A Recursively Recurrent Neural Network (R2N2) Architecture for Learning Iterative Algorithms

Danimir T. Doncevic[a,b] Alexander Mitsos[c,a,d] Yue Guo[e] Qianxiao Li[e] Felix Dietrich[f] Manuel Dahmen[a,*] Ioannis G. Kevrekidis[g,*]

[a] Institute of Energy and Climate Research, Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, Jülich 52425, Germany
[b] RWTH Aachen University Aachen 52062, Germany
[c] JARA-ENERGY, Jülich 52425, Germany
[d] RWTH Aachen University, Process Systems Engineering (AVT.SVT), Aachen 52074, Germany
[e] Department of Mathematics, National University of Singapore, Singapore
[f] Department of Informatics, Technical University of Munich, Boltzmannstr. 3, 85748 Garching b. Munich, Germany
[g] Departments of Applied Mathematics and Statistics & Chemical and Biomolecular Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA

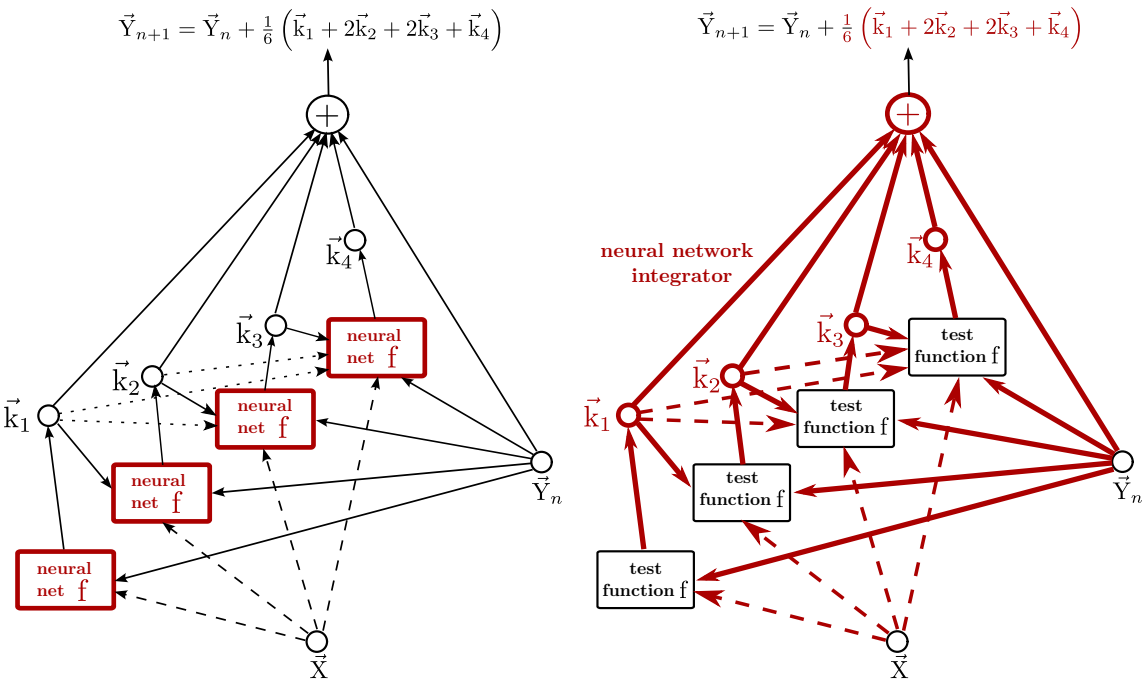
Abstract: Meta-learning of numerical algorithms for a given task consists of the data-driven identification and adaptation of an algorithmic structure and the associated hyperparameters. To limit the complexity of the meta-learning problem, neural architectures with a certain inductive bias towards favorable algorithmic structures can, and should, be used. We generalize our previously introduced Runge–Kutta neural network to a recursively recurrent neural network (R2N2) superstructure for the design of customized iterative algorithms. In contrast to off-the-shelf deep learning approaches, it features a distinct division into modules for generation of information and for the subsequent assembly of this information towards a solution. Local information in the form of a subspace is generated by subordinate, *inner*, iterations of recurrent function evaluations starting at the current *outer* iterate. The update to the next outer iterate is computed as a linear combination of these evaluations, reducing the residual in this space, and constitutes the output of the network. We demonstrate that regular training of the weight parameters inside the proposed superstructure on input/output data of various computational problem classes yields iterations similar to Krylov solvers for linear equation systems, Newton-Krylov solvers for nonlinear equation systems, and Runge–Kutta integrators for ordinary differential equations. Due to its modularity, the superstructure can be readily extended with functionalities needed to represent more general classes of iterative algorithms traditionally based on Taylor series expansions.

Keywords: *Numerical Analysis, Meta-Learning, Machine Learning, Runge-Kutta Methods, Newton-Krylov Solvers, Data-driven Algorithm Design*

1 Introduction

The relationship between a class of residual recurrent neural networks (RNN) and numerical integrators has been known since Rico-Martinez et al. (1992) proposed the architecture shown in Figure 1a for nonlinear system identification. Together with more recent work of the authors (Mitsos et al., 2018; Guo et al., 2022) it motivates the present work. The salient point of the RNN proposed by Rico-Martinez et al. (1992) is that it provides the structure of an integrator, in that case a fourth-order explicit Runge–Kutta (RK) scheme, within which the right-hand-side (RHS) of an ordinary differential equation (ODE) can be approximately learned from time series data. To this end, the integrator is essentially hard-wired in the forward pass of the RNN – a first instance of the body of work nowadays referred to as "neural ordinary differential equations" (Chen et al., 2018). However, upon closer inspection, the architecture reveals a complementary utility: When the model equations are known, it is the *structure and weights associated with an algorithm* that can be discovered, see Figure 1b. We study the latter setting in this work. Algorithms encoded by this architecture inherit two structural features from a RK scheme: i) they are based on recurrent function evaluations (inner recurrence), and ii) they compute iterates by adding a weighted sum of these inner evaluations that acts

*M. Dahmen, Institute of Energy and Climate Research, Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, Jülich 52425, Germany
E-mail: m.dahmen@fz-juelich.de
I. Kevrekidis, Departments of Applied Mathematics and Statistics & Chemical and Biomolecular Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA
E-mail: yannisk@jhu.edu



(a) When the architecture and integrator weights are hard-wired (black connections and their weights), a model of the unknown system (thick red squares) can be approximated, e.g., by multi-layer perceptrons.

(b) When the model equations are known (black squares), the structure and weights (thick red connections and their weights) associated with a solution algorithm can be discovered for that system.

Fig. 1. A recurrent neural network architecture templated on Runge–Kutta integrators, adapted from Rico-Martinez et al. (1992). The illustration shows the classical fourth-order Runge–Kutta scheme, where $\vec{Y}$ and $\vec{X}$ are the dependent and independent variables, respectively, f is the RHS of an ODE, and $\vec{k}_i$ are stage values. Recurrent function evaluations are computed internally, and the network itself can be applied iteratively via external recurrence. Given input and output data for training, a model (a) or an algorithm (here, an initial value solver) for that model (b) can be learned.

as *correction* to the input (outer recurrence). One critical observation is that such nested recurrent features also characterize (matrix)-free Krylov subspace methods such as restarted GMRES (Saad and Schultz, 1986) or Newton–Krylov–GMRES (Kelley, 1995), where finite differences (i.e., linear combinations) between recurrent function evaluations provide estimates of directional derivatives in various directions. This estimation of derivatives, or, more broadly, of Taylor series as in the RK case, forms the backbone of either of these two classical algorithm families, RK and Krylov methods. Observing these parallels, we conjecture that a recursively recurrent neural network (R2N2) can approximate many traditional numerical algorithms based on Taylor series expansions.

Returning to Figure 1b, we see that the (hyper-)parameters of the R2N2 are naturally linked to those of the algorithm it furnishes, e.g., weights connecting to the output can encode quadratures, and the internal recurrence count determines the number of function evaluations and the dimension of the subspace that is generated. It is this particular connectivity, highlighted in red, that is subject to discovery, whether it be encoding a Butcher tableau in the case of RK (Butcher, 2016; Guo et al., 2022) or finding the combination of weights that convert function evaluations into directional derivatives for Newton–Krylov (NK).

The question that motivates this work is thus: *Can meta-learning applied to the architecture and parameters of the R2N2 discover old and new algorithms "personalized" to certain problem classes?* We advocate that the R2N2 provides an architecture search space particularly suited for answering this question. The parameters of the R2N2 can be

partitioned *a priori* into those that are hard-wired and those subject to meta-learning. We can focus on the discrete features, i.e., determining the operations and connections that yield a best performant algorithm, or on the (continuous) weights, i.e., on how to best combine these operations. The former is also referred to as *algorithm configuration*, while the latter is a special case of *parameter tuning* (Hoos, 2011). Algorithm configuration has been optimized in a previous effort of the authors (Mitsos et al., 2018), albeit for a different, less expressive architecture with all weight parameters collapsed into a scalar stepsize. More recently, we worked with a fixed architecture as in Figure 1b such that Butcher tableaus personalized to specific classes of initial-value problems (IVPs) were learned (Guo et al., 2022). The present work seeks to extend this latter application to further numerical algorithms, in particular NK, and to thereby demonstrate that the R2N2 gives rise to a potent *superstructure* for optimal algorithm discovery. In follow-up work, we aim to show that jointly optimizing the weights and architecture from such a superstructure realizes learning of optimal iterative numerical algorithms, and to explore the use of approximate physical models as preconditioners within the superstructure.

1.1 Related work

Automated algorithm configuration and tuning dates back to Rice (1976) and studies meta-algorithms or meta-heuristics that pick the best algorithm from a set of (parametrized) algorithms for certain problem classes by manipulating the (hyper)parameters of a solver (Hoos, 2011). Speed-ups up to a factor of 50 have been demonstrated, e.g., for satisfiability problems (KhudaBukhsh et al., 2016) or mixed-integer problems (Hutter et al., 2010) by leveraging problem structure. This is in contrast to classical numerical algorithms that are designed to work with little or no *a priori* knowledge about the internal structure of the problem instances to be solved, and are often biased towards worst-case performance criteria (Gupta and Roughgarden, 2020). In contrast, algorithms adapted to specific problem classes typically incur a generalization weakness on other problems (Wolpert and Macready, 1997). Mitsos et al. (2018) modeled algorithms as feedback schemes with a cost ascribed to each operation, such that the design of iterative algorithms was posed as an optimal control problem of mixed-integer nonlinear (MINLP) type. They restricted operations to monomials of function evaluations and derivatives, thus specifying a family of algorithms. Depending on the analyzed problem, the procedure would recover known, established algorithms from this family, but also new algorithms that were optimal for the problems considered.

