}(\cdot, h) = A(\cdot)$. In this work, we also consider parametric approximate solvers $\hat{A} : \mathcal{F} \times \mathbb{R}_+ \times \Theta \to \mathcal{Y}$ where Θ is a set of solver parameters that can be optimized according to problem settings.

Learning ODE integrators

Classical numerical methods design the solver $\hat{A}(\cdot, h)$ by requiring it to perform well *over a large and, in general, unstructured class* $\mathcal{F}$. For example, one might seek

$$\sup_{F \in \mathcal{F}} \|L(\hat{A}(F, h), F)\| = \mathcal{O}(h^{\alpha}), \qquad \alpha > 0. \tag{12}$$

However, often in practice we are not interested in such a worst-case approach. In fact, we may want to solve a special class of problems belonging to $\mathcal{F}$, and we may only be interested in the average performance of our method on this class of problems. Hence, instead of (12), we may require

$$E_{\mu}[\|L(\hat{A}(F, h), F)\|] = \int_{F \in \mathcal{F}} \|L(\hat{A}(F, h), F)\| d\mu(F) = \mathcal{O}(h^{\alpha}), \qquad \alpha > 0, \tag{13}$$

where μ is a probability measure on $\mathcal{F}$ and may be supported on a very small subset. This imparts structure in $\mathcal{F}$ through μ, and our algorithm is now only required to perform well in expectation under this structure.

Learning ODE integrators

In the simplest case of explicit, one-step integrators, one iterates the following formula based on an integrator $I_{\hat{A}}$ that computes

$$\hat{\mathbf{y}}_{n+1} = I_{\hat{A}}(\mathbf{f}, \hat{\mathbf{y}}_n, h), \qquad \hat{\mathbf{y}}_0 = \mathbf{y}_0. \tag{14}$$

This produces an approximate sequence $\hat{\mathbf{y}}_n \approx \mathbf{y}(nh)$. In fact, we can understand the mapping from $F = (\mathbf{y}_0, \mathbf{f})$ to a continuous-time interpolation of $\{(nh, \hat{\mathbf{y}}_n)\}$ as a solver $\hat{A}(\cdot, h) : \mathcal{F} \to \mathcal{Y}$.

Learning ODE integrators

RK-like Neural Network (RK-NN) Architecture

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}\left((\hat{\mathbf{y}}_n)\right) \\
\mathbf{k}_i &= \mathbf{f}\left(\hat{\mathbf{y}}_n + h\sum_{j=1}^{i-1}\theta_{i-1,j}\mathbf{k}_j\right), \quad i = 2,\ldots,m \\
\hat{\mathbf{y}}_{n+1} &= \hat{\mathbf{y}}_n + h\sum_{i=1}^{m}\theta_{ci}\mathbf{k}_i.
\end{aligned} \tag{15}$$

$\{\theta_{i,j}, \theta_{ci}\}$ are the trainable parameters. We apply a softmax activation $\theta_{ci} = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}}$ for $i = 1,\ldots,m$ to guarantees the consistency.

