Personalized Algorithm Generation: A Case Study in Meta-Learning ODE Integrators

Yue Guo* Felix Dietrich† Tom Bertalan‡
Danimir T. Doncevic § Manuel Dahmen¶ Ioannis G. Kevrekidis‖
Qianxiao Li**

Abstract

We study the meta-learning of numerical algorithms for scientific computing, which combines the mathematically driven, handcrafted design of general algorithm structure with a data-driven adaptation to specific classes of tasks. This represents a departure from the classical approaches in numerical analysis, which typically do not feature such learning-based adaptations. As a case study, we develop a machine learning approach that automatically learns effective solvers for initial value problems in the form of ordinary differential equations (ODEs), based on the Runge-Kutta (RK) integrator architecture. By combining neural network approximations and meta-learning, we show that we can obtain high-order integrators for targeted families of differential equations without the need for computing integrator coefficients by hand. Moreover, we demonstrate that in certain cases we can obtain superior performance to classical RK methods. This can be attributed to certain properties of the ODE families being identified and exploited by the approach. Overall, this work demonstrates an effective, learning-based approach to the design of algorithms for the numerical solution of differential equations, an approach that can be readily extended to other numerical tasks.

*Department of Mathematics, National University of Singapore, 117543, Singapore.

†Institut für Informatik, TU München, Boltzmannstr. 3, 85748 Garching b. München, Germany.

‡Department of Chemical and Biomolecular Engineering, Whiting School of Engineering, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA.

§Institute of Energy and Climate Research – Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, 52425 Jülich, Germany; RWTH Aachen University, Aachen 52062, Germany.

¶Institute of Energy and Climate Research – Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, 52425 Jülich, Germany.

‖Department of Chemical and Biomolecular Engineering and Department of Applied Mathematics and Statistics, Whiting School of Engineering, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA.

Department of Mathematics, National University of Singapore, 117543, Singapore; Institute of High Performance Computing, A*STAR, 138632, Singapore. **Corresponding author: qianxiao@nus.edu.sg.

1 Introduction

In computational mathematics, one is interested in developing solvers for different types of problems, such as algebraic equations, differential equations or optimization problems. In an abstract setting, these can be written as

$$L(y, F) = 0, \tag{1}$$

where $y \in \mathcal{Y}$ is the unknown, $F \in \mathcal{F}$ is the problem instance and $L : \mathcal{Y} \times \mathcal{F} \to \mathcal{X}$ is a mapping representing the problem type. The sets $\mathcal{Y}, \mathcal{F}, \mathcal{X}$ are usually some subsets of normed vector spaces and can be finite or infinite dimensional, depending on the application.

Here are some examples:

- An algebraic equation $f(y) = 0$ can be recast as

  $$L(y, F) = f(y) = 0,$$

  where $F = f$.

- An optimization problem $\min_y f(y)$ can be recast as

  $$L(y, F) = f(y) - \min_z f(z) = 0,$$

  where $F = f$.

- 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.

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.

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

$$\sup_{F\in\mathcal{F}} \|L(\hat{A}(F, h), F)\| = \mathcal{O}(h^\alpha), \qquad \alpha > 0. \tag{2}$$

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 $\mathcal{F}$ (e.g. only integrate symplectic ODEs), and we may only be interested in the average performance of our method on this class of problems. Hence, instead of (2), we may require

$$E_\mu[\|L(\hat{A}(F,h),F)\|] = \int_{F\in\mathcal{F}} \|L(\hat{A}(F,h),F)\| d\mu(F) = \mathcal{O}(h^\alpha), \qquad \alpha > 0, \tag{3}$$

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. In other words, we want to find an approximate solver that is adapted to a restricted problem class.

One can think of this as "personalized" algorithm: one tuned to the class of problems, over the range of parameters of interest. This goes beyond algorithm *parameter* tuning, to possibly include algorithm structure design and also combinations of multiple algorithms. Success over large and unstructured classes or problems would, of course, allow the algorithm (and the associated code) to be portable across many physical models, and has long been an obvious advantage for scientific computation - both for learning how to perform it, and for performing it. For specific applications there are almost always some tuning involved: integrators for stiff vs. nonstiff problems; symplectic integrators for Hamiltonian vs. "general" nonsymplectic ones, lower vs. higher order optimization algorithms, etc. Yet, tuning the algorithm to the specific problem was left to the practitioner interested in the specific problem: Which algorithm? What order? What accuracy? How frequent the adaptation? - and has been mostly done "by hand". This is not surprising, since incorporating complex and varied structures into algorithms requires a detailed understanding of the structure of the problem at hand, and often has to be treated on a case by case basis. However, machine learning allows us to contemplate delegating this task to the computer: we need to first choose the class of problems of interest, and then devise sufficiently general superstructures (in this case, superstructures for NN architectures), that will perform the personalized tuning. For example, we can parameterize the approximate solver as a neural network $\hat{A}(\cdot,\cdot;\boldsymbol{\theta})$ where $\boldsymbol{\theta} \in \Theta$ is a vector of fitting parameters, that can be determined from training using appropriate error metrics over a range of tasks of interest. The parametrization $\hat{A}(\cdot,\cdot;\boldsymbol{\theta})$ represents a meta-architecture, where $\boldsymbol{\theta}$ specifies how the algorithm can operate on the task F to produce an approximate solution. The optimal way that the task F is used to produce the solution will depend on the structure of the problem induced by μ, and machine learning can help us find an approximately optimal way to do so. This is also a form of *meta-learning*, since we want a solver that performs well on not just one *task* F, but on a distribution of tasks [26]. This naturally leads to the use of optimization (e.g., a Hamilton-Jacobi-Bellman approach) for the generation of optimal algorithms (See [56], [55] and [35]). Such a use of optimization over superstructures for optimal algorithm generation has been recently proposed and illustrated in [40]; we will return to this in Section 6.

In this paper, we will investigate a particular realization of this general problem by studying ODE integrators, which are approximate solvers for some initial value problems. In particular, we develop a meta-learning type of algorithm to generate effective and specialized integrators adapted to specified problem settings.

The rest of the paper is organized as follows. In Section 2, we formulate the precise problem of novel integrators via meta-learning. Next, in Section 3 we introduce our meta-architecture to achieve this, based on the Runge-Kutta family of integrators, and our learning algorithm based on novel losses derived from Taylor expansions. In Section 4, we demonstrate the effectiveness of our learned integrators on selected benchmark function families and provide some analysis to understand the origin of the improvement over classical methods. We conclude with discussions on related work in Section 5, together with some general observations and future directions in Section 6.

2 Problem Formulation for Case Study

We focus on a particular realization of the general problem we discussed before: learning high-accuracy integrators adapted to integrating specific families of ordinary differential equations (ODEs). We begin by introducing the background and basic notions of ODE integrators, with particular emphasis on the Runge-Kutta family of explicit integrators, which form the basis of our neural network parameterization. We conclude this section with the precise mathematical formulation of our integrator learning problem.

2.1 Ordinary Differential Equations and Integrators

Consider a time-homogeneous ordinary differential equation describing an initial value problem in $\mathbb{R}^d$

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{y}(t)), \qquad \boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^d, \qquad \boldsymbol{y}(0) = \boldsymbol{y}_0 \in \mathbb{R}^d. \tag{4}$$

Here, $\boldsymbol{f}$ is a vector field driving the evolution equation. Instead of one specific $\boldsymbol{f}$, we will consider a family H of vector fields. For simplicity, we will assume that H contains only Lipschitz-continuous functions, so that eq. (4) admits a unique solution. Note that eq. (4) includes as a special case time-inhomogeneous equations $\mathrm{d}\boldsymbol{y}/\mathrm{d}t = \boldsymbol{f}(t, \boldsymbol{y})$, since we may always define an additional variable $\tau(t)$ such that $\mathrm{d}\tau/\mathrm{d}t = 1$ and redefine $\tilde{\boldsymbol{y}} = (\tau, \boldsymbol{y})$ and $\tilde{\boldsymbol{f}}(\tilde{\boldsymbol{y}}) = [1, \boldsymbol{f}(\tau, \boldsymbol{y})]$. The only caveat is that this redefinition requires $\boldsymbol{f}$ to be Lipschitz in t, whereas for general ODE theory this condition can be relaxed [28]. Nevertheless, for numerical computation such a technical issue is less important, and thus we will hereafter only consider the time-homogeneous case without loss of generality.

For general $\boldsymbol{f}$, eq. (4) does not admit an explicit closed-form solution, and one often resorts to a numerical approximation via a solver. Let $F = (\boldsymbol{f}, \boldsymbol{y}_0)$ define a problem

instance and the algorithm parameter h represents a desired level of precision of the solution, then an integrator builds an approximate solution iteratively. In the simplest case of explicit, one-step integrators, one iterates the following formula based on an integrator $I_{\hat{A}}$ that computes

$$\hat{\boldsymbol{y}}_{n+1} = I_{\hat{A}}(\boldsymbol{f}, \hat{\boldsymbol{y}}_n, h), \qquad \hat{\boldsymbol{y}}_0 = \boldsymbol{y}_0. \tag{5}$$

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

The accuracy of the integrator is measured by the *local* and *global* truncation errors. We write the solution of eq. (4) with $t = nh$ as $\boldsymbol{y}_n := y(nh)$. The local truncation error is defined as the one-step error between the integrator and the true solution, i.e.

$$E_{h,1}(\boldsymbol{y}_0) = \|\hat{\boldsymbol{y}}_1 - \boldsymbol{y}_1\|. \tag{6}$$

The integrator is called *consistent* if $E_1(\boldsymbol{y}_0) = o(h)$ for each $\boldsymbol{y}_0 \in \mathbb{R}^d$. On the other hand, the global truncation error is

$$E_{h,n}(\boldsymbol{y}_0) = \|\hat{\boldsymbol{y}}_n - \boldsymbol{y}_n\|. \tag{7}$$

The integrator is said to be *convergent* if $\lim_{h\to 0} \max_{m\leq n} E_{h,m}(\boldsymbol{y}_0) = 0$.

One has finer measures of performance in terms of the *order of convergence*. In particular, we say that a convergent integrator is of global order $p > 0$ if

$$\max_{m\leq n} E_{h,m}(\boldsymbol{y}_0) = \mathcal{O}(h^p). \tag{8}$$

2.2 Explicit Runge-Kutta Integrators

We use the prediction from the RK method with targeted order to ensure the order of our method. In general, our approach is not restricted to the explicit RK family, and other integration methods like multi-stage DAE solvers, etc. are also possible. Now, let us introduce the family of integrators known as explicit Runge-Kutta integrators (ERK) [28]. While these methods are well known, we give a brief account here in order to motivate subsequent developments in our learning-based approach, which depends on the structures of RK integrators.

Let us write the solution of the ODE eq. (4) as

$$\boldsymbol{y}\left(t_{n+1}\right) = \boldsymbol{y}\left(t_n\right) + \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{y}(\tau))\mathrm{d}\tau = \boldsymbol{y}\left(t_n\right) + h\int_0^1 \boldsymbol{f}\left(\boldsymbol{y}\left(t_n + h\tau\right)\right)\mathrm{d}\tau. \tag{9}$$

Then, an approximate solution can be found by applying quadrature to the last integral

$$\boldsymbol{y}_{n+1} \approx \boldsymbol{y}_n + h\sum_{i=1}^{m} b_i \boldsymbol{f}\left(\boldsymbol{y}\left(t_n + c_i h\right)\right), \quad n = 0, 1, \ldots \tag{10}$$

It remains to approximate $\boldsymbol{y}(t_n + c_i h)$ by vectors $\boldsymbol{\xi}_i, i = 1, 2, \ldots, m$. We set $c_1 = 0$, then $\boldsymbol{\xi}_1 = \boldsymbol{y}_n$. The idea behind *explicit Runge-Kutta (ERK)* methods is to express each $\boldsymbol{\xi}_i, i = 2, 3, \ldots, m$, by updating $\boldsymbol{y}_n$ with a linear combination of $\boldsymbol{f}(\boldsymbol{\xi}_1), \ldots, \boldsymbol{f}(\boldsymbol{\xi}_{i-1})$. This leads to the integrator

$$\begin{aligned} \boldsymbol{\xi}_m &= \boldsymbol{y}_n + h \sum_{j=1}^{m-1} a_{m,j} \boldsymbol{f}(\boldsymbol{\xi}_j), \\ \hat{\boldsymbol{y}}_{n+1} &= \boldsymbol{y}_n + h \sum_{i=1}^{m} b_i \boldsymbol{f}(\boldsymbol{\xi}_i). \end{aligned} \tag{11}$$

Appropriate choices of the coefficients $\{a_{i,j}, b_i\}$ then ensures that our approximation is accurate to the desired level. To determine these coefficients, we expand and equate the Taylor series of $\boldsymbol{y}_{n+1}$ with that of $\hat{\boldsymbol{y}}_{n+1}$ about $\boldsymbol{y}_n$. Note that the conditions do not define an ERK integrator uniquely, and any choices of the coefficients, from which the same order accuracy can be obtained, are considered as ERK methods.