Mitsos et al. (2018) solved the algorithm generation MINLPs via the deterministic global solver BARON (Tawarmalani and Sahinidis, 2005). Possible gains of tailored algorithms must be weighed up against the required effort for finding them. Machine learning (ML) can decrease this effort. This has recently been demonstrated forcefully by Fawzi et al. (2022) who improved the best known algorithms on several instances of matrix multiplication, an NP-hard problem, and Mankowitz et al. (2023) who found new state-of-the-art sorting algorithms, both using deep reinforcement learning built on top of the AlphaZero framework (Silver et al., 2018). This landmark achievement is expected to spur new interest in algorithm discovery via ML. ML methods can optimize the performance of algorithms on a problem class implicitly given through data (Balcan, 2020; Gupta and Roughgarden, 2020). Such data-driven algorithm design has been applied to several computational problems, e.g., learning to solve graph-related problems (e.g., (Tang et al., 2020)), learning sorting algorithms (e.g., (Schwarzschild et al., 2021)), learning to branch (Khalil et al., 2016; Balcan et al., 2018), meta-learning optimizers (e.g., (Andrychowicz et al., 2016; Metz et al., 2020)), and our previous work of meta-learning RK integrators (Guo et al., 2022). In the context of this problem, the R2N2 defines a *superstructure* for a class of iterative algorithms. The notion of superstructure is used in many disciplines to denote a union of structures that are candidate solutions to a problem, e.g., in optimization-based flowsheet design within process systems engineering (Yeomans and Grossmann, 1999; Mencarelli et al., 2020). In general, optimizing a superstructure requires *integer* optimization techniques as in Mitsos et al. (2018). Given the neural network interpretation of the R2N2, however, (heuristic) methods for neural architecture search, e.g., (Elsken et al., 2019a; Li et al., 2021a), may be considered. Consequently, the optimal algorithmic procedure corresponds to an optimized neural architecture. In this work, we avoid integer optimization by essentially fixing the neural architecture for each respective numerical experiment, and optimizing the weights therein.

The R2N2 belongs to the class of recurrent neural networks (RNNs), which are a natural fit for iterative algorithms. For instance, RNN architectures

templated on RK integrators have been suggested several decades ago (Rico-Martinez et al., 1992; Rico-Martinez et al., 1994; Rico-Martinez et al., 1995; González-García et al., 1998). These architectures can be used for both, computing the outputs of an integrator, e.g., (Rico-Martinez et al., 1992; González-García et al., 1998) and identifying terms in a differential equation, e.g., (Rico-Martinez et al., 1994; Nascimento et al., 2020; Lovelett et al., 2020; Zhao and Mau, 2020; Goyal and Benner, 2021). Further, as the RK network of Rico-Martinez et al. (1992) learns the residual between the input and the output data, it constitutes the first occurrence of a *residual* network (ResNet, He et al. (2016)). Several newer ResNet-based architectures retain structural similarities with numerical methods (Lu et al., 2018). For instance, FractalNet (Larsson et al., 2017) resembles higher-order RK schemes in its macrostructure, as does the ResNet-derived architecture by Schwarzschild et al. (2021). Lu et al. (2018) proposed a ResNet augmented by a module derived from linear multistep methods (Butcher, 2016). Beyond architectural similarity with numerical methods, neural networks proposed in literature can also be designed such that the mathematical mapping they represent shares desirable properties with that of a method, e.g., symplectic mappings (Jin et al., 2020) for symplectic integrators or contracting layers (Chevalier et al., 2021) for fixed-point iterations. Dufera (2021) and Guo et al. (2022) trained networks to match the derivatives of ODEs or of their RK-based solution expansion at training points, respectively. Finally, direct learning of algorithms from neural network-like graphs has been proposed, e.g., by Tsitouras (2002), Denevi et al. (2018), Mishra (2018) and Venkataraman and Amos (2021).

Contributions

We build on the RK-NN of our previous work (Guo et al., 2022) and introduce the R2N2 superstructure for iterative numerical algorithms. The function to be evaluated inside the R2N2 architecture is itself explicitly given as part of the input problem instance, which is a major difference to operator networks that learn parameter-to-solution mappings like those in Lu et al. (2019) and Li et al. (2021b). Thus, higher-order iterative algorithms for equation solving and numerical integration, that are traditionally constructed through Taylor series expansion, can be approximated by the R2N2 superstructure. We demonstrate that both NK methods for solving systems of equations and RK methods for solving IVPs are encompassed by the proposed R2N2 as a joint superstructure. Further, in numerical experiments we show that the trained R2N2 can match, and sometimes improve upon, the iterations performed by NK and RK algorithms for a given number of function evaluations. A comparison of the R2N2 to GMRES on linear equation systems – which are the basic building block of iterative solvers for nonlinear systems – provides insight into the operations of the R2N2. In these experiments, our particular realization of this superstructure has a strong inductive bias that alleviates the need for certain configuration decisions, i.e., integer optimization, in algorithm design. The weights that remain to be trained correspond to coefficients or hyperparameters of the algorithms in question, and we tune these using PyTorch (Paszke et al., 2019).

The remainder of this article is structured as follows. In Section 2, we give a general problem definition for learning algorithms from task data and specify the problem for iterative algorithms. Section 3 introduces the R2N2 superstructure for learning iterative algorithms and shows its relation to steps of iterative equation solvers and integrators. Section 4 presents results of numerical experiments, where the R2N2 is trained to perform iterations of linear and nonlinear equation solvers and integrators. We summarize our results in Section 5 and discuss future research opportunities in Section 6.

2 Problem definition

Various generic problem formulations for learning algorithms from problem data exist in the literature, e.g., in Balcan (2020), Gupta and Roughgarden (2020), and our prior work (Guo et al., 2022). This section provides background and notation for a generic algorithm learning problem, Section 2.1, and specifies the problem formulation for learning iterative algorithms, Section 2.2.

2.1 Generic problem formulation

Let $\mathbf{x} \in \mathbb{R}^m$ and $\mathcal{F}$ a set of vectors such that $F \in \mathcal{F}$ is a problem instance composed of a continuous function $\boldsymbol{f} : \mathbb{R}^m \mapsto \mathbb{R}^m$ and some optional, additional problem parameters $\mathbf{p}$, e.g., time, such that we can write $F = (\boldsymbol{f}, \mathbf{p})$. Then, a traditional class of problems can be characterized by a functional Π acting on F and a point in $\mathbb{R}^m$. Further, a *solution* of F is any $\mathbf{x}^\star \in \mathbb{R}^m$ for which

$$\Pi(\mathbf{x}^\star, F) = 0. \tag{1}$$

This abstract form encompasses many problem types. The first type we consider here is finding the solution $\mathbf{x}^\star$ of algebraic equations, s.t

$$\Pi(\mathbf{x}^\star, F) = \|\boldsymbol{f}(\mathbf{x}^\star)\| = 0. \tag{2}$$

The second problem type is finding the solution of initial-value problems (IVPs) for a specific end time t, s.t.

$$\Pi(\mathbf{x}^\star, F) = \left\| \mathbf{y}_0 + \int_{\tau=t_0}^{\tau=t} \boldsymbol{f}(\mathbf{y}(\tau))d\tau - \mathbf{x}^\star \right\| = 0, \tag{3}$$

where $\boldsymbol{f}$ is the RHS of an ODE and $\mathbf{y}(\tau) \in \mathbb{R}^m$ is the state of the system at time τ with the initial value $\mathbf{y}_0 = \mathbf{y}(\tau = t_0)$. Equation (3) illustrates the definition of a problem instance and optional parameters $\mathbf{p}$, where $\mathbf{p} = (\mathbf{y}_0, t)$. The solution is the final value of the states, i.e., $\mathbf{x}^\star = \mathbf{y}(\tau = t)$.

We only consider problem instances F for which a solution $\mathbf{x}^\star$ exists. Then, we call any operator mapping $F \mapsto \mathbf{x}^\star$ a solver $\mathcal{S}(F)$. In general, we do not explicitly know these solvers and hence need to approximate their action by the design and use of numerical algorithms. That is, algorithms act as approximate solvers $\mathcal{A}(F, \boldsymbol{\theta}) \approx \mathcal{S}(F)$, parametrized by $\boldsymbol{\theta}$, such that

$$\Pi\left(\mathcal{A}(F, \boldsymbol{\theta}), F\right) \leq \delta,$$

where $\delta \in \mathbb{R}^+$ is sufficiently small and ideally user-defined. The challenge of designing numerical methods pertains to finding a structure linking mathematical operations parametrized by a set of parameters $\boldsymbol{\theta}$, which together yield performant solvers for problems that can be recast to Problem (1).

Due to the effort expended in designing algorithms, another important consideration is the range of applicability of the algorithm, i.e., the size of the set of problems it can solve. Traditional algorithms for problems like (2) and (3) often consider a certain worst-case performance in a given problem class $\mathcal{F}$. In contrast, we are interested in the average/expected performance of algorithms over a specific problem class, which is defined by a distribution μ over $\mathcal{F}$. Then, finding performant, or even optimal, algorithms for such a class of problems requires solving

$$\min_{\boldsymbol{\theta}} \mathbb{E}_\mu \left[\|\Pi\left(\mathcal{A}(F, \boldsymbol{\theta}), F\right)\|\right] + R(\mathcal{A}(F, \boldsymbol{\theta}), \boldsymbol{\theta}). \tag{4}$$

Problem (4) seeks parameters $\boldsymbol{\theta}$ which minimize the expectation of the residual norm of solutions computed to Equation (1) using $\mathcal{A}(F, \boldsymbol{\theta})$ over problem instances distributed according to μ. The second term in (4), $R(\mathcal{A}(F, \boldsymbol{\theta}), \boldsymbol{\theta})$, is reserved for some additional regularization penalty that can promote certain properties in the approximate solution $\mathcal{A}(\cdot, \boldsymbol{\theta}^\star)$, such as a desired convergence order (Guo et al., 2022). The regularization can also penalize the parameter values $\boldsymbol{\theta}$ directly, e.g., in L_2 regularization. The algorithmic structure itself can be described by discrete variables, e.g., to indicate whether an operation or module exists in the algorithm. However, to include such discrete variables one requires a metric that determines which structure is optimal. This is typically assessed over a prolonged number of iterations and requires integer optimization techniques, see, e.g., Mitsos et al. (2018). In this work, the goal is rather to demonstrate that the iterations of several iterative algorithms have a common superstructure. Consequently, we focus on tuning the parameters of this superstructure towards different problem classes. Thus, Problem (4) resembles a regular multi-task learning problem in the context of statistical learning (Baxter, 2000), where *tasks* are equated with problem instances F. To handle such a problem, the expectation term can be approximated by drawing samples of task data from μ and minimizing some loss function for them.