Learning ODE integrators

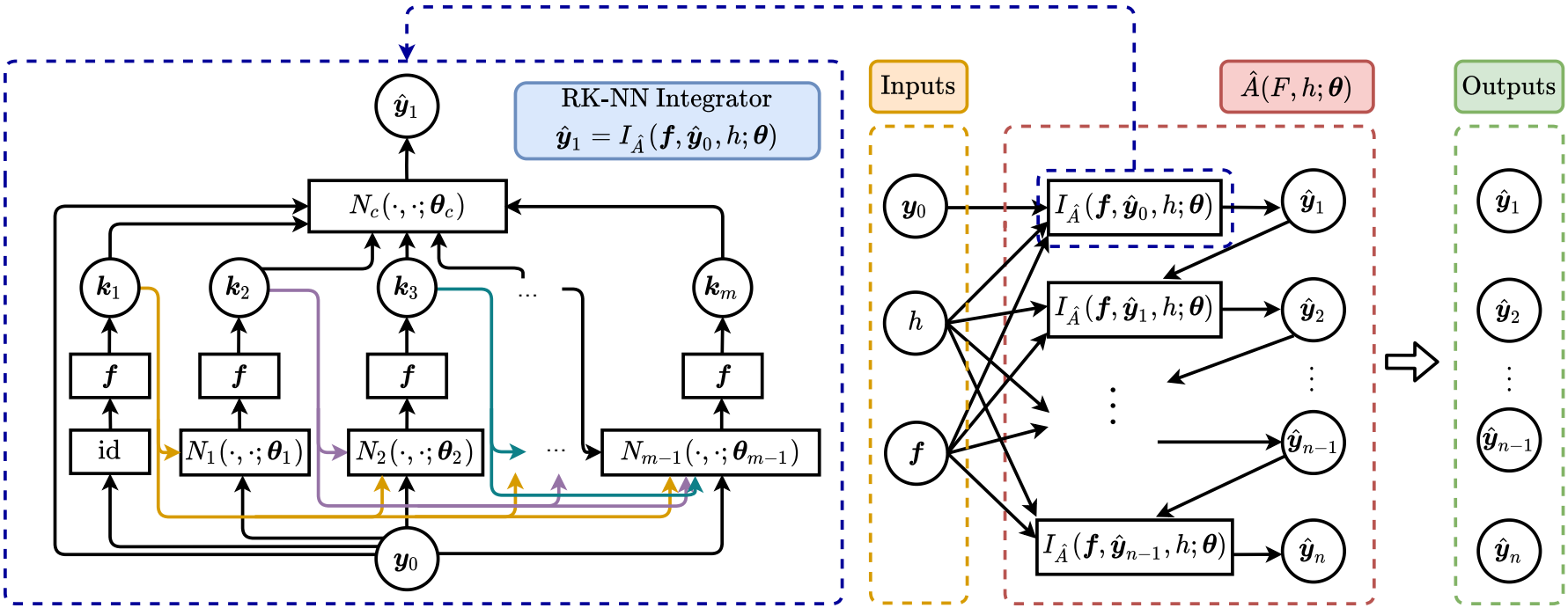


Figure: RK-like Neural Network (RK-NN) Architecture.

Learning ODE integrators

Let μ be a probability measure on $\mathcal{F}$, representing a particular distribution of tasks F, we also consider a measure over the step sizes $h \sim \nu$. Let $\mathcal{L} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+$ be a loss functions which is minimized when its first two arguments are equal. Then, we consider the following optimization problem

$$\begin{aligned} \min_{\theta \in \Theta} \quad & \mathcal{E}_{F\sim\mu, h\sim\nu}\left[\mathcal{L}(\mathbf{y}_n, \hat{\mathbf{y}}_n) + \mathcal{R}(\hat{A}(\cdot,\cdot;\theta), F, h)\right] \\ s.t. \quad & \mathbf{y}_n = A(F)(nh) = \mathbf{y}_0 + \int_0^{nh} \mathbf{f}(\mathbf{y}(s))ds, \\ & \hat{\mathbf{y}}_n = \hat{A}(F, h; \theta)(nh) = I_{\hat{A}}(\mathbf{f}, \hat{\mathbf{y}}_{n-1}, h; \theta), \\ & F = (\mathbf{f}, \mathbf{y}_0), \\ & n \geq 0. \end{aligned} \tag{16}$$

The last term $\mathcal{R}(\hat{A}(\cdot,\cdot;\theta), F, h)$ represents a regularization term that allows us to promote certain order of accuracy.

Learning ODE integrators

We make the simple choice of a scaled square loss

$$\mathcal{L}(\mathbf{y}_n, \hat{\mathbf{y}}_n) = \frac{\|\mathbf{y}_n - \hat{\mathbf{y}}_n\|^2}{\|\mathbf{y}_n - \hat{\mathbf{y}}_n^{(RK)}\|^2}, \tag{17}$$

where $F = (\mathbf{f}, \mathbf{y}_0)$, $\mathbf{y}_n = A(F)(nh)$, $\hat{\mathbf{y}}_n = \hat{A}(F, h; \theta)(nh) = I_{\hat{A}}(\mathbf{f}, \hat{\mathbf{y}}_{n-1}, h; \theta)$ and $\hat{\mathbf{y}}_n^{(RK)} = \hat{A}_{RK}(F, h)(nh) = I_{\hat{A}_{RK}}(\mathbf{f}, \hat{\mathbf{y}}_{n-1}^{(RK)}, h)$. $\hat{\mathbf{y}}$ is the prediction from our RK-NN integrator and $\hat{\mathbf{y}}^{(RK)}$ is from the RK method. Here, we consider one-step prediction by setting $n = 1$.

Learning ODE integrators

Rregularizer

$$\mathcal{R}(\hat{A}(\cdot,\cdot;\theta),F,h)=\sum_{i=1}^{\alpha}\left\|\left.\frac{d^i}{dh^i}\right|_{h=0}(\mathbf{y}_1-\hat{\mathbf{y}}_1)\right\|_2^2, \tag{18}$$

which promotes the desired order of convergence.