2.3 Learning Integrators

Because a large number of equations need to be solved to obtain the coefficients, deriving high-order ERK integrators is nontrivial. Moreover, the order of convergence p is forced upon all $(p+1)$-times continuously differentiable functions, which may be a much larger family than what one might be interested in integrating in practice. Here, we explore the following: Can one obtain better integrators adapted to a smaller, structured family of problems $\mathcal{F}$?

To this end, we parameterize a family of approximate solvers

$$\mathcal{A}(\Theta) := \{\hat{A}(\cdot, \cdot; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \Theta\}, \tag{12}$$

where Θ is some subset of a Euclidean space representing the fitting parameters (trainable weights).

Let μ be a probability measure on $\mathcal{F}$, representing a particular distribution of tasks F. 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

$$\begin{aligned} \min_{\boldsymbol{\theta} \in \Theta} \quad & \mathbb{E}_{F \sim \mu, h \sim \nu} \left[\mathcal{L}(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n) + \mathcal{R}(\hat{A}(\cdot, \cdot; \boldsymbol{\theta}), F, h) \right] \\ s.t. \quad & \boldsymbol{y}_n = A(F)(nh) = \boldsymbol{y}_0 + \int_0^{nh} \boldsymbol{f}(\boldsymbol{y}(s)) ds, \\ & \hat{\boldsymbol{y}}_n = \hat{A}(F, h; \boldsymbol{\theta})(nh) = I_{\hat{A}}(\boldsymbol{f}, \hat{\boldsymbol{y}}_{n-1}, h; \boldsymbol{\theta}), \\ & F = (\boldsymbol{f}, \boldsymbol{y}_0), \\ & n \geq 0. \end{aligned} \tag{13}$$

Here we define a formula in the last term $\mathcal{R}(\hat{A}(\cdot,\cdot;\boldsymbol{\theta}), F, h)$, which is independent of the choice for h. We just consider the situation near $t = 0$, only depending on the structure of the task F. This represents a regularization term that allows us to promote certain order of accuracy, and we shall discuss its importance in section 3.2.

Problem eq. (13) is the central formulation of this paper, where we rephrase the problem of finding an effective integrator as an optimization problem. Note that this formulation takes explicit account of the fact that the problem instance $F = (\boldsymbol{f}, \boldsymbol{y}_0)$ is not generic, but rather belongs to a potentially structured function class $\mathcal{F}$ endowed with a probability measure μ, representing the distribution of tasks. Moreover, note that we also consider a measure over the step sizes $h \sim \nu$, indicating the fact that we are not always looking for integrators that work equally well for all step sizes. Finally, we note that eq. (13) is a population risk minimization problem in the language of machine learning, and hence to solve it we often need to replace the respective expectations by averages over samples from the respective probability measures. In the next section, we will discuss the parameterization of RK-like integrators using neural networks and the choice of loss functions and regularizers that enables one to solve eq. (13) to yield novel integrators.

3 Model Architecture and Choice of Loss Functions

In this section, we outline our method for solving eq. (13). We begin with the parameterization of the family of solvers $\mathcal{A}(\Theta)$ using neural networks, after which we introduce the crucial choice of loss functions and regularizers which enable us to learn accurate integrators.

3.1 NN Architectures Parameterizing RK-like Integrators

Recall that the Runge-Kutta (RK) family of integrators form a sequential linear combination of function evaluations to build the integrator via approximate quadrature. Here, we can build a neural network that parameterizes the general form of RK-type integrators. The integrator is constructed as $\boldsymbol{k}_1 = \boldsymbol{f}\left(\mathrm{id}(\hat{\boldsymbol{y}}_n)\right)$ where id is the identity function, $\boldsymbol{k}_i = h\boldsymbol{f}\left(\hat{\boldsymbol{y}}_n + \sum_{j=1}^{i-1} \theta_{i-1,j}\boldsymbol{k}_j\right)$ for $i = 2, \ldots, m$, and $\hat{\boldsymbol{y}}_{n+1} = \hat{\boldsymbol{y}}_n + \sum_{i=1}^{m} \theta_{ci}\boldsymbol{k}_i$. The neural network architecture based on the above formulation is given in the left side of fig. 1.

There are two components in the RK-like neural network (RK-NN):

1. Submodels $N_i, i = 1, 2, \ldots, m-1$, provide inputs to the function $\boldsymbol{f}$, given $\hat{\boldsymbol{y}}_t$, $\boldsymbol{k}$ from the last step, time step h and parameters $\boldsymbol{\theta}_i = (\theta_{i,1}, \theta_{i,2}, \ldots, \theta_{i,i})$.

2. Combined model N_c has output as $\hat{\boldsymbol{y}}_{n+1}$, given $\hat{\boldsymbol{y}}_n$, all the variables $\boldsymbol{k}_m$, time step h and parameters $\boldsymbol{\theta}_c = (\theta_{c1}, \theta_{c2}, \ldots, \theta_{cm})$.

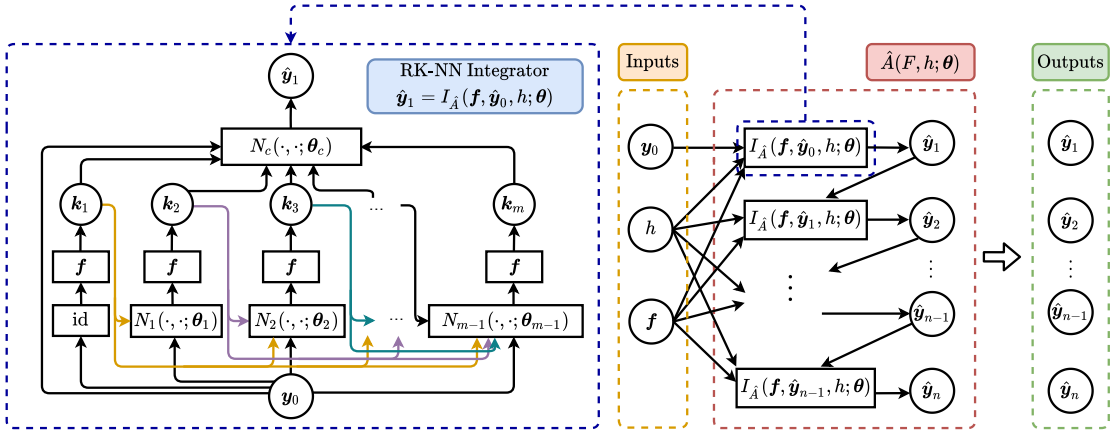


Figure 1: RK-like Neural Network (RK-NN) Architecture.

Each of these components are represented by a linear layer. If we set $m = 3$, the architecture represents a generalized RK3 integrator. Recall the similar architecture of RK-m method, where each $N_i(\cdot,\cdot;\boldsymbol{\theta}_i)$ is constructed by $\hat{\boldsymbol{y}}_n + \sum_{j=1}^{i} \theta_{i,j}\boldsymbol{k}_j$. $\theta_{i,j} \in \mathbb{R}$, and $\hat{\boldsymbol{y}}_n$, $\boldsymbol{k}_j$ are d-dimensional vectors. The trainable parameters are the coefficients before each $\boldsymbol{k}_j$. Despite the linear combination of $\boldsymbol{k}_j$'s in each layer, the unfolded structure contains nonlinearities induced by $\boldsymbol{f}$. The transformation in the last layer N_c of the whole RK network architecture has the form $\hat{\boldsymbol{y}}_n + \sum_{i=1}^{m} \theta_{ci}\boldsymbol{k}_i$. To make sure that the sum of θ_{ci} is equal to 1, we apply a softmax activation $\theta_{ci} = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}}$ for $i = 1, \ldots, m$, where $z_i \in \mathbb{R}$ are the trainable variables in the final layer. We use softmax to allow for direct application of unconstrained optimization methods. The downside is that we always have positive values for the coefficients, which cannot be exactly 0 or 1. However, the softmax activation can approach 0 or 1 quickly, and we found in practice that this did not pose a problem. The total number of trainable parameters in RK-NN is $m + \sum_{i=1}^{m-1} i = \frac{m(m+1)}{2}$.

An important point in the choice of architectures is that we want the integrators to be consistent, i.e., the local truncation error should vanish as $h \to 0$. This is ensured by our parametric construction, as shown below.

Proposition 3.1 *For any $\boldsymbol{\theta} \in \Theta$, the NN parameterization in RK-NN (shown in fig. 1) is consistent.*

Proof 3.1 *By the definition of explicit RK method and the Taylor expansion of $\boldsymbol{f}$ at $\hat{\boldsymbol{y}}_n$, we have $\boldsymbol{k}_1 = h\boldsymbol{f}(\hat{\boldsymbol{y}}_n)$ and $\boldsymbol{k}_i = h\boldsymbol{f}(\hat{\boldsymbol{y}}_n + \sum_{j=1}^{i-1} \theta_{i-1,j}\boldsymbol{k}_j) = h\boldsymbol{f}(\hat{\boldsymbol{y}}_n) + o(h)$ for $i = 2, \ldots, m$. Then the predicted value is $\hat{\boldsymbol{y}}_{n+1} = \hat{\boldsymbol{y}}_n + \sum_{i=1}^{m} \theta_{ci}\boldsymbol{k}_i = \hat{\boldsymbol{y}}_n + \sum_{i=1}^{m} \theta_{ci} h\boldsymbol{f}(\hat{\boldsymbol{y}}_n) + o(h)$. The sum of θ_{ci} is equal to 1 since $\theta_{ci} = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}}$. Thus, we obtain $\hat{\boldsymbol{y}}_{n+1} = \hat{\boldsymbol{y}}_n + h\boldsymbol{f}(\hat{\boldsymbol{y}}_n) + o(h)$. The taylor expansion of the true solution is $\tilde{\boldsymbol{y}}(t_{n+1}) = \hat{\boldsymbol{y}}_n + h\boldsymbol{f}(\hat{\boldsymbol{y}}_n) + o(h)$. We observe that coefficients before the first order of h are the same, then $E_1(\hat{\boldsymbol{y}}_n) = \|\hat{\boldsymbol{y}}_{n+1} - \tilde{\boldsymbol{y}}(t_{n+1})\| = o(h)$, which shows this integrator is consistent.*

3.2 Choice of Loss Function and Regularizer

To solve eq. (13), we need to define the loss function $\mathcal{L}$ and the regularizer $\mathcal{R}$.