2.2 Iterative numerical algorithms

We focus on iterative algorithms, which construct a sequence of iterates $\{\mathbf{x}_k\}$ approaching a solution of F. Starting with the initial point $\mathbf{x}_0$, the algorithm computes new iterates by applying

$$\mathbf{x}_{k+1} = \mathcal{A}^{iter}(F, \boldsymbol{\theta}; \mathbf{x}_k), \tag{5}$$

such that the sequence ideally converges to some $\hat{\mathbf{x}}$:

$$\hat{\mathbf{x}} = \lim_{k \to \infty} \mathbf{x}_k.$$

If we have $\Pi(\hat{\mathbf{x}}, F) = 0$, then $\mathcal{A}^{iter}$ is convergent to the solution of F.

In the following, we restrict the possible realizations of $\mathcal{A}^{iter}$ strongly by narrowing our attention to iterative algorithms that apply *additive* step updates in a generalized Krylov-type subspace $\mathcal{K}_n$. This subspace is spanned by vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}$, i.e.,

$$\mathcal{K}_n(\boldsymbol{f}, \mathbf{x}) = \text{span}\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}, \tag{6}$$

that are generated by recurrent function evaluations at $\mathbf{x}_k$, i.e.,

$$\mathbf{v}_0 := \boldsymbol{f}\left(\mathbf{x}_k\right), \tag{7a}$$

$$\mathbf{v}_j = \boldsymbol{f}\left(\mathbf{x}_k + \sum_{l=0}^{j-1} b_{jl}\mathbf{v}_l\right), \quad j = 1, \ldots, n-1, \tag{7b}$$

for some $b_{jl} \in \mathbb{R}$. The next iterate $\mathbf{x}_{k+1}$ is computed by adding a linear combination of this basis to $\mathbf{x}_k$, i.e.,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \sum_{j=0}^{n-1} c_j \mathbf{v}_j, \tag{8}$$

for some $c_j \in \mathbb{R}$. Equation (7b) describes an *inner* iteration of the algorithm, while Equation (8) constitutes an *outer* iteration. Together, Equations (7)—(8) define a common *superstructure* for iterative algorithms. This structure has parallels to Krylov subspace methods (Saad, 2003), and RK methods (Butcher, 2016), respectively, see Section 3.2.

Considering only iterative algorithms within this superstructure has several advantages compared to alternatives that rely on deep learning with heavily parametrized models. First, the iterative nature of the superstructure greatly reduces the total amount of parameters needed to learn solution procedures. Second, Equations (7) and (8) eliminate most of the functional forms admissible under Equation (5). And third, directly embedding $\boldsymbol{f}$ in the superstructure avoids the need to learn an extra representation of the problem within the solver mapping. The resulting superstructure resembles an RNN, and automatic differentiation frameworks for the training of neural networks such as PyTorch (Paszke et al., 2019) can be used to determine the remaining free parameters $\boldsymbol{\theta}$.

3 R2N2 superstructure for iterative algorithms

We recently proposed the RK-NN, a neural network architecture templated on RK integrators, to *personalize* coefficients of a RK method to a specific problem class (Guo et al., 2022). In this work, we extend our view of the RK-NN to that of a more general superstructure for iterative numerical algorithms, the R2N2, that is applicable to different problem classes such as equation solving and numerical integration. Section 3.1 describes the architecture of the R2N2. Each forward pass through the R2N2 is interpreted as an iteration of a numerical algorithm (*outer* recurrence) that invokes one or many recurrent function evaluations (*inner* recurrence), starting at the current iterate, to compute the next iterate. The R2N2 is equivalent to the RK-NN from Guo et al. (2022) for the specific case of minimizing empirical risk (Problem (4)) for classes of IVPs, Problem (3). However, as a small addition to the original RK-NN, the R2N2 superstructure now allows presetting various routines for function evaluation, see supplementary material (**SM1**). In Section 3.2, we show that the applicability of the R2N2 as a superstructure is substantially extended beyond the case of solving IVPs. Section 3.3 and Section 3.4 conclude this section with remarks about training and implementation of the R2N2 superstructure.

3.1 Neural architecture underlying the R2N2 superstructure

The proposed R2N2 superstructure is portrayed by Figure 2. It represents the computation of one iteration of a numerical algorithm, i.e., the mathematical function defined by the RNN architecture can substitute for $\mathcal{A}^{iter}(F, \boldsymbol{\theta}; \mathbf{x}_k)$ in Equation (5), where a problem instance F contains $\boldsymbol{f}$ and $\mathbf{p}$. Each step requires the current iterate $\mathbf{x}_k$ as an input, where the initial $\mathbf{x}_0$ is typically supplied within $\mathbf{p}$. Further, a function $\boldsymbol{f} : \mathbb{R}^m \mapsto \mathbb{R}^m$ is prescribed externally as part of a task, but remains unchanged for all iterates. Finally, a parameter h for scaling of the layer computations is part of the input to the superstructure. For some problem classes, we choose h according to problem parameters $\mathbf{p}$, e.g., the timestep in integration. Whenever h is not specified, assume $h = 1$, i.e., no scaling. The initial layer, which is the left-most in Figure 2, is always a direct function evaluation at the current iterate, $\boldsymbol{f}(\mathbf{x}_k)$, i.e., we have $\mathbf{v}_0 = \boldsymbol{f}(\mathbf{x}_k)$. The remaining $n-1$ layers of the superstructure each output a $\mathbf{v}_j$, $j = 1, \ldots, n-1$ by applying a composition of $\boldsymbol{f}$ and $\boldsymbol{N}_j$. $\boldsymbol{N}_j$ linearly combines its inputs $\mathbf{v}_0, \ldots, \mathbf{v}_{j-1}$ using trainable parameters $\boldsymbol{\theta}_j$ and, scales the term with h and adds it to $\mathbf{x}_k$ to provide $\mathbf{x}'_j$, the input to $\boldsymbol{f}$ in the j-th layer:

$$\mathbf{x}'_j = \boldsymbol{N}_j\left(\mathbf{x}_k; \mathbf{v}_0, \ldots, \mathbf{v}_{j-1}; h\right) = \mathbf{x}_k + h \sum_{l=0}^{j-1} \theta_{j,l} \mathbf{v}_l \tag{9}$$

The $\{\mathbf{v}_j\}$, including $\mathbf{v}_0$, span an n-dimensional subspace in which the output layer $\boldsymbol{N}_n$ computes the next iterate $\mathbf{x}_{k+1}$ using the trainable parameters $\boldsymbol{\theta}_n$, i.e.,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h \sum_{j=0}^{n-1} \theta_{n,j} \mathbf{v}_j. \tag{10}$$

This completes one forward pass through the R2N2 superstructure. Note that parameters $\boldsymbol{\theta}$ are partitioned into $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n-1}, \boldsymbol{\theta}_n\}$ and that we can identify $h\theta_{j,l}$ with b_{jl} from Equation (7b) for $j =$

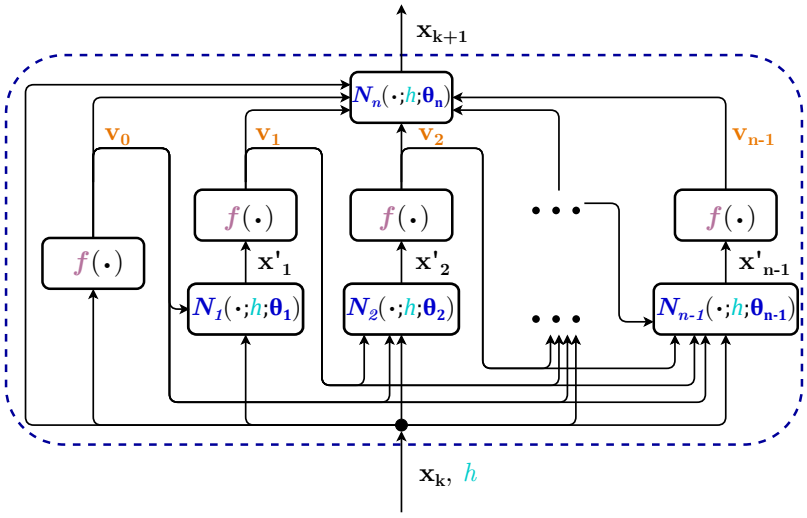


Fig. 2. Proposed recursively recurrent neural network-based superstructure of an iterative algorithm. The recurrent cell is delimited by the dashed blue line and computes $\mathbf{x}_{k+1}$ as a function of the current iterate $\mathbf{x}_k$, a scaling parameter h (cyan), and a function $\boldsymbol{f}$ (magenta). Trainable weights $\boldsymbol{\theta}_j$ are contained within each of the layer modules $\boldsymbol{N}_j$ with $j = 1, 2, \ldots, n-1$. $\boldsymbol{\theta}_n$ denote the trainable weights of the output module $\boldsymbol{N}_n$ (all in blue). $\mathbf{x}_k$ has a skip connection to each layer and to the output module. Intermediate function arguments, i.e., inputs to $\boldsymbol{f}$ in layer j, are called $\mathbf{x}'_j$ and subspace members, i.e., outputs of $\boldsymbol{f}$ in layer j, are denoted by $\mathbf{v}_j$ (orange).

$1, \ldots, n-1$ and $\theta_{n,j}$ with c_j from Equation (8). Figure 3 shows the computation of multiple consecutive iterations with the proposed R2N2 superstructure.