Learning ODE integrators

Algorithm 3.1

Data: $\mathcal{D} = \{F_j, h_j\}_{j=1}^N$;
Initialize: Random $\boldsymbol{\theta}_0$ for the solver $\hat{A}(\cdot,\cdot;\boldsymbol{\theta}_0) : \boldsymbol{\theta}_0 \in \Theta, h > 0$;
Set tolerance $\epsilon > 0$; Optimizer `Opt`;
for $k = 0, 1, \ldots, \#Iterations$ **do**
  for all F_j, h_j **do**
    Calculate $\boldsymbol{y}_n^{(j)} = A(F_j)(nh_j)$;
    Calculate $\hat{\boldsymbol{y}}_n^{(j)} = \hat{A}(F_j, h_j; \boldsymbol{\theta})(nh_j)$;
    Calculate $\hat{\boldsymbol{y}}_n^{(RK)(j)} = \hat{A}_{RK}(F_j, h_j)(nh_j)$;
    Calculate the scaled loss: $\mathcal{L}(\boldsymbol{y}_n^{(j)}, \hat{\boldsymbol{y}}_n^{(j)})$;
    Calculate the regularizer: $\mathcal{R}(\hat{A}(\cdot,\cdot;\boldsymbol{\theta}), F_j, h_j)$;
  end for
  Evaluate $\ell = \frac{1}{N}\sum_{j=1}^N \left[\mathcal{L}(\boldsymbol{y}_n^{(j)}, \hat{\boldsymbol{y}}_n^{(j)}) + \mathcal{R}(\hat{A}(\cdot,\cdot;\boldsymbol{\theta}), F_j, h_j)\right]$;
  Update parameters $\boldsymbol{\theta}$ using `Opt` to minimize ℓ;
  Compute the relative error: $\gamma = \frac{1}{N}\sum_{j=1}^N \mathcal{L}(\boldsymbol{y}_n^{(j)}, \hat{\boldsymbol{y}}_n^{(j)})$;
  if $\gamma < \epsilon$ **then**
    break;
  end if
end for
return Solver $\hat{A}(\cdot,\cdot;\boldsymbol{\theta})$.

Learning ODE integrators

Linear Task Family: The simplest task family is the pairs of stable linear functions and initial conditions $(\mathbf{f}, \mathbf{y}_0) \in \mathcal{F}$, which has the form

$$\begin{aligned} \mathcal{F} &= \{\mathbf{y} \mapsto -a\mathbf{y} \mid a > 0\} \times \{\mathbb{R}\}, \\ \mu &= \text{Distribution}(\{\mathbf{y} \mapsto -a\mathbf{y}; a \sim U(1,5)\}) \times U(-5,5). \end{aligned} \tag{19}$$

In this case, the closed-form solution is $\boldsymbol{y}(t) = e^{-at}\boldsymbol{y}_0$.

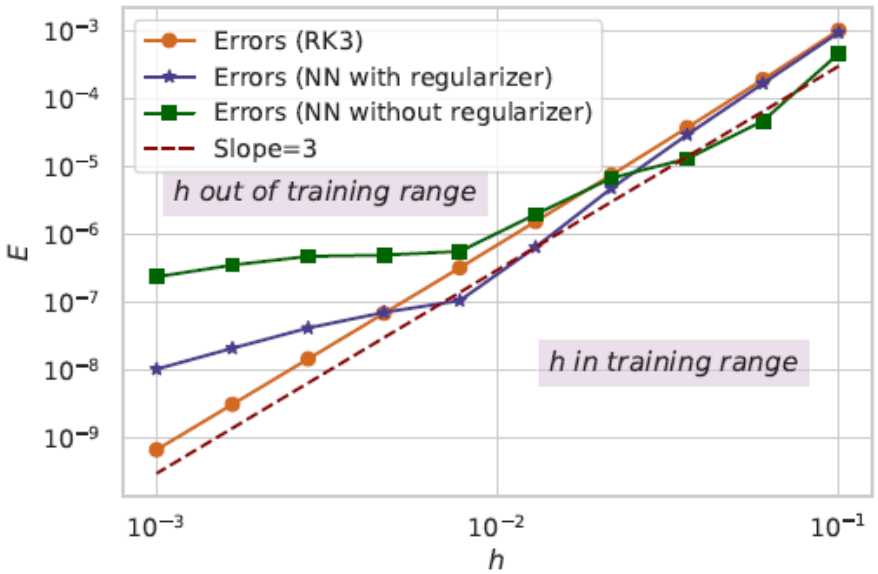
Square Task Family: $\mathbf{f}$ is a scaled element-wise square function $f(\mathbf{y})_i = -ay_i^2$ and $\mathbf{y}_0 \sim U(1,3)$, thus $\mathcal{F}$ has the form