Loss function We make the simple choice of a scaled square loss

$$\mathcal{L}(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n) = \frac{\|\boldsymbol{y}_n - \hat{\boldsymbol{y}}_n\|^2}{\|\boldsymbol{y}_n - \hat{\boldsymbol{y}}_n^{(RK)}\|^2}, \tag{14}$$

where $F = (\boldsymbol{f}, \boldsymbol{y}_0)$, $\boldsymbol{y}_n = A(F)(nh)$, $\hat{\boldsymbol{y}}_n = \hat{A}(F, h; \boldsymbol{\theta})(nh) = I_{\hat{A}}(\boldsymbol{f}, \hat{\boldsymbol{y}}_{n-1}, h; \boldsymbol{\theta})$ and $\hat{\boldsymbol{y}}_n^{(RK)} = \hat{A}_{RK}(F, h)(nh) = I_{\hat{A}_{RK}}(\boldsymbol{f}, \hat{\boldsymbol{y}}_{n-1}^{(RK)}, h)$. $\hat{\boldsymbol{y}}$ is the prediction from our RK-NN integrator and $\hat{\boldsymbol{y}}^{(RK)}$ is from the RK method. Here, we consider one-step prediction by setting $n = 1$. We use the difference between the RK prediction and the true solution as a scale stabilize numerics, since the errors tend to be small for small h. If we expect our RK-NN method to be trained to a specific order α, RK-α is chosen in eq. (14) correspondingly.

In the case where the true solution $\boldsymbol{y}_1 \equiv \boldsymbol{y}(h)$ is not known, we can compute its Taylor expansion near $h = 0$ up to appropriate order and use it as a surrogate. Note that computing the Taylor expansion only requires $\boldsymbol{f}$:

$$\boldsymbol{y}(h) = \boldsymbol{y}(0) + \sum_{m=1}^{n} \frac{1}{m!} \frac{\mathrm{d}^m \boldsymbol{y}(0)}{\mathrm{d}h^m} h^m + \mathcal{O}\left(h^{n+1}\right), \tag{15a}$$

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}h} = \boldsymbol{f}(\boldsymbol{y}), \tag{15b}$$

$$\frac{\mathrm{d}^m \boldsymbol{y}}{\mathrm{d}h^m} = \left(\frac{\mathrm{d}^m y_1}{\mathrm{d}h^m}, \ldots, \frac{\mathrm{d}^m y_d}{\mathrm{d}h^m}\right)^T, \quad \frac{\mathrm{d}^m y_i}{\mathrm{d}h^m} = \sum_{j=1}^{d} \frac{\partial \left(\frac{\mathrm{d}^{m-1} y_i}{\mathrm{d}h^{m-1}}\right)}{\partial y_j} f_j, \tag{15c}$$

The appropriate order of the computed Taylor expansion depends on the desired integrator accuracy. For example, to obtain a third-order integrator, $n \geq 3$ in eq. (15a) is chosen as a surrogate of the true solution and we choose RK3 as the reference algorithm.

Regularizer The loss function alone cannot ensure that we can achieve a desired order of accuracy, since the mean squared loss has vastly different contributions from different values of h. To overcome this issue, we introduce a regularizer that promotes high order of convergence of the global truncation error over a span of integration step sizes. Recall that to obtain an integrator with $\mathcal{O}(h^\alpha)$ global error, we need the local truncation error to be $\mathcal{O}(h^{\alpha+1})$. This is achieved by ensuring $\frac{d^i}{dh^i}\big|_{h=0} (\boldsymbol{y}_1 - \hat{\boldsymbol{y}}_1) = 0, \forall i = 1, \ldots, \alpha$, or equivalently, $\sum_{i=1}^{\alpha} \| \frac{d^i}{dh^i}|_{h=0} (\boldsymbol{y}_1 - \hat{\boldsymbol{y}}_1) \|_2^2 = 0$. The latter is scalar-valued, thus convenient to turn into a regularizer

$$\mathcal{R}(\hat{A}(\cdot, \cdot; \boldsymbol{\theta}), F, h) = \sum_{i=1}^{\alpha} \left\| \frac{d^i}{dh^i}\bigg|_{h=0} (\boldsymbol{y}_1 - \hat{\boldsymbol{y}}_1) \right\|_2^2, \tag{16}$$

which promotes the desired order of convergence.

In implementation, the derivatives can be evaluated at exactly $h = 0$ through automatic differentiation in Tensorflow. In the case where the true solution is not known and we use the Taylor expansion surrogate eq. (15a), automatic differentiation can still be applied at $h = 0$ to the surrogate.

3.3 Learning algorithm

Having defined the loss functions and regularizers, it remains to train the network using standard stochastic gradient methods, with sample means to approximate the expectations in eq. (13). The performance of the RK-NN integrator is quantified by the relative error γ compared with the reference RK method, $\gamma < 1$ implies an improvement. The entire learning algorithm is summarized in Alg. 1. The code is open source and can be found at `https://github.com/GUOYUE-Cynthia/Meta-Learning-ODE-Integrators`.

Algorithm 1: Learning Algorithm.

Data: $\mathcal{D} = \{F_j, h_j\}_{j=1}^N$;
Initialize: Random $\boldsymbol{\theta}_0$ for operator $\hat{A}(\cdot,\cdot;\boldsymbol{\theta}_0) : \boldsymbol{\theta}_0 \in \Theta, h > 0$;
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
  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
end
return Operator $\hat{A}(\cdot,\cdot;\boldsymbol{\theta})$.

Remark 1 (Computational complexity and memory usage) *In terms of inference (i.e. integrating a new ODE using the trained RK-NN), computational complexity and memory usage are the same as the traditional RK method. This is because the difference between RK-NN and traditional RK methods lies only in the value of the learned*

coefficients. During training, the memory cost of storing weights is independent of the dimension of the ODE, because the number of parameters in RK-NN is determined by the number of RK stages. The computational complexity of training is more delicate, and depends on a variety of factors, including the form of the ODE family, the software implementation of back-propagation, and hardware optimization. Instead of theoretical estimates, we show empirically (See supplementary material, ***SM5****) that training RK-NN integrator on an example linear family and a nonlinear family has favorable scaling between $O(d)$ and $O(d^2)$ as the number of dimensions d increases. Thus, it can be applied to moderately sized integration problems.*

4 Results

We now present the results of RK-NN on various test problems, with particular emphasis on how the learned integrators differ fundamentally from the classical RK methods. In all experiments, we use the Adam optimizer [31] to train the RK-NN. To check the accuracy order, we use the error defined in eq. (7), hereafter abbreviated as E, as the evaluation standard to quantify the global error. Note that the order of accuracy from different integrators can be inferred from the slopes of the log-scale plots.

4.1 Learning High-order Integrators

We first demonstrate that the learning algorithm indeed learns high-order integrators for several test problem settings. In all experiments, the integration time step h used for training is sampled uniformly in $(0.01, 0.1)$.

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

$$\begin{aligned} \mathcal{F} &= \{\boldsymbol{y} \mapsto -a\boldsymbol{y} \mid a > 0\} \times \{\mathbb{R}\}, \\ \mu &= \text{Distribution}(\{\boldsymbol{y} \mapsto -a\boldsymbol{y}; a \sim U(1,5)\}) \times U(-5,5). \end{aligned} \tag{17}$$

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

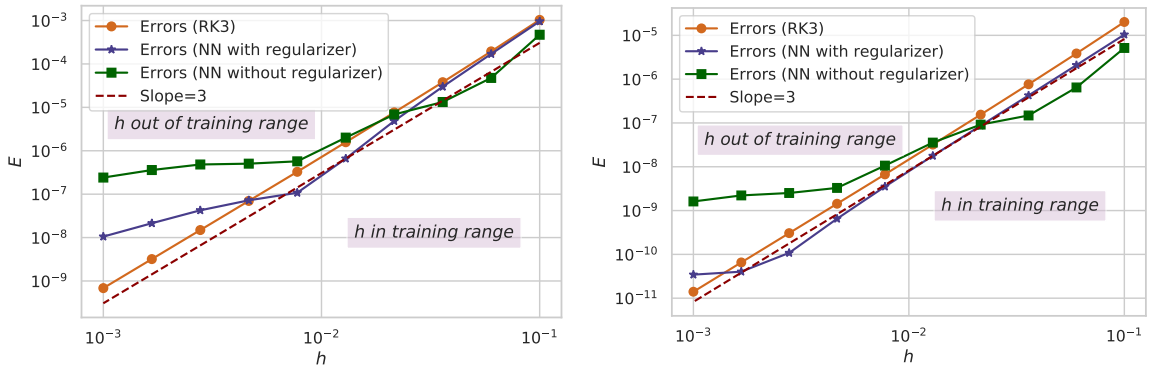
Square Task Family $\boldsymbol{f}$ is a scaled element-wise square function $f(\boldsymbol{y})_i = -ay_i^2$ and $\boldsymbol{y}_0 \sim U(1,3)$, thus $\mathcal{F}$ has the form

$$\begin{aligned} \mathcal{F} &= \{\boldsymbol{y} \mapsto -a\boldsymbol{y}^2 \mid a > 0\} \times \{\mathbb{R}\}, \\ \mu &= \text{Distribution}(\{\boldsymbol{y} \mapsto -a\boldsymbol{y}^2; a \sim U(0.1,0.5)\}) \times U(1,3). \end{aligned} \tag{18}$$

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

Figure 2 compares the performance of the learned integrator to that of RK3. For the sake of comparison, we will set $m = 3$ in our RK-like neural network (see fig. 1).

Moreover, we set $\alpha = 3$ in the regularizer to promote a similar global truncation order as RK3 integrator. We observe from fig. 2 that we can indeed learn integrators that out-performs the RK3 method when h is in the training range, but becomes worse when extending to a larger range of h. This is significant since we did not need to compute the Butcher Tableau explicitly, and the coefficients of the RK-NN are chosen automatically via machine learning. This makes the method easily scalable to higher order methods, unlike explicitly derived RK integrators which can become very complex. It is also evident that the learned RK-NN is different from a usual RK3 method computed from the Butcher Tableau. Furthermore, to validate the importance of the regularizer defined in eq. (16), we train RK-NN without the regularizer and compare its performance in fig. 2. We observe that in this case, the global accuracy depends more sensitively on h, and deteriorates rapidly outside the range of h used for training.



(a) Linear Task Families.

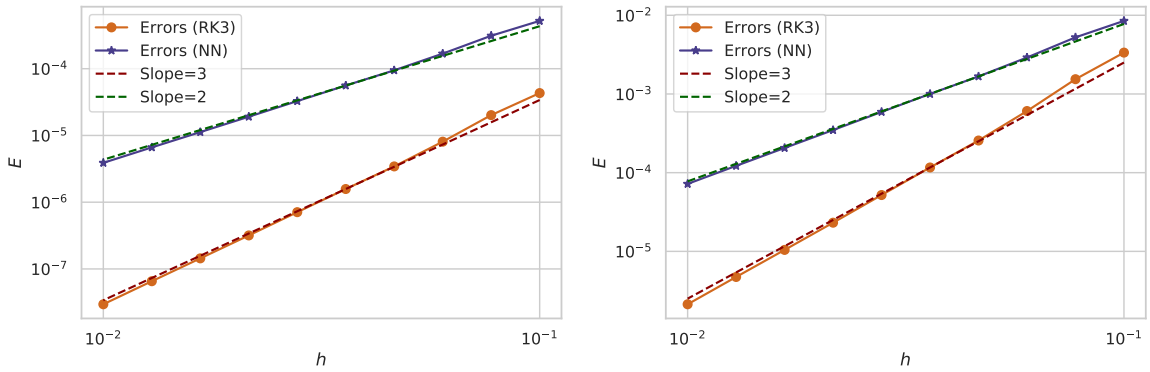
(b) Square Task Families.

Figure 2: Error comparison among RK3 method, **three-stage** RK-NN with and without **third-order** Taylor-based loss as regularizer. The training time step range is $h \in (0.01, 0.1)$ whereas the testing range extends to $h \in (0.001, 0.1)$.

4.2 Generalization Across Task Families

From the learning formulation, it is clear that our goal is to maximize the performance of our integrator on a specific task family $\mathcal{F}$. Hence, it is expected that the effectiveness of the learned integrators may not generalize across different families. Figure 3 shows that this is indeed the case. This is expected, since the learned integrator is adapted to the family of integration tasks that it is trained on. Nevertheless, note that in all cases the learned integrator maintains consistency, which is ensured by the construction of the parametric NN family (See proposition 3.1). *This shows in particular that we have not learned a generic RK3 integrator.*

Although the limited generalization ability may appear to be a limitation of the approach, it turns out that this enables us to obtain very efficient integrators that outperforms classical RK integrators on such restricted families. The next part discusses



(a) Train on linear task families but test on square task families.