3.2 Relation to Krylov subspace methods

Numerical algorithms for solving (non-)linear equations as well as integrating sets of ordinary differential equations are traditionally based on local Taylor series expansions, and on the use of the first term (the Jacobian) – or even sometimes the second term (the Hessian) – in performing the numerical operations required to arrive at the next iteration. The solution of sets of linear equations resulting from sets of nonlinear equations, e.g., Newton iterations involving the solutions of linear equations, underpins a lot of today's scientific computing. Due to the memory constraints of the early supercomputers (like the Cray 1), algorithms solving sets of linear equations through matrix-vector products and the construction of Krylov subspaces, e.g., GMRES, blossomed in the 1970s and 80s. Most importantly for us, these ideas evolved into *matrix-free* algorithms, like those embodied in matrix-free NK–GMRES. And through such ideas, algorithms evolved towards performing their tasks through intelligently planned *recursive function evaluations*. This leads to the premise that many scientific computations can be performed through systematic, recursive function evaluations (interspersed by brief low-dimensional tasks, like Gram-Schmidt orthogonalization, or least-squares solutions in low-dimensional subspaces) – and thus many algorithms essentially comprise of an intelligent (recursive) concatenation of function evaluations, enhanced by some ancillary computation. Thus, many traditional numerical algorithms *are* protocols for recursively recurrent function evaluations (plus ancillary computation). Both matrix-free NK–GMRES (for solving systems of nonlinear equations) and numerical initial-value problem solvers (of which RK is a notorious example) can be seen to naturally lead to R2N2 architectures. The analogy between an RK method and the R2N2 was already demonstrated in our previous work (Guo et al., 2022). Here, we show that the R2N2 also represents Krylov and NK subspace methods up to the ancillary computations.

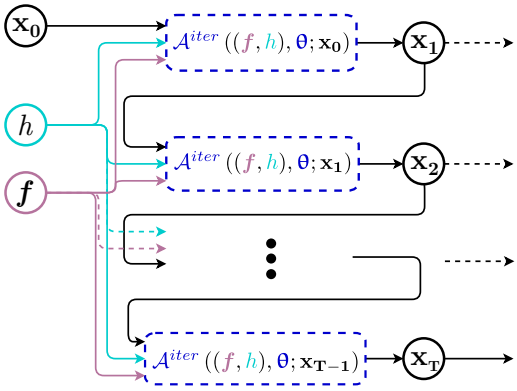


Fig. 3. T consecutive iterations using the R2N2 superstructure from Figure 2 compute a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$. The blue dashed cell delimits the iteration $\mathcal{A}^{iter}$ computed in a forward pass, i.e., one pass through the R2N2. The task-related inputs h (cyan) and $\boldsymbol{f}$ (magenta) apply to each iteration. The initial guess $\mathbf{x}_0$ is an input only to the first iteration.

3.2.1 Krylov subspace solvers

Krylov subspace solvers compute an approximate solution to linear systems of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{11}$$

where $\mathbf{A} \in \mathbb{R}^{m\times m}$ and $\mathbf{b} \in \mathbb{R}^m$, in an n-dimensional subspace $\mathcal{K}_n$, $n \leq m$, such that the norm of a residual $\mathbf{r}$, i.e.,

$$\|\mathbf{r}\| = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|,$$

is minimized (Saad, 2003). The Krylov subspace $\mathcal{K}_n$ is given by

$$\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0) = \text{span}\left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{n-1}\mathbf{r}_0\right\}, \tag{12}$$

and is fully determined by $\mathbf{A}$ and the initial residual $\mathbf{r}_0$. In absence of prior knowledge, $\mathbf{r}_0$ is usually computed by choosing $\mathbf{x}_0 = \mathbf{0}$, i.e., $\mathbf{r}_0 = -\mathbf{b}$. The remaining Krylov vectors spanning the subspace are obtained by recurrent left-side multiplication of $\mathbf{r}_0$ with $\mathbf{A}$. Finally, the approximate solution follows as

$$\overline{\mathbf{x}} = \mathbf{x}_0 + \mathbf{q}^\star, \tag{13}$$

$$\mathbf{q}^\star \in \arg\min_{\mathbf{q}\in\mathcal{K}_n} \|\mathbf{A}\left(\mathbf{x}_0 + \mathbf{q}\right) - \mathbf{b}\|. \tag{14}$$

$\mathbf{q}^\star$ can be expressed as a linear combination of the vectors spanning the subspace $\mathcal{K}_n$, i.e,

$$\mathbf{q}^\star = \mathbf{V}_n\boldsymbol{\beta}, \tag{15}$$

where $\mathbf{V}_n$ is an $n \times n$ matrix that contains the members of $\mathcal{K}_n$ as columns and $\boldsymbol{\beta} \in \mathbb{R}^n$ is a vector of coefficients found by *solving* Problem (14). Contemporary Krylov methods like GMRES (Saad and Schultz, 1986) orthonormalize the basis of $\mathcal{K}_n(\mathbf{A}, \mathbf{r}_0)$. This operation requires additional computation but enables explicit residual minimization.

Instead of solving Equation (14), the R2N2 superstructure approximates the solutions generated by the Krylov method, Equations (12) – (15), as follows. We set $\boldsymbol{f}(\mathbf{x}) := \mathbf{A}\mathbf{x} - \mathbf{b}$ such that the initial function evaluation returns $\mathbf{v}_0 = \boldsymbol{f}(\mathbf{0}) = \mathbf{r}_0 = -\mathbf{b}$. Due to the nature of the layer modules $\boldsymbol{N}_j$, Equation (9), all layers after the initial layer will return outputs $\mathbf{v}_j$ that are in the span of $\left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \ldots, \mathbf{A}^{j-1}\mathbf{r}_0\right\}$ such that the subspace generated by the R2N2 coincides with the Krylov subspace, Equation (12). The output module of the R2N2, Equation (10), learns a fixed linear combination of these $\mathbf{v}_j$ through its parameters $\boldsymbol{\theta}_n$. By identifying these $\boldsymbol{\theta}_n$ as $\boldsymbol{\beta}$, $\mathbf{x}_k$ as $\mathbf{x}_0$ and $h = 1$, one forward pass through the R2N2 can, in principle, imitate one outer iteration of the Krylov subspace method for a single problem instance $(\mathbf{A}, \mathbf{b})$.

On the other hand, one pass through the R2N2 is cheaper than an iteration of a Krylov method, since the R2N2 does not perform the orthonormalization and explicit residual minimization. Both methods require $n-1$ matrix-vector products to build the n-dimensional subspace, given that $\mathbf{r}_0$ is obtained for free. Finally, we point out that some Krylov-based solvers, e.g., GMRES, can be iteratively restarted to compute a better approximation to a solution of a linear system (Saad and Schultz, 1986). This restarting procedure is naturally represented by recurrent passes through the R2N2, i.e., a restart corresponds to updating the iterate from $\mathbf{x}_k$ to $\mathbf{x}_{k+1}$.

3.2.2 Newton-Krylov solvers

Newton-Krylov solvers essentially approximate Newton iterations for the solution of a nonlinear equation system $\boldsymbol{f}(\mathbf{x}) = \mathbf{0}$, $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$, where the linear subproblem that arises in each Newton iteration is addressed using a Krylov subspace method (Kelley, 1995, 2003; Knoll and Keyes, 2004). The k-th linear subproblem requires solving

$$\mathbf{J}(\mathbf{x}_k)\Delta\mathbf{x}_k = -\boldsymbol{f}(\mathbf{x}_k).$$

Therefore, the residual $\mathbf{r}_0$ of the linear solver corresponds to $\boldsymbol{f}(\mathbf{x}_k)$ and the matrix $\mathbf{A}$ is substituted by the Jacobian of $\boldsymbol{f}$ at $\mathbf{x}_k$, $\mathbf{J}(\mathbf{x}_k) \in \mathbb{R}^{m\times m}$. The corresponding k-th Krylov subspace then reads

$$\mathcal{K}_n^{(k)}\left(\mathbf{J}(\mathbf{x}_k), \boldsymbol{f}(\mathbf{x}_k)\right) = \text{span}\left\{\mathbf{v}_0, \ldots, \mathbf{v}_{n-1},\right\}, \tag{16}$$

$$\mathbf{v}_j = \mathbf{J}(\mathbf{x}_k)^j \boldsymbol{f}(\mathbf{x}_k) \quad \forall\, j \in \{0, \dots, n-1\}.$$

In the practical use-case of Newton-Krylov methods, $\mathbf{J}(\mathbf{x}_k)$ is not computed explicitly. Instead, matrix-vector products $\mathbf{J}(\mathbf{x}_k)\mathbf{z}$, where $\mathbf{z} \in \mathbb{R}^m$ is a vector, are approximated using a directional derivative of $\boldsymbol{f}$ at $\mathbf{x}_k$ (Kelley, 2003), i.e.,

$$\mathbf{J}(\mathbf{x}_k)\mathbf{z} \approx \frac{\boldsymbol{f}(\mathbf{x}_k + \epsilon\mathbf{z}) - \boldsymbol{f}(\mathbf{x}_k)}{\epsilon},$$

where ϵ is a small value in the order of 10^{-8}. The RHS corresponds to forward-differencing, which is shown to lie within the R2N2 superstructure in the supplementary material (**SM1**). Typically, multiple iterates $\mathbf{x}_k$ have to be computed to sufficiently approximate a solution of $\boldsymbol{f}(\mathbf{x}) = \mathbf{0}$. Such iterative behavior can be reproduced by performing multiple recurrent passes through the R2N2. Since $\mathbf{J}(\mathbf{x}_k)$ is always computed with the current iterate $\mathbf{x}_k$, each pass through the R2N2 necessarily corresponds to a new Newton iteration if the remaining identities noted in the previous section, Section 3.2.1, on linear Krylov solvers are applied again. Consequently, the limited expressivity due to $\boldsymbol{\theta}_n$ as opposed to the minimization in a Krylov method is inherited, too.