$$\begin{aligned} \mathcal{F} &= \{\mathbf{y} \mapsto -a\mathbf{y}^2 \mid a > 0\} \times \{\mathbb{R}\}, \\ \mu &= \text{Distribution}(\{\mathbf{y} \mapsto -a\mathbf{y}^2; a \sim U(0.1,0.5)\}) \times U(1,3). \end{aligned} \tag{20}$$

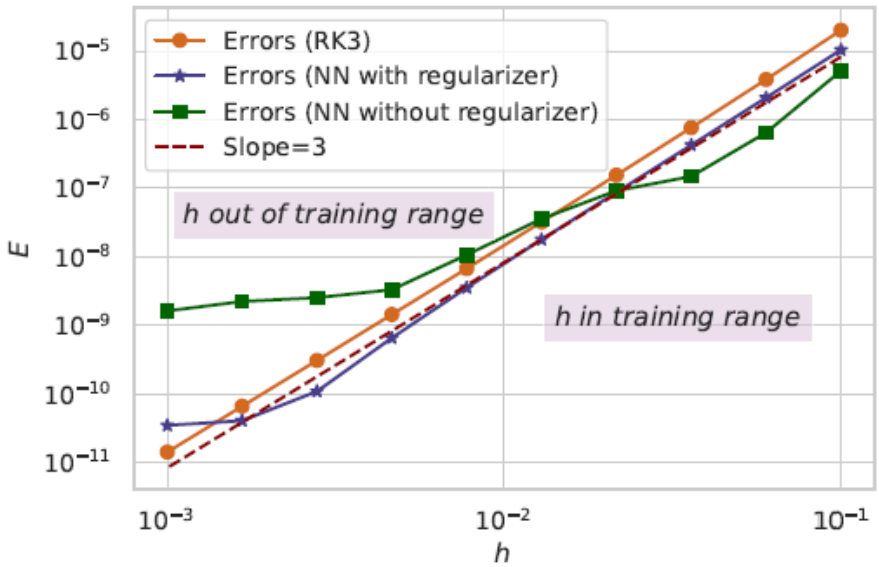
The true solution is $\boldsymbol{y}(t) = (at + 1/\boldsymbol{y}_0)^{-1}$.

The integration time step h used for training is sampled uniformly in $(0.01, 0.1)$.

Learning ODE integrators



(a) Linear Task Families.



(b) Square Task Families.

Learning ODE integrators

Training a two-stage RK-NN integrator to have third-order accuracy. Here, we set $m = 2$ (two-stage RK-NN), but we set $\alpha = 3$ in the regularizer, which promotes a third-order accuracy. The results are shown below for the nonlinear (square) task family.

Learning ODE integrators

Recall the two-stage RK method:

$$\mathbf{k}_1 = hf(\boldsymbol{y}_n), \quad \mathbf{k}_2 = hf(\boldsymbol{y_n} + \theta_1 \mathbf{k}_1), \quad \boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \theta_{c1}\mathbf{k}_1 + \theta_{c2}\mathbf{k}_2, \quad (21)$$

For the square task family with $d = 1$, the equation is

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = f(y) = -ay^2, \qquad y(0) = y_0 \in \mathbb{R}. \quad (22)$$

Then we can obtain

$$y_{n+1} = y_n - (\theta_{c1} + \theta_{c2})\, a y_n^2 h + 2\theta_1 \theta_{c2} a^2 y_n^3 h^2 - \theta_1^2 \theta_{c2} a^3 y_n^4 h^3. \quad (23)$$

Due to Taylor's theorem,

$$\widetilde{y}(t_{n+1}) = y_n - a y_n^2 h + a^2 y_n^3 h^2 - a^3 y_n^4 h^3 + \mathcal{O}(h^4). \quad (24)$$