(b) Train on square task families but test on linear task families.

Figure 3: Train and test on distinct task families. Time steps in this evaluation are in the training range $(0.01, 0.1)$.

this point in detail.

4.3 Outperforming Classical RK Integrators

In general, if an explicit m-stage Runge–Kutta method has order p, then it can be proven that the number of stages must satisfy $m \geq p$. In particular, for $p \geq 5$ one can show that we require $m \geq p + 1$, so no five-stage RK method can reach order 5 global truncation accuracy [6]. However, we now show that if we limit our attention to a specific task family, our approach can learn integrators that overcome this limitation, at least within a limited range of integration step-sizes.

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 in fig. 4 for the nonlinear (square) task family. We observe that we can indeed learn a two-stage RK-NN that has third-order accuracy (the slope of the RK-NN error is 3) for this task family. This is not possible for the general class of Lipschitz functions, for which a two-stage method can only achieve second-order global accuracy.

For this specific family, we can in fact understand this phenomenon precisely. Recall the two-stage RK method for equation eq. (4):

$$\boldsymbol{k}_1 = hf(\boldsymbol{y}_n), \quad \boldsymbol{k}_2 = hf(\boldsymbol{y_n} + \theta_1 \boldsymbol{k}_1), \quad \boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \theta_{c1}\boldsymbol{k}_1 + \theta_{c2}\boldsymbol{k}_2, \tag{19}$$

where $\theta_1, \theta_{c1}, \theta_{c2}$ are constants.

For the square task family with $d = 1$, the right hand side of equation eq. (4) is

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f(y) = -ay^2, \qquad y(0) = y_0 \in \mathbb{R}. \tag{20}$$

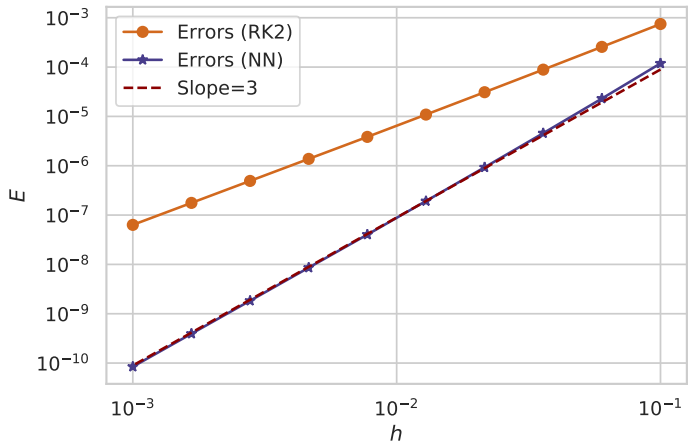


Figure 4: Error analysis on **square task families**, training on $h \in (0.01, 0.1)$ but testing on $h \in (0.001, 0.1)$, using **two-stage** RK-NN integrator with **third-order** Taylor-based loss as the regularizer.

Then we can obtain

$$\begin{aligned} k_1 &= -ay_n^2 h, \quad k_2 = -ay_n^2 h + 2\theta_1 a^2 y_n^3 h^2 - \theta_1^2 a^3 y_n^4 h^3, \\ y_{n+1} &= y_n - (\theta_{c1} + \theta_{c2})\, ay_n^2 h + 2\theta_1 \theta_{c2} a^2 y_n^3 h^2 - \theta_1^2 \theta_{c2} a^3 y_n^4 h^3. \end{aligned} \tag{21}$$

We expand the true solution at t_{n+1}, subject to the initial condition y_n at t_n. Due to Taylor's theorem,

$$\begin{aligned} \widetilde{y}(t_{n+1}) &= y_n + y_n' h + \frac{1}{2} y_n'' h^2 + \frac{1}{6} y_n''' h^3 + \mathcal{O}(h^4) \\ &= y_n - ay_n^2 h + a^2 y_n^3 h^2 - a^3 y_n^4 h^3 + \mathcal{O}(h^4). \end{aligned} \tag{22}$$

Comparison between eq. (21) and eq. (22) shows the conditions for obtaining a third-order integrator by a two-stage RK method:

$$\theta_{c1} + \theta_{c2} = 1, \quad 2\theta_1 \theta_{c2} = 1, \quad \theta_1^2 \theta_{c2} = 1. \tag{23}$$

Note that this relies on the fact that we are only considering the family of vector fields of the form $\{-ay_n^2\}$. In general, it is not possible to obtain third-order accuracy only with a two-stage RK-type integrator. Indeed, the coefficients in the learned RK-NN,

$$\theta_1 = 2, \quad \theta_{c1} = 0.75, \quad \theta_{c2} = 0.25, \tag{24}$$

are consistent with conditions in eq. (23), while the standard RK2 coefficients,

$$\theta_1^{\mathrm{RK}} = 1, \quad \theta_{c1}^{\mathrm{RK}} = 0.5, \quad \theta_{c2}^{\mathrm{RK}} = 0.5, \tag{25}$$

are not.

In this example, two-stage RK-NN can provably achieve third-order accuracy, due to the special structure in the task family. In general, one will not expect this to hold for all values of h, especially those far out of the training regime. Similarly, we also obtain a four-stage RK-NN integrator having sixth-order accuracy when training and testing over $h \in (0.01, 0.1)$, which is shown in supplementary material **SM1**.

4.4 Training a RK-NN Integrator on the Van der Pol Oscillator and the Brusselator

Up to now, we studied the performance of our RK-NN method on 1-dimensional tasks. Now we apply our method to multivariate instances.

Van der Pol Oscillator

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} - a\left(1 - u^2\right)\frac{\mathrm{d}u}{\mathrm{d}t} + u = 0, \tag{26}$$

where the parameter a is a scalar parameter indicating the nonlinearity and the strength of the damping. Another commonly used form based on the transformation $v = \dot{u}$ leads to:

$$\begin{aligned} \dot{u} &= v, \\ \dot{v} &= a\left(1 - u^2\right)v - u. \end{aligned} \tag{27}$$

For the purposes of our method, $\boldsymbol{y}$ is vector-valued with $\boldsymbol{y} = (u, v)$.

The Brusselator

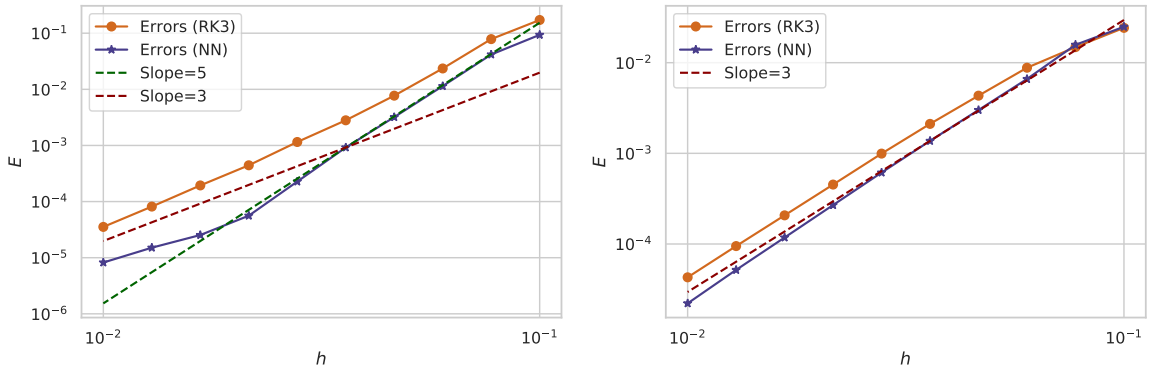
$$\begin{aligned} \dot{u} &= 1 - (b+1)u + au^2 v, \\ \dot{v} &= bu - au^2 v, \end{aligned} \tag{28}$$

where $a, b > 0$ are constants and $u, v \in \mathbb{R}$. u and v represent the dimensionless concentrations of two of the reactants. Same as the notation in the Van der Pol oscillator, the dynamic variable is also the vector $\boldsymbol{y} = (u, v)$ in our method.

Conducting experiments on the mentioned 2-dimensional tasks, we can obtain an integrator better than the traditional RK method on specific ODE problems. The highlight is that our trained integrator brings lower error between the true solution and prediction value than the RK-method with the same accuracy order.

It is worth considering why we do not directly use Taylor series with the same order truncation error as our integrator, now that we can leverage the ability to get the differential of $\boldsymbol{y}$ or h during our training. When it comes to a third-order algorithm, the integrator based on Taylor series is defined as $\boldsymbol{y}(h) = \boldsymbol{y}(0) + \sum_{i=1}^{3} \frac{1}{i!}\boldsymbol{y}^{(i)}(0)h^i$. We will show the comparison among our RK-NN method, RK3 method and Taylor series integrator in the following experiments.

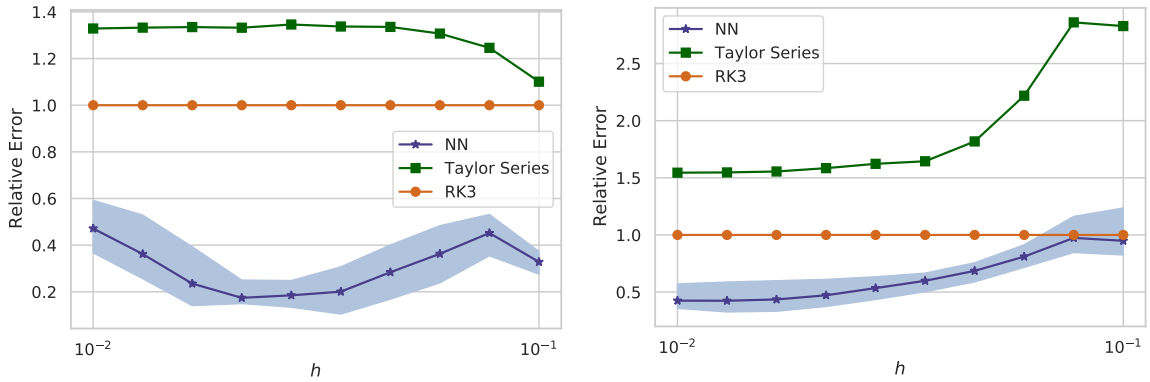
fig. 5 shows that the learned integrator for each task can achieve third-order accuracy respectively on the Van der Pol oscillator and the Brusselator families. To make the comparison clear, we plot the geometric mean of the relative errors with quantified uncertainties with respect to the RK method.



(a) Van der Pol oscillator. (b) The Brusselator.

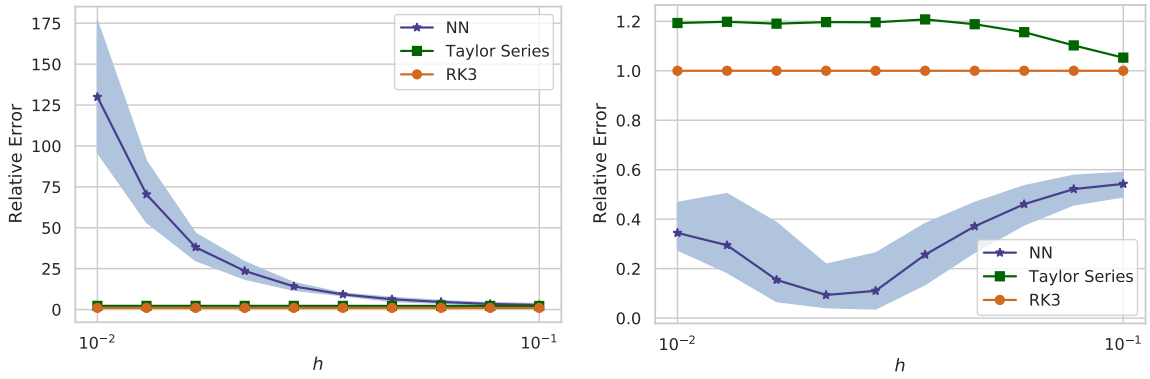
Figure 5: Order Check after training for a **third-order** integrator by **three-stage** RK-NN integrator.