3.3 Training the R2N2 superstructure

Training the R2N2 superstructure implies solving Problem (4) for the trainable parameters $\boldsymbol{\theta}$. The input data for each problem class is sampled from a distribution representing a set of problem instances, the solutions to which are the training targets. In the following, we indicate data samples by an additional subscript $i = 1, \dots, N$, where N is the total number of samples, i.e., $\mathbf{x}_{i,k}$ refers to the i-th sample after the k-th iteration. The output of the k-th pass through an R2N2 is the iterate denoted by $\hat{\mathbf{x}}_{i,k+1}$. As a loss function we use a weighted version of the mean squared error (MSE) between $\hat{\mathbf{x}}_{i,k}$ and $\mathbf{x}_{i,k}^{target}$ or the corresponding residual $\Pi\left(\hat{\mathbf{x}}_{i,k}, F_i\right)$, summed over all iterations, i.e.,

$$MSE_x = \frac{1}{N}\sum_{k=1}^{T}\sum_{i=1}^{N} w_{i,k}\left\|\hat{\mathbf{x}}_{i,k} - \mathbf{x}_{i,k}^{target}\right\|_2^2, \quad (17a)$$

$$MSE_\Pi = \frac{1}{N}\sum_{k=1}^{T}\sum_{i=1}^{N} w_{i,k}\Pi(\hat{\mathbf{x}}_{i,k}, F_i)^2, \quad (17b)$$

where $w_{i,k}$ are the weights belonging to the i-th sample in the k-th iteration. Whether we utilize Equation (17a) or Equation (17b) is problem-specific. For instance, for equation solvers, we can make use of $\boldsymbol{f}(\mathbf{x}_i^\star) := \mathbf{0}$ to compute the residual for Equation (17b) from $\boldsymbol{f}(\mathbf{x}_{i,k+1})$. For integrators on the other hand, we can sample training targets $\mathbf{x}_{i,k+1}^{target}$ from the trajectory computed by some high-order integrator or, if available, use an analytic solution to generate target data. We do not use any regularizers for training the R2N2 in this work.

3.4 Implementation

We implemented the R2N2 superstructure in PyTorch (version 1.8.0) (Paszke et al., 2019). We trained on an Intel i7-9700K CPU using *Adam* (Kingma and Ba, 2015) for 25,000 epochs with the default learning rate of 0.001 or L-BFGS (Liu and Nocedal, 1989) for 5,000 epochs with a learning rate of 0.01. Note that L-BFGS is commonly used to fine-tune networks with comparable architecture as ours that were pretrained with *Adam*, e.g., by Zhao and Mau (2020), suggesting that L-BFGS can improve the training result. A detailed assessment of the capability of different optimizers and strategies to train the R2N2 superstructure is beyond the scope of this work.

4 Numerical experiments

In this section, we demonstrate the ability of the R2N2 superstructure to learn efficient iterations for both equation solvers and integrators. Section 4.2 and Section 4.3 will show that computational benefits of solvers trained for a specific problem class start to arise in solving *nonlinear* problems with such algorithms. In particular, Section 4.2 demonstrates the extended applicability of the RK-NN introduced in Guo et al. (2022) to *nonlinear* equation systems. We start here, however, with *linear* systems of equations in Section 4.1, not because of the computational benefits achievable, but because the comparison between the R2N2 and GMRES (as a matrix-free linear algebra solver) facilitates initial insight into the functionality of the R2N2 when learning iterative solvers for equation systems.

In all experiments that follow, the R2N2 and the classical method it is compared to are allowed the same number of function evaluations per iteration. Thus, for equation solvers, the R2N2 requires less overall operations per iteration, c.f. Section 3.2, and for integrators, the amount of overall operations per iteration will be identical. We have consistently used 70% of the generated data for training and an independent sample of 30% for the test results that are presented in the following.

4.1 Solving linear equation systems

First, we study the solution of linear equations, see Equation (11). For the illustrative experiments we consider examples where a single task is given by $F_i = (\mathbf{A}_1, \mathbf{b}_i)$ with $\mathbf{A}_1 \in \mathbb{R}^{m\times m}$, $\mathbf{b}_i \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^m$ and $m = 5$. $\mathbf{A}_1$ is a fixed, randomly-generated symmetric positive definite matrix overlaid with an additional boost to its diagonal entries to influence its spectrum. Right-hand sides (RHS) $\mathbf{b}_i$ are sampled uniformly around a fixed randomly-chosen mean. See supplementary material (**SM2**) for details. As training input we use the data set of tasks $\{F_i\} = \{(\mathbf{A}_1, \mathbf{b}_i)\}$ and always select $\mathbf{x}_0 = \mathbf{0}$ as the initial point. Therefore, the resulting n-dimensional Krylov subspace is always spanned by $\left\{\mathbf{b}_i, \mathbf{A}_1\mathbf{b}_i, \ldots, \mathbf{A}_1^{n-1}\mathbf{b}_i\right\}$. When injecting the input data to the R2N2, we resort to $\boldsymbol{f}(\mathbf{x}_i) := \mathbf{A}_1\mathbf{x}_i - \mathbf{b}_i$. We use $\boldsymbol{f}\left(\mathbf{x}_{i,k+1}^t\right) = \mathbf{0}$ as training targets for all samples i and all steps k. Therefore, the training loss at a step k, derived from Equation (17b), becomes

$$MSE_{f,k} = \sum_{k=1}^{T} w_k \sum_{i=1}^{N} \left(\mathbf{A}_1\hat{\mathbf{x}}_{i,k} - \mathbf{b}_i\right)^2, \qquad (18)$$

where w_k becomes relevant when training over multiple iterations, $T > 1$, and is tuned by hand.

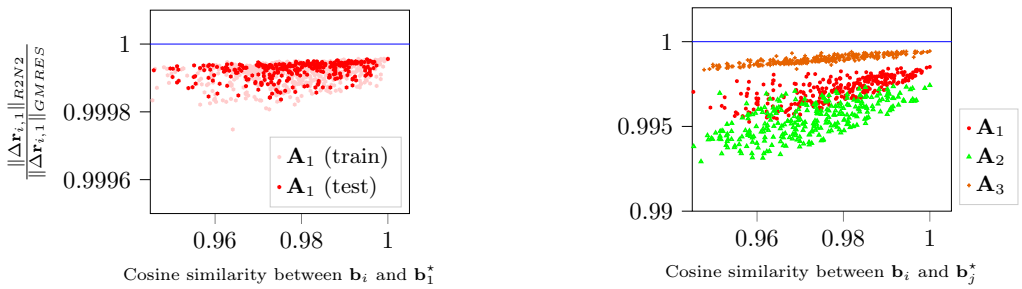
To evaluate the performance of the R2N2, we compute the reduction of the norm of the residual that was achieved after k iterations through the R2N2, i.e.,

$$\Delta\mathbf{r}_{i,k} = \|\mathbf{b}_i\| - \|\mathbf{A}_1\hat{\mathbf{x}}_{i,k} - \mathbf{b}_i\|. \qquad (19)$$

For the results that follow, we compare the R2N2 to the SciPy implementation of the solver GMRES (Saad and Schultz, 1986; Virtanen et al., 2020). GMRES is set to use the same number of inner iterations, i.e., function evaluations as the R2N2. Considering the restarted version of GMRES, GMRES(r), a forward pass through the R2N2 represents one outer iteration (Saad and Schultz, 1986). We evaluate the reduction in residual norm divided by the one achieved using GMRES, i.e., $\frac{\|\Delta\mathbf{r}_{1,i}\|_{R2N2}}{\|\Delta\mathbf{r}_{1,i}\|_{GMRES}}$, with the subscripts 'R2N2' indicating the R2N2 and 'GMRES' indicating the solver GMRES.

The training result of this first experiment is shown in Figure 4a. Notably, the performance of the R2N2 is upper-bounded by the performance of GMRES, as GMRES minimizes the residual in the subspace spanned by the Krylov vectors. Given that this subspace is invariant with respect to the operations that can be learned by the layers of the R2N2 for the linear problem instances, the R2N2 cannot improve on the performance of GMRES in this first case study. The experiment with a fixed $\mathbf{A}_1$ was chosen to illustrate this upper bound of the R2N2 and does not yield a proper problem distribution for the training set – with $\mathbf{A}_1$ fixed, only a mapping from $\mathbf{b}_i$ to $\mathbf{x}_i^\star$ needs to be learned. Thus, we now draw two additional matrices, $\mathbf{A}_2, \mathbf{A}_3 \in \mathbb{R}^{m\times m}$ from the distribution (see (**SM2**)). Through the addition of $\mathbf{A}_2$ and $\mathbf{A}_3$, the performance, i.e., the adaptation, of the resulting R2N2 on problem instances formed with either of the three matrices is decreased compared to the previous case (see Figure 4b).

Next, we study the performance of the R2N2 over multiple iterations together with the extrapolation capabilities of the R2N2. Here, the R2N2 is compared with the restarted version of GMRES, *GMRES(r)*, where each iteration uses the output of the previous iteration as an initial value. We train the R2N2 to minimize loss after three outer iterations, given by Equation (18). We set $w_k = 4^k$, which was found to yield decent results. The training dataset is again generated from the random draw of samples of $\mathbf{b}_i$ combined with the three matrices $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$. The solid red line in Figure 5 shows the convergence of the R2N2 by plotting the average residual of the test set for $\mathbf{A}_1$ only (for clarity). The R2N2 reduces the residual over all consecutive outer iterations, i.e., even beyond the third outer iteration, which indicates a first type of successful extrapolation of the R2N2: Iterates $\mathbf{x}_k$ for $k > 3$ have not been contained by the training input distribution, yet the R2N2 iterations progress towards the solution of the problem beyond that point. Related to this extrapolation on RHS given by iterates $\mathbf{x}_k$ is the orange dash-dotted line that examines the R2N2 on uniform random RHSs $\mathbf{b}_i$, normalized to the length of the test set samples. The R2N2 is shown to converge to a solution for *all* of these RHSs, i.e., the convergence is due to a property of the R2N2 and $\mathbf{A}_1$. We analyze this further in supplementary material (**SM4**). Finally, we analyze two additional types of extrapolation applied to $\mathbf{A}_1$: i) raising the noise level of its random component by up to a factor of 7, ii) reducing or increasing the induced spectrum in $\mathbf{A}_1$, respectively. See supplementary material (**SM3.1**) for details. Trajectories for the resulting test matrices – again combined with the initial RHS samples of $\mathbf{b}_i$ – are plotted as light-red dotted lines in Figure 5. The results demonstrate that the R2N2 learns a solver



(a) Training and testing with only $\mathbf{A}_1$. Light dots show training samples.