Indeed, the coefficients in the learned RK-NN are

$$\theta_1 = 2, \quad \theta_{c1} = 0.75, \quad \theta_{c2} = 0.25, \quad (25)$$

Learning ODE integrators

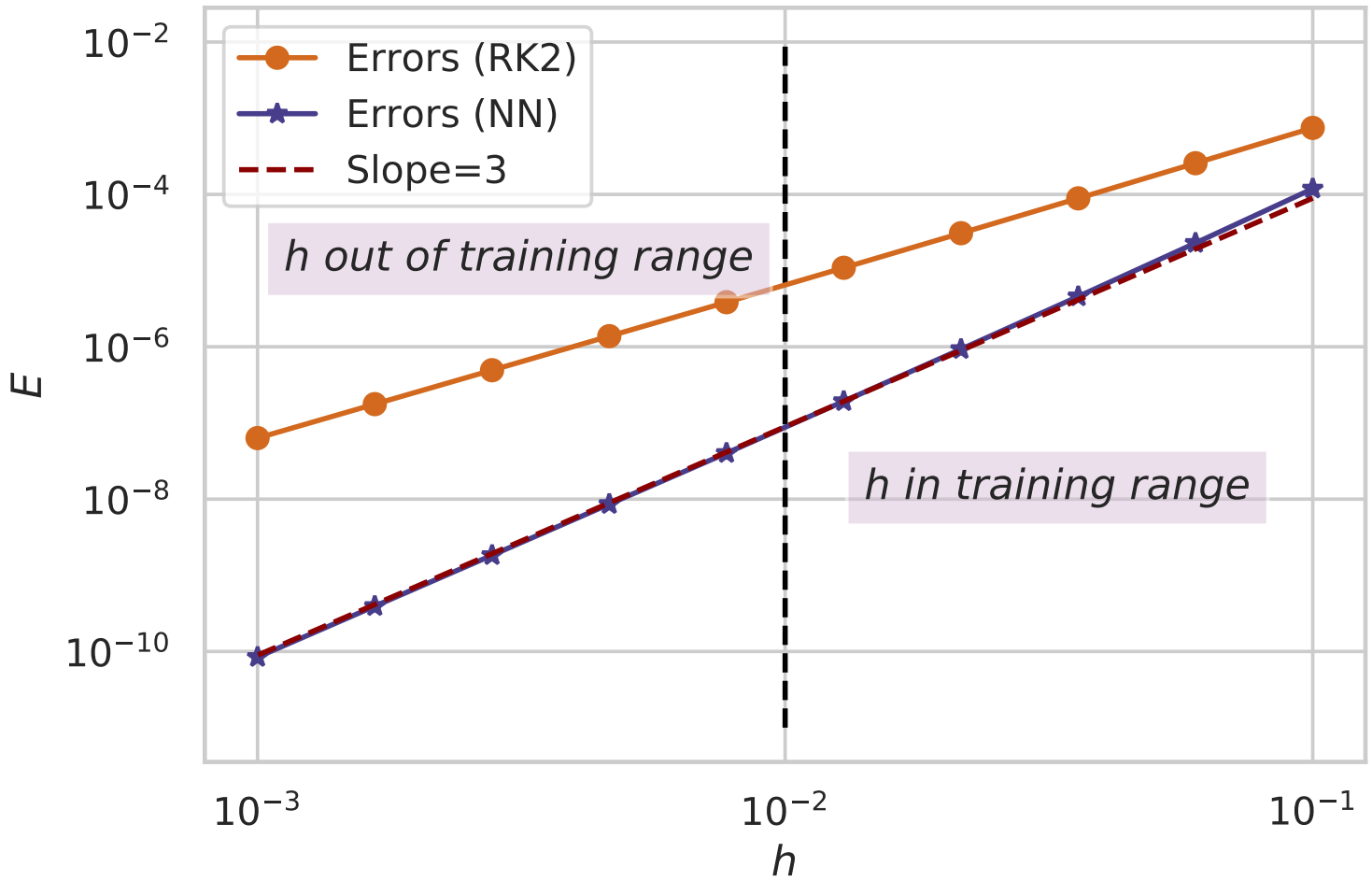


Figure: Error analysis on **square task families**, training on $h \in (0.01, 0.1)$ but testing on $h \in (0.001, 0.1)$, using **two-stage** RK-NN integrator with **third-order** Taylor-based loss as the regularizer.

Reference

Y. Guo, F. Dietrich, T. Bertalan, D. T. Doncevic, M. Dahmen, I. G. Kevrekidis, and Q. Li. Personalized algorithm generation: A case study in learning ODE integrators. SIAM Journal on Scientific Computing, 44(4):A1911–A1933 (2022).

Thanks!