For Van der Pol oscillator families, the inputs are sampled with $a \sim U(1,2)$ as vector field parameters and $\{\boldsymbol{y}_0 = (u_0, v_0); u_0 \sim U(-4,-3), v_0 \sim U(0,2)\}$ as initial conditions. Better performance can be obtained within our training range, which is shown in fig. 6a. When we evaluate outside the training range of parameter a, fig. 7a tells us the trained RK-NN cannot works badly when $a \in (0,1)$ but well when $a \in (2,3)$ in fig. 7b. This illustrates our RK-NN performs accurately on the training range, but it is hard to say how well it works outside. Similar results can be achieved when testing outside the training range of the initial values (u_0, v_0) in fig. 8.



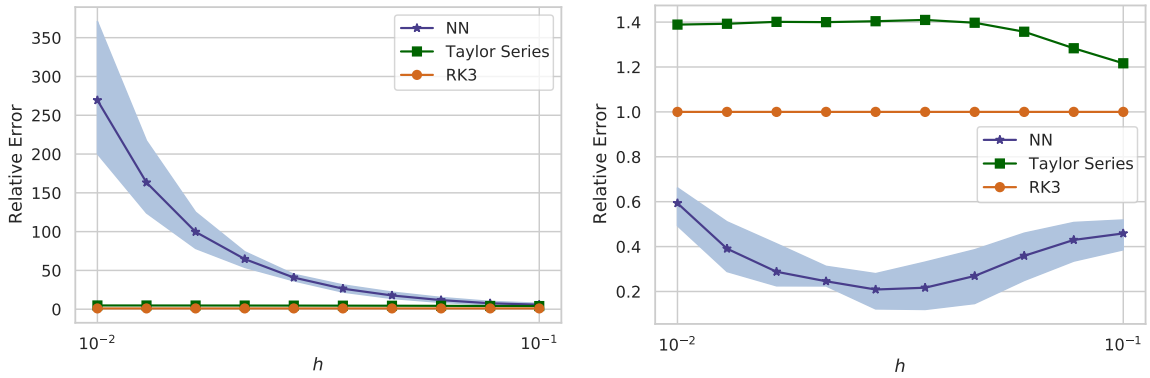
(a) Van der Pol oscillator. (b) The Brusselator.

Figure 6: Relative error analysis.



(a) $a \in (0, 1)$. (b) $a \in (2, 3)$.

Figure 7: Evaluation on Van der Pol oscillator by using a outside the training range.



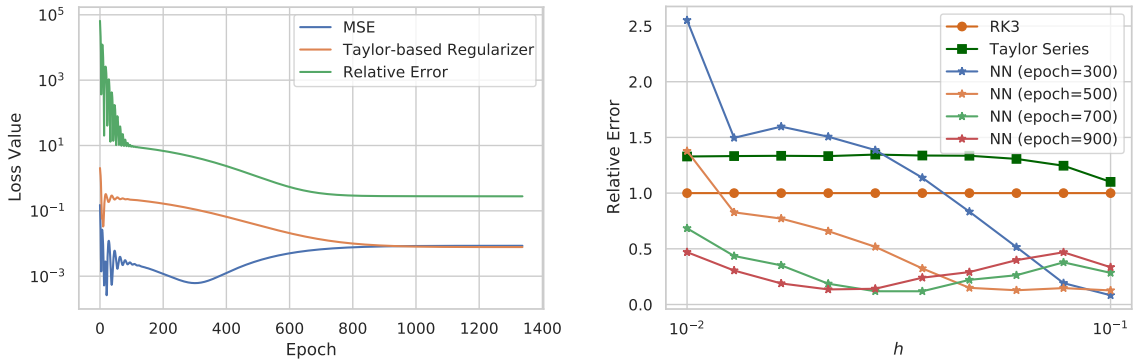
(a) $u_0 \in (-1, 1), v_0 \in (-1, 1)$. (b) $u_0 \in (3, 4), v_0 \in (-2, 0)$.

Figure 8: Evaluation on the Van der Pol oscillator by using inputs (u_0, v_0) outside the training range.

Training the RK-NN integrator on the Brusselator families with $b \sim U(0.5, 2)$ and $a = 1$, we use $\{\boldsymbol{y}_0 = (u_0, v_0); u_0 \sim U(1.5, 3), v_0 \sim U(2, 3)\}$ as initial conditions. We observe our method performs favorably inside our training range from fig. 6b. The trained integrator has worse performance if the inputs are outside the training range, which is shown in supplementary material **SM2**. We also evaluate the worse performance on the Van der Pol oscillator and the Brusselator when time step h is outside the training range in supplementary material **SM3**.

To better understand the training process and the role of the two losses (MSE without a scale, Taylor-based regularizer), the curves of losses during the training are shown in fig. 9a. Notice that there is a minimum of MSE loss (epoch=300) for the Van der Pol oscillator, and then it increases to a stable value (epoch from 300 to 900). The performance at that minimum point and subsequent trend are shown in fig. 9b. Here, the relative error is smaller compared with that from the final trained integrator when

h is around 0.1. Obviously, as h increases, the prediction will become more inaccurate. Since the MSE error obtained in the training process is the average of the MSE losses at different time steps, the errors obtained when h is large will dominate the average. With this specific parameterization (at the MSE minimum point), the integrator performs well when h is around 0.1, but its accuracy cannot be extended to the entire training range. However, the final training results (fig. 6a) show that the integrator performs slightly worse under the larger time step, but favorable performance is obtained for all h within the training range. This is enfoced by minimizing the Taylor loss, which ensures the appropriate order of the integrator, despite increasing the MSE.



(a) Training loss curve.

(b) Relative errors at different epochs.

Figure 9: The performance of the Van der Pol oscillator during training is depicted in these two figures. The left hand side shows the loss curves of MSE between the predicted value and true value, Taylor-based regularizer, and the relative error compared with the reference RK method. The right hand side illustrates relative error analysis on the Van der Pol oscillator around the minimum point and later.

In summary, we observe that when applied to a problem coming from the family RK-NN is trained on, the RK-NN can achieve higher accuracy compared with the classical RK methods within the training range of h. In simple cases, this can be proved theoretically (e.g., section 4.3). In general, we demonstrated this numerically, and we observe that RK-NN finds "personalized" RK-like schemes for each problem family with superior performance (See supplementary material **SM4** for more quantitative comparisons). To further illustrate this point, we include table 1 that outputs the learned Butcher Tableau values for each problem and compares them with those coefficients in three generic RK3 methods. One can observe that in each problem, we obtain specialized values, which illustrates the adaptiveness of our method.

5 Related Work

In this section, we discuss related work in the literature. We divide them into four categories, namely learning solvers by neural networks, learning continuous-time dy-

Table 1: Parameter comparisons between three generic RK3 methods and three-stage RK-NN integrators over different task families. We repeatedly trained our RK-NN from multiple random initializations of neural networks and present three sets of parameters here, which are denoted by serial numbers in the Van der Pol oscillator and the Brusselator task families, as shown in this table.

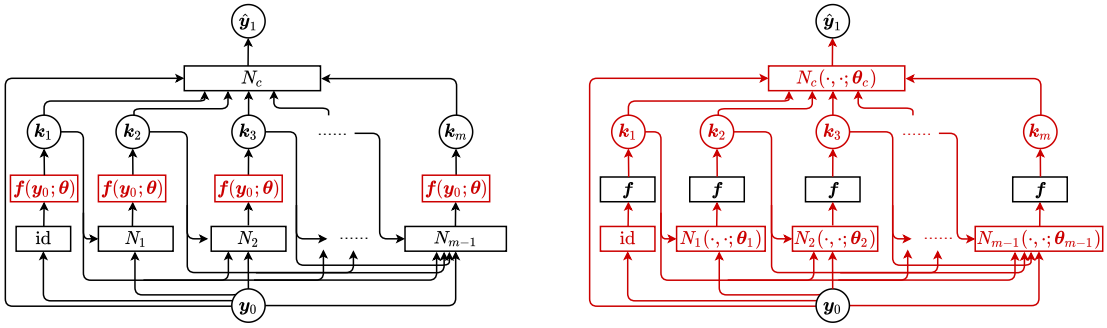
	$\theta_{1,1}$	$\theta_{2,1}$	$\theta_{2,2}$	θ_{c1}	θ_{c2}	θ_{c3}
Linear Task	0.9682	0.1036	1.0331	0.5124	0.3218	0.1657
Square Task	0.9206	0.8227	1.2625	0.5533	0.3704	0.0763
Van der Pol(#1)	0.6880	0.3382	0.7205	0.4333	0.2746	0.2920
Van der Pol(#2)	0.7224	0.2493	0.6112	0.3718	0.3203	0.3079
Van der Pol(#3)	0.7607	0.3976	0.7357	0.4704	0.2749	0.2547
Brusselator(#1)	0.7072	-0.1350	0.7387	0.2462	0.4344	0.3194
Brusselator(#2)	0.5459	-0.1794	0.9770	0.2258	0.4672	0.3070
Brusselator(#3)	0.6841	-0.0699	0.7215	0.2536	0.4178	0.3286
RK3(#1)	0.6667	-0.5	0.5	-0.25	0.75	0.5
RK3(#2)	0.6667	0.1667	0.5	0.25	0.25	0.5
RK3(#3)	0.5	-1	2	0.1667	0.6667	0.1667

namical systems using integrator-embedded parametrization, learning the integrators themselves, and finally meta-learning in general.

In the first direction, general methodologies for learning operators using ANNs have been proposed in [7, 44, 38]. This can be applied in particular to learning anti-derivative operators that can then be used for the solution of differential equations. The key difference in our case is that we do not represent generic operators. Instead, we parametrize solvers using a generalized integrator superstructure, from which we can derive novel loss functions that promote high-order accuracy.

In the second direction, integrators are useful in computing the solution of dynamical equations and neural networks are effective in approximating general vector fields corresponding to dynamical models. Thus, their combination naturally brings improved performance on modelling dynamical systems, and there are ample studies on using integrator embedded neural network architectures to learn dynamics from data. Based on the trajectory data, [33, 27, 32] use a feed-forward neural network as an integrator, which is trained to predict the state of the system. In addition, combined with some specific integrator scheme, neural networks can be used to accurately infer the vector fields corresponding to the ODEs. For example, [48, 47, 20] combine neural network parametrization of vector fields with RK integrator and learn from dynamical data. [37] combine networks with a grey-box RK RNN scheme and learn only the unknown terms of a differential equation right-hand-side (hardwiring the known terms). Similarly, the trapezoidal rule integrator [46, 2] and very high order implicit RK integrators are used to handle higher dimensional equations [43]. The goal of all

the work mentioned above is identifying the unknown vector field driving the evolution from observed trajectory data, using the structures of known integrators. For example, a RK integrator is used to predict the state value after one time step in fig. 10a, which allows for more accurate approximation of the unknown vector field when the trajectory data comes from the underlying continuous-time system. Backward error analysis can be used to establish the accuracy of such approximations [58]. However, our case is the opposite: the driving vector field is known, but what we seek is the structure of the (optimal) integrator through parameterization as a composite neural network as shown in fig. 10b.



(a) Learning vector field using embedded RK integrator with Neural Net $\boldsymbol{f}(\boldsymbol{y}_0;\boldsymbol{\theta})$.

(b) Learning the integrator parameterized by general form of RK method.

Figure 10: Schematic representation of the implementation of RK method. Highlight is approximated by neural networks. The vector field is trainable in fig. 10a and the integrator is built by neural networks in fig. 10b. Here $\hat{\boldsymbol{y}}_1$ is the prediction by RK integrator with initial condition $\boldsymbol{y}_0$. N_c and each N_i describe weighted summation.