(b) Testing on instances featuring $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$.

Fig. 4. Relative performance of R2N2 vs GMRES with $n = 4$ inner iterations for problem instances of Problem (11). Each dot indicates a different RHS $\mathbf{b}_i$. The blue lines indicate the baseline achieved by GMRES. Subscripts 'R2N2' and 'GMRES' stand for the R2N2 and GMRES, respectively. Each method returns a step in a 4-dimensional subspace, which is generated via 3 function evaluations. The residual reduction is computed using Equation (19) with the respective $\mathbf{A}_j$ for $j \in \{1, 2, 3\}$. The scales in a) and b) are different. By $\mathbf{b}_j^\star$ we denote the sample for which the relative performance of R2N2 is maximized for each case. In (a), the training data set features only problems with $\mathbf{A}_1$, while in (b), three different matrices are used.

adapted to a problem data set, and, further, that the R2N2 extrapolates reasonably well beyond that data set, with a performance decrease as $\mathbf{A}$ becomes more different from the training distribution. The extrapolation experiments are presented in more detail – including some failure cases – in (**SM3.1**). We also show that vanilla neural networks (NNs) of comparable size cannot learn a solver like the R2N2 (**SM3.2**), and that a R2N2 modified to *predict* solutions does better than the NNs in this task too (**SM3.3**).

As a final example, we demonstrate that the R2N2 can also be applied to an embedding of the problem class. We generate a random 15-dimensional orthonormal matrix $\mathbf{Q}$ from the Haar distribution via SciPy (Virtanen et al., 2020; Mezzadri, 2006), and consider the problem class with instances

$$\mathbf{Q}\mathbf{A}_1\mathbf{Q}^T\mathbf{x} = \mathbf{Q}\mathbf{b}_i, \tag{20}$$

where by abuse of notation $\mathbf{A}_1$ and $\mathbf{b}_i$ are made 15-dimensional by zero-padding the new dimensions. The convergence of the residual of this embedded problem is shown in Figure 6. Evidently, as an adapted solver, the R2N2 is also applicable to this embedding of the problem class it was trained on. By inspecting the embedding, we see that the resulting subspace vectors are also rotated by $\mathbf{Q}$, i.e.,

$$\mathcal{K}_n\left(\mathbf{Q}\mathbf{A}\mathbf{Q}^T, \mathbf{Q}\mathbf{b}\right) = \left(\mathbf{Q}\mathbf{b}, \mathbf{Q}\mathbf{A}\mathbf{b}, \ldots, \mathbf{Q}\mathbf{A}^{n-1}\mathbf{b}\right),$$

as the $\mathbf{Q}^T\mathbf{Q}$ in the inner of the matrix-vector products cancels out each time.

4.2 Solving nonlinear equation systems

As an illustrative nonlinear equation system, we consider Chandrasekhar's H-function in conservative form, an example that was extensively used in Kelley's book on Newton-Krylov solvers (Kelley, 2003). The discretized form of this equation reads

$$\left(\boldsymbol{f}(\mathbf{x})\right)_j = \mathbf{x}_j - \left(1 - (\mathbf{A}_c\mathbf{x})_j\right)^{-1}, \tag{21}$$

for $j = 1, \ldots, m$ with the entries of parametric matrix $\mathbf{A}_c \in \mathbb{R}^{m\times m}$ defined by

$$A_{c,j\iota} = \frac{c\mu_j}{2m\left(\mu_j + \mu_\iota\right)},$$

with $\mu_\iota = \frac{(\iota - \frac{1}{2})}{m}$ and $\iota = 1, \ldots, m$ also indexing the discretization points. The derivation of the equations is given in supplementary material (**SM2**). We combine problems with $m = 10$ and $m = 20$ discretization points and $c \in \{0.875, 0.905, 0.935\}$ to span a training set. For all of these problems, we sample initial values $\mathbf{x}_{0,i}$ from a normal distribution with mean $\mathbb{1}^{(m)}$, where $\mathbb{1}^{(m)}$ is an m-dimensional vector of ones, and variance $0.2 \cdot \mathbb{1}^{(m)}$. We again use (18) as a loss function, where targets are $\boldsymbol{f}\left(\mathbf{x}_{k+1,i}^t\right) = 0$ for all samples i and iterations k. The SciPy implementation of Kelley's Newton-Krylov GMRES (NK-GMRES) serves as a benchmark (Virtanen et al., 2020; Kelley, 2003).

We evaluate the performance after k nonlinear iterations by the reduction in the norm of the resid-

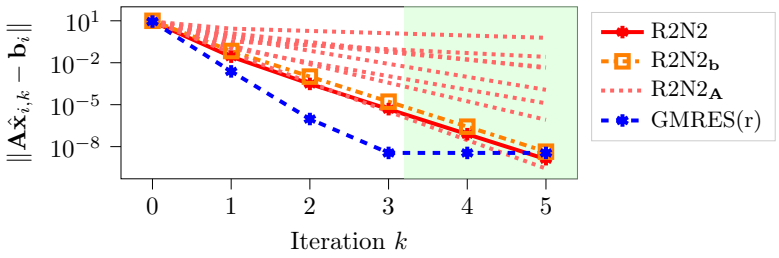


Fig. 5. Convergence of the R2N2 vs GMRES(r) for solving (11) for multiple samples of $\mathbf{b}_i$. The lines connect the sample mean at each (outer) iteration k. Both solvers use $n = 4$ inner iterations. The R2N2 was trained on $T = 3$ outer iterations, using Equation (18) with $w_k = 4^k$ as a loss, and then applied over 5 iterations. Iterations within the shaded square area, $k > 3$, have not been seen in training and indicate one type of extrapolation. The orange-squared line 'R2N2$_\mathbf{b}$' shows extrapolation in the RHS, and the dotted lines 'R2N2$_\mathbf{A}$' show extrapolations in matrices $\mathbf{A}$ (see (**SM3.1**) for details).

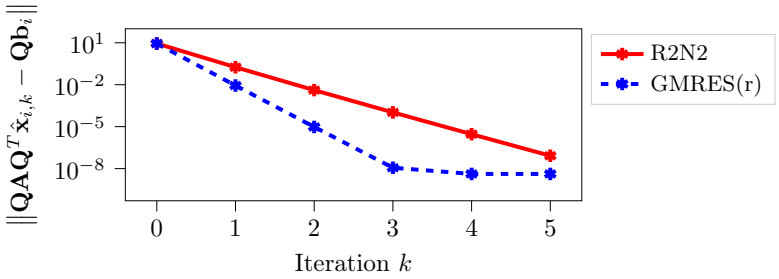
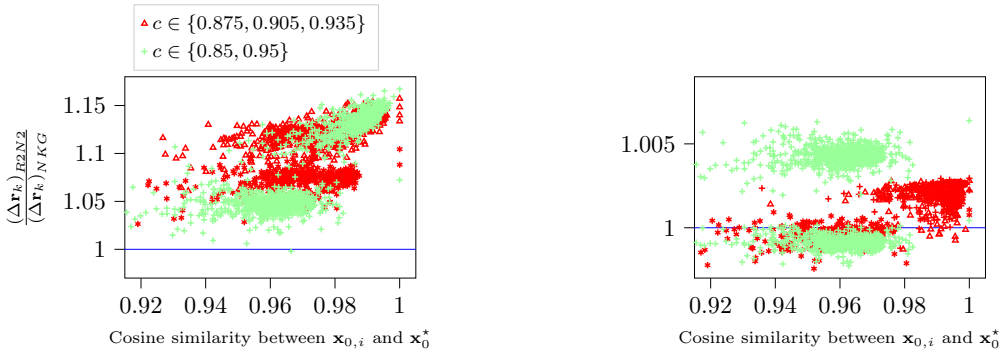


Fig. 6. Convergence of the R2N2 applied to the embedded problem (20). The R2N2 was trained on samples from the original problem (11) (same R2N2 as the one used in Figure 5). Both solvers again use $n = 4$ inner iterations. The lines connect the sample mean at each (outer) iteration k.

ual:

$$\Delta\mathbf{r}_k = \|\boldsymbol{f}(\mathbf{x}_0)\| - \|\boldsymbol{f}(\mathbf{x}_k)\|\,. \tag{22}$$

Figure 7 showcases the performance of the R2N2 on individual samples after training for 2 nonlinear iterations. The R2N2 is able to achieve more progress within these first two iterations than NK-GMRES even though it does not explicitly perform the minimization in the Krylov subspace to compute the step. The advantage in performance is maintained over a range of values for coefficient c in $\mathbf{A}_c$ both within the training range (red markers) and outside of the training range (light green). We therefore deduce that the R2N2 has learned to construct a subspace that contains more of the true solution for a specific problem class than the subspace formed by NK-GMRES. This is possible because the subspace $\mathcal{K}_n(\mathbf{J}(\mathbf{x}_k), -\boldsymbol{f}(\mathbf{x}_k))$, Equation (16), used in the k-th iteration depends not only on $(\boldsymbol{f}, \mathbf{x}_k)$ but also on the trainable parameters $\boldsymbol{\theta}_j$ in the $\boldsymbol{N}_j$ modules of the R2N2, cf. Equation (7b). Finally, we observed no notable difference between problem samples with $m = 10$ and $m = 20$. We applied the R2N2 from Figure 7 for a total of 7 nonlinear iterations, see Figure 8. The advantage of the R2N2 over NK-GMRES vanishes for 3 and more nonlinear iterations, but the R2N2 does still converge to an approximate solution when applied iteratively. Note, however, that this promising finding does not guarantee that the R2N2 can be trained to converge to a solution of arbitrary nonlinear equation systems. Furthermore, Figure 8 shows extrapolation to problems with $m = 100$. Similar to the case in the previous subsection, the R2N2 is agnostic of the problem dimension, which in this case allows generalization across various discretizations of the problem. Thus, overall the R2N2 is applicable to solving instances of Equation (21) without explicit access to either of its generative properties, i.e., c and m, something an unstructured neural network trained on problem data was found to be incapable of, cf. (**SM2**). Finally, we studied extrapolation for different initial guesses. The test set denoted by $\{\mathbf{x}_0'\}$ is generated with 10-fold variance compared to the training set, whereas the test set denoted $\{\mathbf{x}_0''\}$



(a) After $k = 1$ nonlinear iterations.

(b) After $k = 2$ nonlinear iterations.

Fig. 7. Relative performance of R2N2 vs NK-GMRES using $n = 3$ inner iterations and one (a) or two (b) outer iterations on Problem (21) with various coefficients c and various initial guesses $\mathbf{x}_{0,i}$. Note the differences in scales between the two subfigures. Instances where c is extrapolated outside of the training range are plotted in light green. The residuals are computed using Equation (22) with subscripts 'R2N2' and 'NKG' for the R2N2 and the NK-GMRES solver, respectively. $\mathbf{x}_0^\star$ refers to the sample for which the relative performance of R2N2 is best in the respective experiment.

consists of initial guesses centered around a different vector, here $5 \cdot \mathbb{1}^m$s. For the latter, the R2N2 shows some convergence, also continuing over further iterations that were not plotted herein. After 15 iterations, all test samples except for a couple (that are still converging) have reached a tolerance of $1e{-}8$. NK-GMRES, on the other hand, produces more outliers: after 15 iterations, only 63.2 % of samples have converged. This number is increased to 76.7 % after 20 iterations, 98 % after 50 iterations and 99.4 % after 75 iterations. That is, the mean curves are dominated by the instances that are slow to converge. This distinct difference in performance between the R2N2 and NK-GMES merits further study.

4.3 Solving initial-value problems

Finally, we study the solution of initial-value problems with the known RHS $\boldsymbol{f}$, i.e., Problem (3). We covered learning integrators thoroughly in our previous work (Guo et al., 2022) where we employed a Taylor series-based regularization to promote certain orders of convergence as property of the RK-NN. We now demonstrate that RK-NN integrators that outperform classical RK integrators can also be learned without special regularizers and, as a further extension of our previous work, that the RK-NN integrators work over multiple timesteps. All RK-NNs in this section are trained from the R2N2 superstructure and we thus name them R2N2 in the remainder of the section.

We reiterate the van der Pol oscillator from our previous work, i.e.,

$$\dot{x}_{(1)}(t) = x_{(2)}(t), \tag{23a}$$

$$\dot{x}_{(2)}(t) = a\left(1 - x_{(1)}^2(t)\right) x_{(2)}(t) - x_{(1)}(t), \tag{23b}$$

with $\mathbf{x} = \left(x_{(1)}, x_{(2)}\right)$. For data generation, we sample coefficients $a \sim \mathcal{U}(1.35, 1.65)$, initial values $x_{(1)}(t = t_0) \sim \mathcal{U}(-4, -3)$ and $x_{(2)}(t = t_0) \sim \mathcal{U}(0, 2)$ and timesteps $h \in [0.01, 0.1]$ equidistantly Note that $\mathbf{x}(t = t_0)$ and h are contained in the problem parameters $\mathbf{p}$. For Problem (3), we use them directly as the inputs $\mathbf{x}_k$ and h of the R2N2, cf. Figure 2. We generate target data (that also servers as ground truth) for the timesteps h using SciPy's *odeint* (Virtanen et al., 2020) with error tolerance set to 10^{-8}. Training loss is calculated by the following specification of Equation (17a):

$$MSE_x = \sum_{k=1}^{T} \sum_{i=1}^{N} \frac{\left(\hat{\mathbf{x}}_{i,k} - \mathbf{x}_{i,k}^t\right)^2}{h_i^p}.$$

The losses are summed over T integration steps of a trajectory with the timestep h, i.e., at times $t_0 + h, t_0 + 2h, \ldots, t_0 + Th$. Further, the denominator of the loss terms allows weighting samples based on the timestep of a specific sample, h_i, and, in particular, weighting according to an expected convergence order p. We set $p = n$ for the results in Section 4.3, where n is the number of layers of the RK-NN. The function evaluations in the R2N2 directly evaluate the RHS of Equation (23). One pass

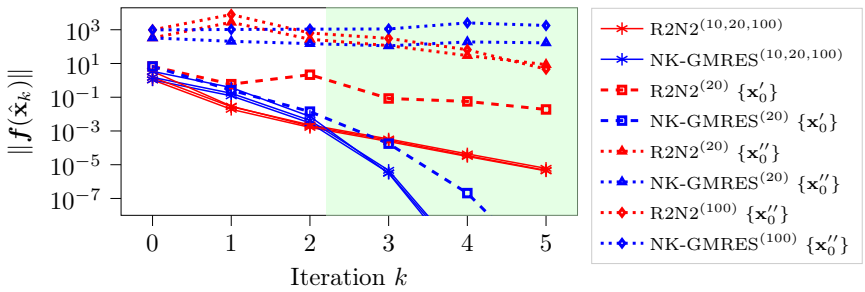


Fig. 8. Convergence of the R2N2 (red) vs NK-GMRES (blue) for 8 nonlinear iterations of solving (21). Both solvers use 3 inner iterations. The R2N2 used herein coincides with the one shown in Figure 7b, i.e., it was trained only on the first 2 nonlinear iterations. The shaded area $k > 2$ denotes performing more iterations than were used for training. The superscripts indicate the set of problem dimensions m used. Performance for samples of $\{\mathbf{x}_0\}$ from the training range are shown by solid lines. Dashed lines use initial samples $\{\mathbf{x}_0'\}$ with 10-fold noise around $\mathbb{1}^{(m)}$, whereas dotted lines use initial samples $\{\mathbf{x}_0''\}$ centered at $5 \cdot \mathbb{1}^{(m)}$.

through the R2N2 therefore resembles one step of a RK method.

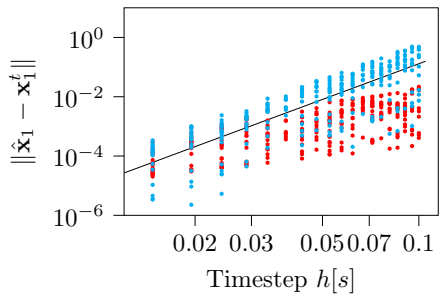
To assess the performance of the learned integrators, we compare the R2N2 instantiated with n layers to classical RK methods of n stages, denoted by RK-n. The training data contains samples with varying coefficients a_i, timesteps h_i and initial values $\mathbf{x}_i(t = 0)$. The error for either integrator is evaluated against a ground truth approximated by *odeint* and denoted $\mathbf{x}_k^t$. Results using $n = 3$ on the van der Pol oscillator, Equation (23), are shown in Figure 9. In Figure 9a–9d, we show results for training on $T = 1$ step of integration and evaluating on $T = 1$, $T = 2$, $T = 3$, and $T = 5$ steps, respectively, i.e., for $k > 1$, the iterates $\mathbf{x}_k$ that go into the R2N2 as inputs lie outside of its training range. We thus test for, and verify, some limited generalization. Over the first three timesteps, the R2N2 integrates the van der Pol oscillator equations more accurately than RK-3. We want to stress here that the classical RK method uses fixed coefficients for the computation of stage values and for the computation of the step itself. Therefore, the R2N2 can improve over the classical RK with regards to both of these computational steps. Evidently, we are able to learn coefficients $\boldsymbol{\theta}_j$ in the stages $\boldsymbol{N}_j$ and $\boldsymbol{\theta}_n$ for the output $\boldsymbol{N}_n$ that lead to a better approximation of the integral for problems from the training distribution, including limited generalization, than the RK method does.

Overall, we find that the accuracy of the R2N2 gradually decreases over the number of timesteps such that the R2N2 is not more accurate than RK-3 after the 5-th iteration anymore. The trend observed in Figure 9a–9d, although less pronounced, is consistent with the behavior of the R2N2 over multiple iterations of equation solving.

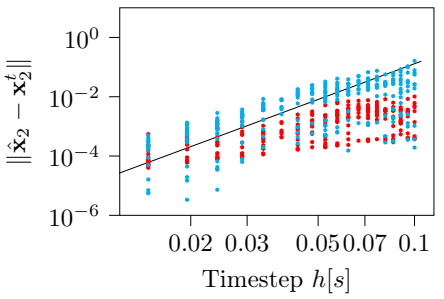
5 Conclusion

This work proposes an alternative, augmented perspective for the use of the RK-NN, a recurrent neural network templated on RK integrators, as a recursively recurrent superstructure for a wider class of iterative numerical algorithms. The R2N2 superstructure embeds function evaluations inside its layers and feeds the output to all successive layers via forward skip connections and linear combinations. The embedding of function calls into the architecture disentangles the algorithm to be learned from the function it acts on. We have shown that the R2N2 superstructure provides an inductive bias towards iterative algorithms based on recurrent function evaluation, e.g., the well-established Krylov subspace solvers and RK methods. Our numerical experiments demonstrate that, thus, the R2N2 superstructure can mimic steps of Krylov, Newton-Krylov and RK algorithms, respectively.

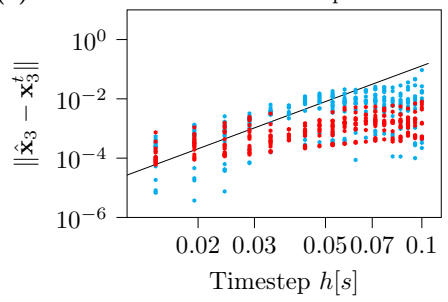
In particular, when learning a single step of linear equation solvers (Section 4.1), the performance of the R2N2 is bounded by that of the benchmark GMRES. This is consequential considering that the subspaces generated by the two approaches are equal but GMRES further *minimizes* the residual in this subspace, whereas the R2N2 learns a linear combination that can coincide with the minimizer, cf. Equation (14), for at most a single problem instance. In contrast, a nonlinear equa-



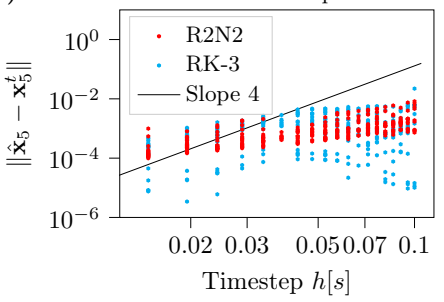
(a) The R2N2 error after one step is shown.



(b) The R2N2 error after two steps is shown.



(c) The R2N2 error after three steps is shown.



(d) The R2N2 error after five steps is shown.