In this reverse direction of designing integrator structures, there are a number of previous related studies [13, 14, 15, 16, 17, 52]. In these works, the authors numerically compute RK coefficients from analytically derived constraints. With the combination of neural networks, the parameters in integrator structures can be learned to meet specific requirements. To learn the coefficients in RK4, one constructs the loss function by the scaled distance between true and approximation value [57]. [1] and [9] are respectively concerned with the RK2 and RK3 method and augment the loss function to penalize deviation of the coefficients from the equations that are required for RK methods. However, the present method differs in a number of ways. First, we introduce a new loss term, based on Taylor series, which can automatically discover high order integrators, without the need to manually derive weight constraints, as is done in [28]. This makes our method easy to implement and scalable to high orders of accuracy. Second, having trajectory data from the true solution is not compulsory in our method, even though previous work always relies on true labels to calculate the error, since the Taylor series expansion is a local property and is entirely determined by the vector field $\boldsymbol{f}$ (See section 3.2).

Lastly, from the machine learning viewpoint, our approach aims to learn solvers

that perform well on a target family of tasks, rather than one fixed task. In this sense, it is related to multi-task learning [8], but with notable differences. Multi-task learning with deep neural networks aims to find shared feature extractors relevant for multiple pre-defined tasks, each of which is accomplished by specific task-dependent final layers. While we also consider multiple tasks, task-specific layer design does not appear in our architecture. Most importantly, in our setting the algorithm must generalize to new unseen tasks in the task family. In contrast, multi-task learning focuses on (often a small number of) pre-defined tasks. Based on this observation, it is hence more appropriate to view our setup as a generalized version of meta-learning [54]. Recent algorithmic developments in meta-learning include Model-Agnostic Meta-Learning (MAML) [18], iMAML [45] and Reptile [42]. These methods aim to learn network intializations that can quickly adapt to unseen problem instances via gradient descent. While our RK-NN method does not have such an adaptation step, we have a similar component that uses the task information (vector field) to compute a solution trajectory by numerical integration. Hence, one can regard this as a generalized version of MAML if we replace the gradient descent adaptation steps in the latter with the numerical integration step in the former. It is a worthwhile future direction to investigate whether we can integrate the ideas from MAML with our approach to further improve the performance of the model *on a chosen task instance*.

6 Summary and Outlook

In this paper, we study how to effectively combine machine learning and numerical analysis ideas to arrive at efficient and adapted algorithms. As a case study, we developed a method to automatically learn high-order integrators for specified ODE families, based on the RK algorithmic superstructure. A key idea is the definition of a RK-like neural network architectural superstructure (RK-NN), together with a Taylor series based regularizer that ensures high order accuracy and adaptivity to the problem class. Instead of computing the Butcher tableau, we focus on the performance of our model under multiple tasks following some task distribution. Based on that, we can sample and train RK-NN by minimizing the sum of the scaled least-squares error and the regularizer. In the average sense under this distribution, the method has superior performance to the classical RK method because it can exploit the structure that may be present. We apply this method to various examples, including the Van der Pol oscillator and the Brusselator, where we demonstrate that RK-NN brings lower global truncation error than the classical RK method. Overall, this represents a basic step towards a systematic investigation of learning-based approaches towards numerical algorithm design and adaptation.

Here, we considered minimizing the global errors among distinct integrators with respect to some fixed time step. However, prior work has shown that numerical computation can be improved if we adapt the step size during the evaluation. The order of accuracy of a numerical integration scheme is a convergence statement for vanishing

step size. In practice, and with finite step size, the actual errors of the numerical solution compared to the true solution can vary widely. A typical approach to control this is to use a higher-order method in each iteration to estimate the local error. The step size will be reduced if this estimated error is above a user-defined tolerance. Seminal work on this idea was done by Romberg [49] in the context of integration. Considering the explicit forward Euler method as an example, the adaptive step size method uses half of the current step size as a more accurate version. If the error estimate is below the tolerance, we consider the step successful and continue with the next step. Else, we adapt the step size through a specific equation. A particular challenge for adaptive step size control is the increased computational cost due to the "second" evaluation of the method in each iteration. To minimize the number of function evaluations, E. Fehlberg developes multiple versions of the Runge-Kutta type integrators and minimizes the coefficients in the scheme while simultaneously minimizing the error of the fourth-order scheme [13, 14, 15, 16, 17], leading to "Runge-Kutta-Fehlberg". Dormand and Prince later develop a scheme that optimizes the coefficients for a 5th-order method. They also use the first-same-as-last (FSAL) property to reduce the number of function evaluations necessary for both a fourth and fifth order accurate scheme [29, 10]. In particular, their method of combining six function evaluations to obtain results for a fourth- and fifth-order accurate RK scheme simultaneously has been implemented in the ODE45 solver of MATLAB. Inspired by these studies, we can design an algorithm based on the approach introduced in this paper to automatically learn the adaptive step size for improved robustness in the future.

Throughout this paper we have considered a "fixed superstructure": RK-NNs with a fixed recurrent architecture, thus restricting the search space to RK integrators of a fixed stage and with fixed zero entries in the Butcher tableau. In the future, we envision that more choices will be left to the optimizer, i.e., the number of RK stages, together with all entries in the corresponding Butcher tableau will be learned as part of a meta-learning procedure. This in general includes implicit methods, as explored in [2]. Such additional degrees of freedom would define a problem that encompasses the general family of RK-based algorithms. We have started to work on more general Krylov-inspired recurrent NN architectures. In order to find an optimal algorithm within such a superstructure, *the neural network architecture and its parameters* need to be optimized jointly. The use of global, mixed-integer optimization over superstructures in order to discover optimal algorithms has been advocated (and illustrated) in [40], where the meta-learning problem was formulated as an optimal control problem. It is interesting to summarize the experience of these authors with their approach, which they called "optimal algorithm generation" rather than "meta-learning": Depending on the subclass of problems over which they optimized, they sometimes found standard algorithms (that are documented in their textbooks); at other times they found algorithms unknown (to them), that, upon literature search, had been previously discovered; and upon occasion, they found algorithms for the problem subclass that "made perfect sense", and that somebody could have discovered, but apparently had not yet been discovered and documented. A major conclusion was that the opti-

mal algorithm was extremely sensitive to the fine scale details of the particular problem subclass, s.t. algorithms both previously unknown and also general could not be found. Our argument here is that this "generality weakness" is in effect a "personalized strength": that we will maybe have integrators (and more generally algorithms, and sometimes hardware computers, like D.E.Shaw's ANTON [51]) tuned to a particular class of equations, with particular types/ranges of initial/boundary conditions, over particular scales. The recent paper by Brenner and co-workers [4] on ML-discovered, problem-dependent PDE discretizations also moves in the same overall direction. The more times a problem needs to be solved repeatedly, the more the effort for discovering a "personalized" optimal algorithm is justified.

In the context of machine learning, one particular type of meta-learning, called neural architecture search (NAS) refers to a set of techniques for finding the optimal network architecture for a given task [26]. Beyond the search space, which is herein defined by the superstructure, NAS methods are categorized by their search strategy and their performance estimation strategy [12]. Current research is focused on search strategies based on reinforcement learning [59] and Bayesian optimization (e.g., [30, 36]), while evolutionary algorithms have been employed in NAS for decades (e.g., [3]). Pruning, i.e., deletion of neural network parameters based on some importance metric, can be interpreted as a widely used search strategy for NAS when the aim is to discover sparse model representations. Early pruning methods were based on Taylor series approximations of the sensitivity of the loss function with respect to network parameters [34, 25]. For deep networks, pruning based on weight magnitude is an established approach (e.g., [24, 50]), but recent work also considers information criteria such as synaptic saliency to achieve the highest possible model sparsity [53]. To reduce the complexity of the search space NAS methods can further exploit modularity within the superstructure [60, 5, 41]. For instance, for the RK-NN, it is particularly useful to think of all layers involving a function call and all skip connections between these layers as prunable units.

Beyond NAS, meta-learning on a superstructure would facilitate optimizing algorithms with respect to a metric that depicts an optimal trade-off between the complexity and the accuracy of each algorithmic step. A prime example for such a metric is the wall-clock performance of the algorithm for achieving some pre-defined stopping criterion, which is determined by the number of iterations and the cost per iteration (see [40]) Since the cost per iteration depends on the number and type of mathematical operations it involves, e.g., function evaluations, matrix-vector products or computations of a Jacobian, the corresponding superstructure optimization problem constitutes a mixed-integer nonlinear program (MINLP). The NAS methods above are capable of finding approximate solutions to this problem due to their heuristic nature and are thus widely used. In contrast, integer programming (IP) algorithms search the space of possible architectures in a more rigorous manner than NAS methods by constructing relaxations and introducing cutting planes. However, IP algorithms exhibit prohibitive cost for larger networks and are thus not applied to training. One possible exception is [11], in which the authors circumvent the full complexity of the discrete

problem by evaluating neuron importance with a binary value and training a network to minimize the number of important neurons to achieve sparsity. For a smaller network such as the one we used in this work, discrete optimization appears tractable. However, we point out that the theory of IP algorithms requires global solution of all relaxed subproblems with each subproblem corresponding to a neural network training problem. While training may lead to global optima empirically [21], and under some circumstances provably [23], this observation does not hold for all neural networks [19]. Therefore, to *guarantee* finding the best solution, deterministic global optimization is necessary; [40] illustrate this. Neural network architectural superstructures allow for more flexible function bases and may be optimized deterministically in similar fashion by the algorithms for superstructure optimization given in [22] and [39]. We do not envision (due to the tremendous computational difficulty of the problem) that mixed integer global optimization over superstructures will soon become the standard tool for optimal NN architecture generation for a "personalized" computational task over a class of problems of interest. More practical tools, e.g., exploiting modular structure, and informed pruning, will certainly carry the day in the foreseeable future. Yet the superstructure formulation of the optimization problem, and crucially the construction of an intelligent and flexible superstructure, informed by calculus and traditional numerical analysis, appears to us a truly worthy research task in this meta-learning quest for personalized algorithm generation.

References

[1] A. A. Anastassi, Constructing runge–kutta methods with the use of artificial neural networks, Neural Computing and Applications, 25 (2014), pp. 229–236.

[2] J. Anderson, I. Kevrekidis, and R. Rico-Martinez, A comparison of recurrent training algorithms for time series analysis and system identification, Computers & chemical engineering, 20 (1996), pp. S751–S756.

[3] P. J. Angeline, G. M. Saunders, and J. B. Pollack, An evolutionary algorithm that constructs recurrent neural networks, IEEE Transactions on Neural Networks, 5 (1994), pp. 54–65, https://doi.org/10.1109/72.265960.

[4] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner, Learning data-driven discretizations for partial differential equations, Proceedings of the National Academy of Sciences, 116 (2019), pp. 15344–15349.

[5] L. Boecking, P. Philipp, and C. Kulbach, Towards modular neural architecture search, ICLR Workshop on Neural Architecture Search, (2020), https://drive.google.com/file/d/1SLBnJe_X9erS3E0PytxK1ztPAEV37611/view.

[6] J. C. Butcher, Numerical methods for ordinary differential equations, John Wiley & Sons, 2016.

[7] T. Chen and H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems, IEEE Transactions on Neural Networks, 6 (1995), pp. 911–917.

[8] M. Crawshaw, Multi-task learning with deep neural networks: A survey, arXiv preprint arXiv:2009.09796, (2020).

[9] M. Dehghanpour, A. Rahati, and E. Dehghanian, Ann-based modeling of third order runge kutta method, Journal of Advanced Computer Science & Technology, 4 (2015), pp. 180–189.

[10] J. Dormand and P. Prince, A family of embedded Runge-Kutta formulae, Journal of Computational and Applied Mathematics, 6 (1980), pp. 19–26, https://doi.org/10.1016/0771-050X(80)90013-3.

[11] M. ElAraby, G. Wolf, and M. Carvalho, Identifying critical neurons in ann architectures using mixed integer programming, 2020, https://arxiv.org/abs/2002.07259.

[12] T. Elsken, J. H. Metzen, and F. Hutter, Neural architecture search: A survey, Journal of Machine Learning Research, 20 (2019), pp. 1–21, http://jmlr.org/papers/v20/18-598.html.

[13] E. Fehlberg, New high-order Runge-Kutta formulas with step size control for systems of first-and second-order differential equations, ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik, 44 (1964), https://doi.org/10.1002/zamm.19640441310.

[14] E. Fehlberg, New high-order Runge-Kutta formulas with an arbitrarily small truncation error, ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik, 46 (1966), pp. 1–16, https://doi.org/10.1002/zamm.19660460102.

[15] E. Fehlberg, Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems, tech. report, NASA, 1969.

[16] E. Fehlberg, Klassische runge-kutta-formeln vierter und niedrigerer ordnung mit schrittweiten-kontrolle und ihre anwendung auf waermeleitungsprobleme, Computing, 6 (1970), pp. 61–71.

[17] E. Fehlberg, Klassische runge-kutta-nyström-formeln mit schrittweiten-kontrolle für differentialgleichungen \ddot x= f (t, x,\dot x) , Computing, 14 (1975), pp. 371–387.

[18] C. Finn, P. Abbeel, and S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, arXiv preprint arXiv:1703.03400, (2017).

[19] J. Frankle, Revisiting "qualitatively characterizing neural network optimization problems", 2020, https://arxiv.org/abs/2012.06898.

[20] R. Gonzalez-Garcia, R. Rico-Martinez, and I. Kevrekidis, Identification of distributed parameter systems: A neural net based approach, Computers & chemical engineering, 22 (1998), pp. S965–S968.

[21] I. J. Goodfellow, O. Vinyals, and A. M. Saxe, Qualitatively characterizing neural network optimization problems, arXiv preprint arXiv:1412.6544, (2014).

[22] I. E. Grossmann, Review of nonlinear mixed-integer and disjunctive programming techniques, Optimization and Engineering, 3 (2002), pp. 227–252, https://doi.org/10.1023/A:1021039126272.

[23] B. D. Haeffele and R. Vidal, Global optimality in neural network training, in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 4390–4398, https://doi.org/10.1109/CVPR.2017.467.

[24] S. Han, H. Mao, and W. J. Dally, Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding, in 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, Y. Bengio and Y. LeCun, eds., 2016, http://arxiv.org/abs/1510.00149.

[25] B. Hassibi and D. Stork, Second order derivatives for network pruning: Optimal brain surgeon, in Advances in Neural Information Processing Systems, S. Hanson, J. Cowan, and C. Giles, eds., vol. 5, Morgan-Kaufmann, 1993, https://proceedings.neurips.cc/paper/1992/file/303ed4c69846ab36c2904d3ba8573050-Paper.pdf.

[26] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, Meta-learning in neural networks: A survey, 2020, https://arxiv.org/abs/2004.05439.

[27] J. Hudson, M. Kube, R. Adomaitis, I. Kevrekidis, A. Lapedes, and R. Farber, Nonlinear signal processing and system identification: applications to time series from electrochemical reactions, Chemical Engineering Science, 45 (1990), pp. 2075–2081.

[28] A. Iserles, A first course in the numerical analysis of differential equations, no. 44, Cambridge university press, 2009.

[29] P. J. P. J. R. Dormand, New Runge-Kutta algorithms for numerical simulation in dynamical astronomy, Celestial mechanics and Dynamical Astronomy, 18 (1978), pp. 223–232.

[30] K. Kandasamy, W. Neiswanger, J. Schneider, B. Póczos, and E. P. Xing, Neural architecture search with bayesian optimisation and optimal transport, in Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Red Hook, NY, USA, 2018, Curran Associates Inc.

[31] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980, (2014).

[32] K. Krischer, R. Rico-Martínez, I. Kevrekidis, H. Rotermund, G. Ertl, and J. Hudson, Model identification of a spatiotemporally varying catalytic reaction, AIChE Journal, 39 (1993), pp. 89–98.

[33] A. Lapedes and R. Farber, How neural nets work, in Evolution, learning and cognition, World Scientific, 1988, pp. 331–346.

[34] Y. LeCun, J. Denker, and S. Solla, Optimal brain damage, in Advances in Neural Information Processing Systems, D. Touretzky, ed., vol. 2, Morgan-Kaufmann, 1990, https://proceedings.neurips.cc/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

[35] Q. Li, C. Tai, and E. Weinan, Stochastic modified equations and adaptive stochastic gradient algorithms, in International Conference on Machine Learning, PMLR, 2017, pp. 2101–2110.

[36] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, Progressive neural architecture search, in Computer Vision – ECCV 2018, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, eds., Cham, 2018, Springer International Publishing, pp. 19–35.

[37] R. J. Lovelett, J. L. Avalos, and I. G. Kevrekidis, Partial observations and conservation laws: Gray-box modeling in biotechnology and optogenetics, Industrial & Engineering Chemistry Research, 59 (2019), pp. 2611–2620.

[38] L. Lu, P. Jin, and G. E. Karniadakis, Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, arXiv preprint arXiv:1910.03193, (2019).

[39] L. Mencarelli, Q. Chen, A. Pagot, and I. E. Grossmann, A review on superstructure optimization approaches in process system engineering, Computers & Chemical Engineering, 136 (2020), p. 106808, https://doi.org/https://doi.org/10.1016/j.compchemeng.2020.106808, https://www.sciencedirect.com/science/article/pii/S0098135419313924.

[40] A. Mitsos, J. Najman, and I. G. Kevrekidis, Optimal deterministic algorithm generation, Journal of Global Optimization, 71 (2018), pp. 891–913, https://doi.org/10.1007/s10898-018-0611-8.

[41] R. Negrinho, M. Gormley, G. J. Gordon, D. Patil, N. Le, and D. Ferreira, Towards modular and programmable architecture search, in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds., vol. 32, Curran Associates, Inc., 2019, https://proceedings.neurips.cc/paper/2019/file/4ab50afd6dcc95fcba76d0fe04295632-Paper.pdf.

[42] A. Nichol, J. Achiam, and J. Schulman, On first-order meta-learning algorithms, arXiv preprint arXiv:1803.02999, (2018).

[43] M. Raissi, P. Perdikaris, and G. Karniadakis, Physics informed deep learning (part ii): Data-driven, discovery of nonlinear partial differential equations,", arxiv e-prints, p, arXiv preprint arXiv:1711.10566, (2017).

[44] M. Raissi, P. Perdikaris, and G. E. Karniadakis, Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, arXiv preprint arXiv:1711.10561, (2017).

[45] A. Rajeswaran, C. Finn, S. Kakade, and S. Levine, Meta-learning with implicit gradients, arXiv preprint arXiv:1909.04630, (2019).

[46] R. Rico-Martinez, I. Kevrekidis, and K. Krischer, Nonlinear system identification using neural networks: dynamics and instabilities, Neural networks for chemical engineers, (1995), pp. 409–442.

[47] R. Rico-Martinez and I. G. Kevrekidis, Continuous time modeling of nonlinear systems: A neural network-based approach, in IEEE International Conference on Neural Networks, IEEE, 1993, pp. 1522–1525.

[48] R. Rico-Martinez, K. Krischer, I. Kevrekidis, M. Kube, and J. Hudson, Discrete-vs. continuous-time nonlinear signal processing of cu electrodissolution data, Chemical Engineering Communications, 118 (1992), pp. 25–48.

[49] W. Romberg, Vereinfachte numerische Integration, (1955), pp. 30–36.

[50] A. See, M.-T. Luong, and C. D. Manning, Compression of neural machine translation models via pruning, in Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning, Berlin, Germany, Aug. 2016, Association for Computational Linguistics, pp. 291–301, https://doi.org/10.18653/v1/K16-1029, https://www.aclweb.org/anthology/K16-1029.

[51] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, et al., Anton, a special-purpose machine for molecular dynamics simulation, Communications of the ACM, 51 (2008), pp. 91–97.

[52] E. Süli and D. F. Mayers, An introduction to numerical analysis, Cambridge university press, 2003.

[53] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli, Pruning neural networks without any data by iteratively conserving synaptic flow, in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds., vol. 33, Curran Associates, Inc., 2020, pp. 6377–6389, https://proceedings.neurips.cc/paper/2020/file/46a4378f835dc8040c8057beb6a2da52-Paper.pdf.

[54] S. Thrun and L. Pratt, Learning to Learn: Introduction and Overview, Springer US, Boston, MA, 1998, pp. 3–17, https://doi.org/10.1007/978-1-4615-5529-2_1, https://doi.org/10.1007/978-1-4615-5529-2_1.

[55] J. Traub, G. Wasilkowski, and H. Wozniakowski, Information-based complexity", acad. press, INC., New York, (1988).

[56] J. F. Traub, A general theory of optimal algorithms, tech. report, 1980.

[57] C. Tsitouras, Neural networks with multidimensional transfer functions, IEEE transactions on neural networks, 13 (2002), pp. 222–228.

[58] A. Zhu, P. Jin, and Y. Tang, Inverse modified differential equations for discovery of dynamics, arXiv preprint arXiv:2009.01058, (2020).

[59] B. Zoph and Q. V. Le, Neural architecture search with reinforcement learning, 2017, https://arxiv.org/abs/1611.01578.

[60] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, Learning transferable architectures for scalable image recognition, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 8697–8710.

SUPPLEMENTARY MATERIALS

We introduce some additional results in these supplementary materials. In **SM1**, a four-stage RK-NN integrator is trained to sixth-order accuracy over a range of h in a specific task family, which breaks the order barrier of the classic RK method. **SM2** illustrates worse performance on the Brusselator is obtained when initial condition $\boldsymbol{y}_0$ or the equation parameter b is out of the training range. We also evaluate both the Van der Pol oscillator and the Brusselator to see what happens when time step h is out of training range in **SM3**. In **SM4**, RK-NN is compared to three classic RK3 methods with different parameterization. Finally, we show the training complexity of a linear task by plotting the wall-clock time required for one epoch in **SM5**.

SM1. Additional outperforming experiment results

We apply the same approach (Alg.1) to train a superior four-stage RK-NN with $\alpha = 6$ and $m = 4$. It is well known that one requires a seven-stage RK method to obtain sixth-order accuracy for generic ODEs. Figure 11 shows that we can obtain a sixth-order integrator with four-stage RK-NN for square task family eq. (29) (detailed definition is illustrated in section 4 of the main article).

$$\begin{aligned} \mathcal{F} &= \{\boldsymbol{y} \mapsto -a\boldsymbol{y}^2 \mid a > 0\} \times \{\mathbb{R}\}, \\ \mu &= \text{Distribution}(\{\boldsymbol{y} \mapsto -a\boldsymbol{y}^2; a \sim U(0.1, 0.5)\}) \times U(1, 3). \end{aligned} \tag{29}$$

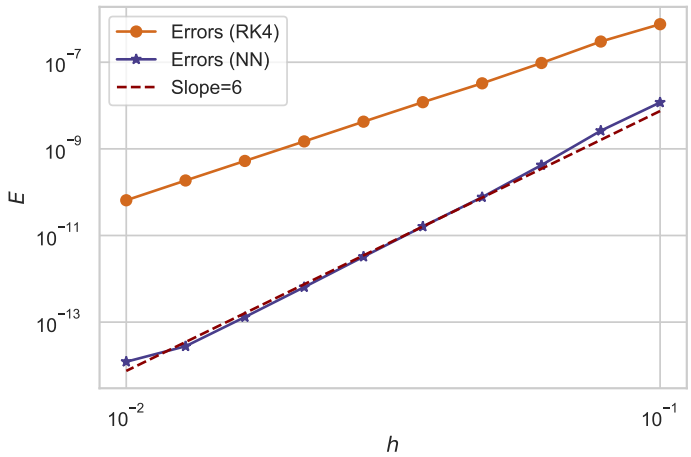
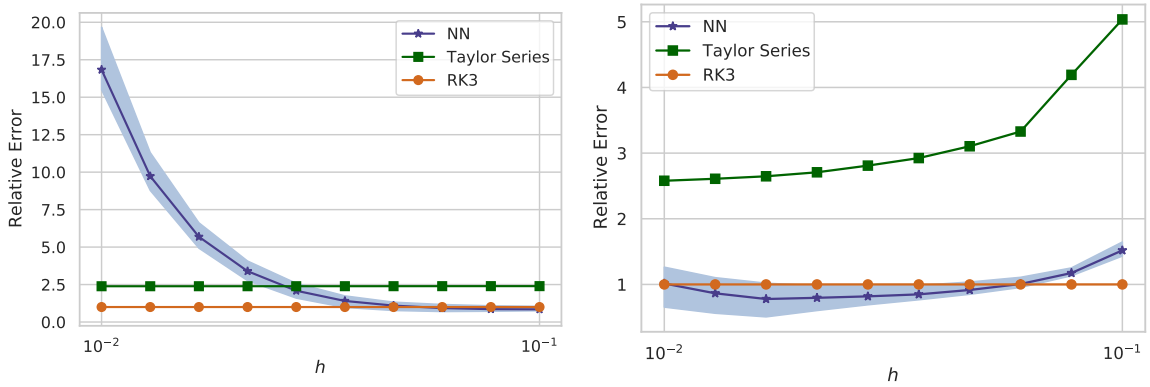


Figure 11: Error analysis on **square task** family, after training and testing on $h \in (0.01, 0.1)$, using **four-stage** RK-NN integrator with **sixth-order** Taylor-based loss as regularizer.

SM2. Error analysis on the Brusselator with initial conditions or parameters outside training range

Training RK-NN integrator on the Brusselator families with $b \sim U(0.5, 2)$ and $a = 1$, we use $\{\boldsymbol{y}_0 = (u_0, v_0); u_0 \sim U(1.5, 3), v_0 \sim U(2, 3)\}$ as initial condition. We test on the initial condition $u_0 \in (0.5, 1), v_0 \in (1, 2)$ or parameter values $b \in (3.5, 4)$. Evaluation results are shown in fig. 12.



(a) $u_0 \in (0.5, 1), v_0 \in (1, 2)$.

(b) $b \in (3.5, 4)$.

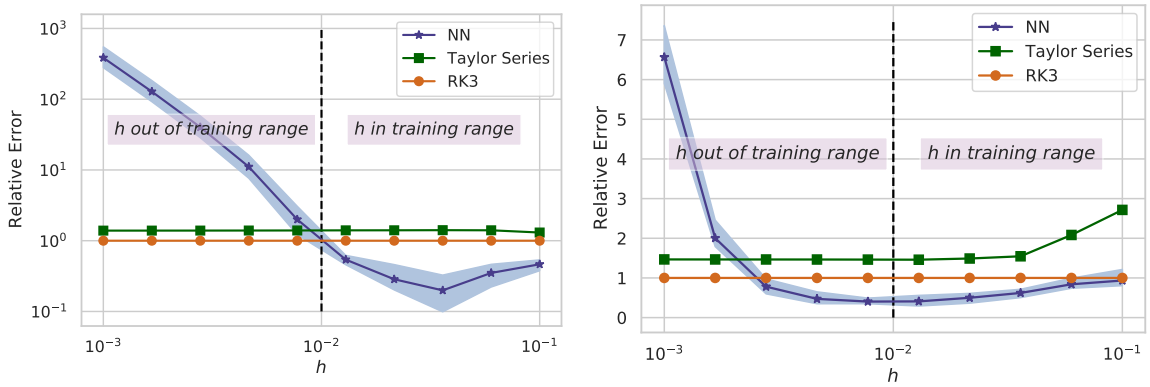
Figure 12: Evaluation on the Brusselator by using inputs outside the training range.

SM3. Evaluation on the Van der Pol oscillator and the Brusselator while time step h out of training range

In the previous experiments, we train and test on $h \in (0.01, 0.1)$. We test the trained integrator on a larger time step range, such as $h \in (0.001, 0.1)$, to study its generalization in terms of time steps. Figure 13 and fig. 14 show that the performance worsens when h goes to a smaller value. This demonstrates that the learned integrators are also adapted to the range of step sizes during training. Note that for most applications, the current range of step sizes gives sufficiently small errors, thus this is not a major issue of the method.

SM4. Comparison with three different traditional RK3 methods

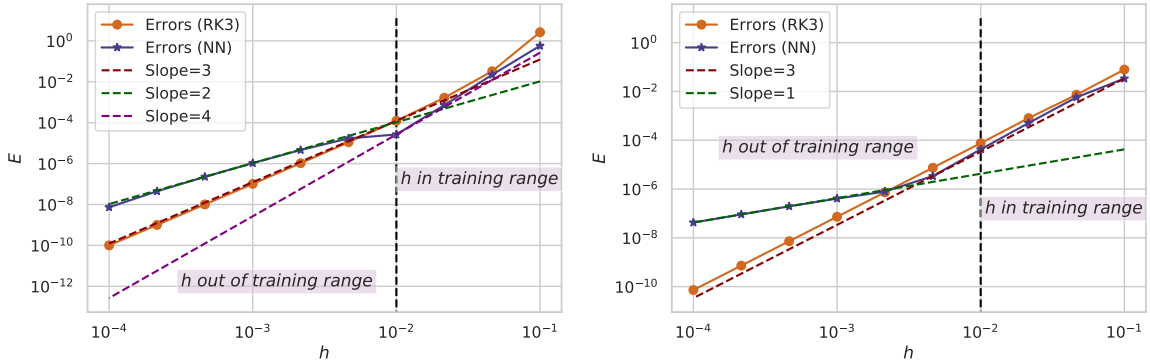
In this paper, our goal is to find an approximate Butcher tableau for some given problems and range of time step h. To figure out whether our method is better, we



(a) Van der Pol oscillator.

(b) The Brusselator.

Figure 13: Evaluation on the relative error when h out of training range.



(a) Van der Pol oscillator.

(b) The Brusselator.

Figure 14: Evaluation on the global error when h out of training range.

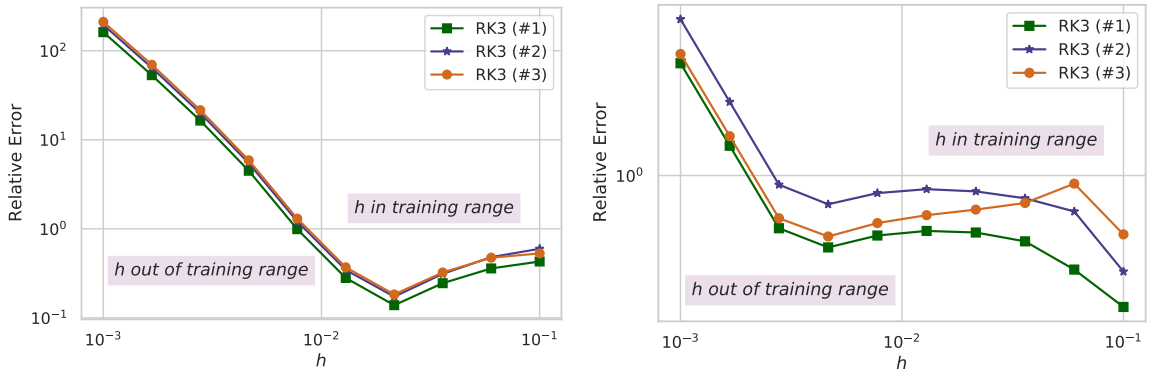
decide to calculate results by RK3 methods with different parameters on our examples. Recall the formulation of RK3:

$$\begin{aligned} \boldsymbol{k}_1 = hf(\boldsymbol{y}_n), \quad \boldsymbol{k}_2 = hf(\boldsymbol{y_n} + \theta_{1,1}\boldsymbol{k}_1), \quad \boldsymbol{k}_3 = hf(\boldsymbol{y_n} + \theta_{2,1}\boldsymbol{k}_1 + \theta_{2,2}\boldsymbol{k}_2), \\ \boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \theta_{c1}\boldsymbol{k}_1 + \theta_{c2}\boldsymbol{k}_2 + \theta_{c3}\boldsymbol{k}_3. \end{aligned} \tag{30}$$

Here is shown three different parameterized RK3:

- RK3 (#1) $\theta_{1,1} = \frac{2}{3}, \theta_{2,1} = -\frac{1}{2}, \theta_{2,2} = \frac{1}{2}, \theta_{c1} = -\frac{1}{4}, \theta_{c2} = \frac{3}{4}, \theta_{c3} = \frac{1}{2}$.
- RK3 (#2) $\theta_{1,1} = \frac{2}{3}, \theta_{2,1} = \frac{1}{6}, \theta_{2,2} = \frac{1}{2}, \theta_{c1} = \frac{1}{4}, \theta_{c2} = \frac{1}{4}, \theta_{c3} = \frac{1}{2}$.
- RK3 (#3) $\theta_{1,1} = \frac{1}{2}, \theta_{2,1} = -1, \theta_{2,2} = 2, \theta_{c1} = \frac{1}{6}, \theta_{c2} = \frac{2}{3}, \theta_{c3} = \frac{1}{6}$.

We define the error from RK3 as E_{RK} and from RK-NN as E_{NN}, then we have relative error $\frac{E_{NN}}{E_{RK}}$. RK-NN performs better if relative error is smaller than 1. fig. 15 illustrates the relative errors by RK3 methods with distinct coefficients are almost the same and our RK-NN has a better performance in the training range of time step $h \in (0.01, 0.1)$.



(a) Van der Pol oscillator.

(b) Brusselator.

Figure 15: Comparison of RK-NN results with different parameterized RK3 methods.

SM5. Training Complexity for ODE systems of different dimension

In fig. 16, we investigate empirically the scalability of our method during training. We plot the wall-clock time required for one epoch of training on an example linear family and a nonlinear family as the dimension d of the ODE increases. We observe that the training cost per epoch has a scaling between $O(d)$ and $O(d^2)$, meaning that our method can be effectively applied to moderately high dimensional systems. In the problems we have tested, the number of epochs required to reach a specified testing accuracy does not increase significantly with ODE dimension, and is in general problem dependent (See table 2).

Table 2: Comparison of the number of training epochs for ODE systems of different dimensions shown in fig. 16

Dimension	1	2	4	8	16	32
Linear Task	201	201	196	191	187	200
Nonlinear Task	2336	1831	1506	1511	1577	1527

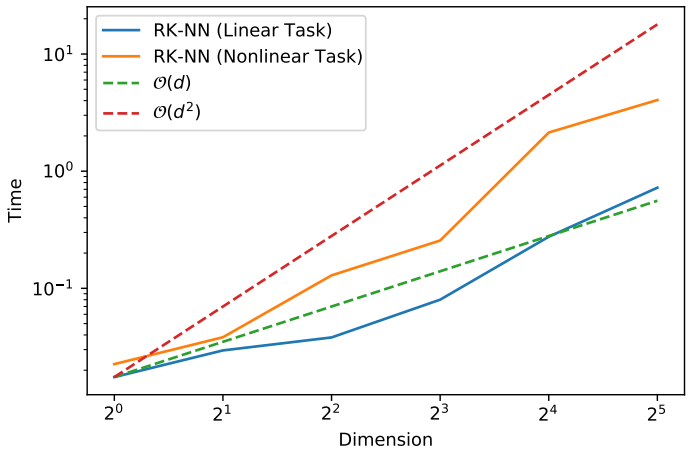


Figure 16: Training time per epoch for different dimensions. The equation of linear tasks is $\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y}, \boldsymbol{y}(0) = \boldsymbol{y}_0$, where $A \in \mathbb{R}^{d\times d}$ is a square matrix. The elements of A are independently and identically distributed with $A_{ij} \sim U(-\frac{1}{\sqrt{d}}, -\frac{2}{\sqrt{d}})$ and $\boldsymbol{y}_0 \sim U(-3, 3)$. The equation of nonlinear tasks is $\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = B\boldsymbol{y}^2, \boldsymbol{y}(0) = \boldsymbol{y}_0$. $B \in \mathbb{R}^{d\times d}$ is a diagonal matrix whose diagonal elements are randomly generated from different distributions. The i-th element $B_{ii} \sim U(-2 + 0.05(i-1), -2 + 0.05(i+1))$ and $\boldsymbol{y}_0 \sim U(1, 3)$. In this figure, the x-axis is the dimension d and the y-axis is the wall-clock time required for each training epoch.