Fig. 9. Performance of R2N2 vs RK-3 integrating the van der Pol oscillator equations, Equation (23), with varying coefficients a, stepsizes h_i and initial conditions $\mathbf{x}_{0,i}$. The figures show the same R2N2, trained on one timestep, and evaluated after one (a), two (b), three (c) and five timesteps (d), respectively. The slope of 4 is added to indicate the nominal local truncation error of RK-3.

tion solver (Section 4.2) can improve upon iterations performed by NK-GMRES, because in this case the R2N2 can learn to construct subspaces in which the residual is reduced more strongly than by NK-GMRES. Applying the R2N2 for multiple outer iterations resembles a restarted iterative solver for linear problems or a Newton-Krylov method for nonlinear problems, respectively. We have empirically demonstrated the ability of the R2N2 to converge to solutions of linear and nonlinear equation solving problems, however, we cannot guarantee the convergence of trained solvers for arbitrary problems. The R2N2 is also capable of extrapolation to similar problems as the ones seen during training. This includes not only extensions to the range of problem parameters but also embedding in higher dimensions or finer discretization. Finally, we have revisited our previous work about learning RK integrators (Guo et al., 2022) by demonstrating successful integration over multiple timesteps (Section 4.3). Our results suggest that the advantage of the R2N2 or RK-NN, respectively, over a classical RK method cannot be sustained over longer time horizons of integration.

In summary, iterative algorithms trained within the R2N2 superstructure can, when possible, find a subspace in which the residual can be reduced more than by their classical counterparts, given the same number of function evaluations.

6 Future research directions

6.1 Application to other computational problems

For further application, the compatibility of the R2N2 superstructure with other problem classes that comply with the general form of Problem (1) and that are solved with iterative algorithms, e.g., eigenvalue computation, PCA decomposition (Gemp et al., 2021), or computation of Neumann series (Liao et al., 2018), could be assessed. Moreover, besides learning an algorithm as a neural network architecture that maps from a set of problems to their solutions, the superstructure proposed in this work can also be deployed in the *inverse* setting, i.e., to identify a problem or function under the action of the known algorithm given input/output

data. This was the original motivation in Rico-Martinez et al. (1995), leading to nonlinear system identification, that can be analyzed using inverse backward error analysis (Zhu et al., 2020, 2023).

6.2 Extension of the superstructure

Other, more intricate algorithms can be represented by the proposed superstructure if its architecture is extended with additional trainable modules. For instance, *general linear methods* for integration (see Butcher (2016)) can be captured by the superstructure if not just the final output is subject to recurrence, but the layer outputs $\mathbf{v}_j$ are too (cf. with Figure 2). Moreover, a preconditioner (see Section 8 of Saad and Van Der Vorst (2000)) can be inserted inside the layers of the superstructure: for instance, a cheap, approximate model of $\boldsymbol{f}$ can be used to to effectively build such a preconditioner (Qiao et al., 2006). Finally, non-differentiable operations that are part of most algorithms, e.g., the checking of an error tolerance, can be included via smoothed relaxations (Ying et al., 2018; Tang et al., 2020).

6.3 Neural architecture search

Larger architectures will eventually call for superstructure optimization to yield parsimonious algorithms, i.e., determining the optimal (sub-)structure for a given set of problem instances together with the optimized parameters. This challenge can be formulated as a mixed-integer nonlinear program (MINLP). Presently, these problems are addressed by heuristic methods, referred to as neural architecture search (NAS, Elsken et al. (2019b); Hospedales et al. (2021)), that are relatively efficient in finding good architectures. For the superstructure, three types of NAS methods appear suited: i) structured search spaces to exploit modularity, e.g., (Liu et al., 2017), (Zoph et al., 2018), (Negrinho et al., 2019) and (Schrodi et al., 2022), ii) adaptively growing search spaces for refining the architecture, e.g., (Cortes et al., 2017), (Elsken et al., 2019a) and (Schiessler et al., 2021), and iii) differentiable architecture search, e.g., (Liu et al., 2019) (Li et al., 2021a). Sparsity promoting training techniques like pruning typically address dense, fully-connected layers and, therefore, are expected to provide little use in the current architecture. Similar reasoning applies to generic regularizers like $\mathcal{L}_2$ regularization. On the other hand, tailored regularizers such as physics-informed losses or the regularizer we proposed in Guo et al. (2022) can prove useful to promote specific algorithmic properties.

More traditional MINLP solvers like those used for superstructure optimization of process systems (Grossmann, 2002; Burre et al., 2022) exhibit certain advantages over NAS methods. They explicitly deal with integer variables allowing sophisticated use of discrete choices e.g., for mutually exclusive architecture choices. Moreover, MINLPs can be solved deterministically to guarantee finding a global solution, e.g., by a branch-and-bound algorithm (Belotti et al., 2013). Global solution of MINLPs involving the superstructure is challenging, since it requires neural network training subproblems to be solved globally. With future advances in computational hardware and algorithms this may become a viable approach. However, substantial effort is needed to utilize such MINLP solvers for the training tasks considered herein.

6.4 Implicit layers

An orthogonal approach for increasing the scope of the superstructure and capitalizing on its modularity is to endow only a subset of its modules or layers with trainable variables. The remaining modules that are not subject to meta-optimization can be implemented by so called *implicit layers* that implement their functionality, see, e.g., (Rajeswaran et al., 2019) and (Lorraine et al., 2020). In future work, we plan to emulate the residual minimization of Krylov solvers by substituting the output layer $\boldsymbol{N}_n$ of the superstructure with differentiable convex optimization layers (Amos and Kolter, 2017; Agrawal et al., 2019). Then, only the optimal subspace generation, i.e., the parameters of the $\boldsymbol{N}_j$ modules, is left to learn, or the subspace generation is optimized with respect to consecutive minimization being performed in that subspace, respectively.

6.5 Dynamical systems perspective

A joint perspective on neural networks and dynamical systems has emerged recently, e.g., (E, 2017), (Haber and Ruthotto, 2017), and (Chang et al., 2017). Similarly, a connection between dynamical systems and continuous-time limits of iterative algorithms has been discussed in literature (Stuart and Humphries, 1998; Chu, 2008; Dietrich et al., 2020), especially for convex optimization (Su et al., 2014; Krichene et al., 2015; Wibisono et al., 2016). Researchers have applied numerical integration schemes to these continuous forms to recover discretized algorithms (Scieur et al., 2017;

Betancourt et al., 2018; Zhang et al., 2018). Conversely, the underlying continuous-time dynamics of discrete algorithms encoded by the proposed R2N2 can be identified based on the iterates they produce, e.g., by their associated Koopman operators (Dietrich et al., 2020). These Koopman operators can then be analyzed to compare various algorithms and, even, to identify conjugacies between them (Redman et al., 2022).

Declaration of Competing Interest

We have no conflict of interest.

Acknowledgements

DTD, AM and MD received funding from the Helmholtz Association of German Research Centres and performed this work as part of the Helmholtz School for Data Science in Life, Earth and Energy (HDS-LEE). YG and QL are supported by the National Research Foundation, Singapore, under the NRF fellowship (project No. NRF-NRFF13-2021-0005). FD received funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — 468830823. The work of IGK is partially supported by the US Department of Energy and the US Air Force Office of Scientific Research.

Bibliography

Agrawal, A., Amos, B., Barratt, S. T., Boyd, S. P., Diamond, S., and Kolter, J. Z. (2019). Differentiable convex optimization layers. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9558–9570.

Amos, B. and Kolter, J. Z. (2017). Optnet: Differentiable optimization as a layer in neural networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 136–145. PMLR.

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Lee, D. D., Sugiyama, M., von Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 3981–3989.

Balcan, M. (2020). Data-driven algorithm design. In Roughgarden, T., editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 626–645. Cambridge University Press.

Balcan, M., Dick, T., Sandholm, T., and Vitercik, E. (2018). Learning to branch. In Dy, J. G. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 353–362. PMLR.

Baxter, J. (2000). A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198.

Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., and Mahajan, A. (2013). Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131.

Betancourt, M., Jordan, M. I., and Wilson, A. C. (2018). On symplectic optimization. *arXiv preprint arXiv:1802.03653*.

Burre, J., Bongartz, D., and Mitsos, A. (2022). Comparison of MINLP formulations for global superstructure optimization. *Optimization and Engineering*, pages 1–30.

Butcher, J. C. (2016). *Numerical methods for ordinary differential equations*. Wiley, Chichester, West Sussex, third edition edition.

Chang, B., Meng, L., Haber, E., Tung, F., and Begert, D. (2017). Multi-level residual networks from dynamical systems view. *arXiv preprint arXiv:1710.10348*.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N.,

and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6572–6583.

Chevalier, S., Stiasny, J., and Chatzivasileiadis, S. (2021). Contracting neural-newton solver. *arXiv preprint arXiv:2106.02543*.

Chu, M. T. (2008). Linear algebra algorithms as dynamical systems. *Acta Numerica*, 17:1–86.

Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M., and Yang, S. (2017). Adanet: Adaptive structural learning of artificial neural networks. In *International Conference on Machine Learning*, pages 874–883. PMLR.

Denevi, G., Ciliberto, C., Stamos, D., and Pontil, M. (2018). Learning to learn around A common mean. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10190–10200.

Dietrich, F., Thiem, T. N., and Kevrekidis, I. G. (2020). On the Koopman operator of algorithms. *SIAM Journal on Applied Dynamical Systems*, 19(2):860–885.

Dufera, T. T. (2021). Deep neural network for system of ordinary differential equations: Vectorized algorithm and simulation. *Machine Learning with Applications*, 5:100058.

E, W. (2017). A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11.

Elsken, T., Metzen, J. H., and Hutter, F. (2019a). Efficient multi-objective neural architecture search via Lamarckian evolution. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Elsken, T., Metzen, J. H., and Hutter, F. (2019b). Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F. J., Schrittwieser, J., Swirszcz, G., et al. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53.

Gemp, I. M., McWilliams, B., Vernade, C., and Graepel, T. (2021). Eigengame: PCA as a Nash equilibrium. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

González-García, R., Rico-Martínez, R., and Kevrekidis, I. G. (1998). Identification of distributed parameter systems: A neural net based approach. *Computers & Chemical Engineering*, 22:S965–S968.

Goyal, P. and Benner, P. (2021). Learning dynamics from noisy measurements using deep learning with a Runge–Kutta constraint. *arXiv preprint arXiv:2109.11446*.

Grossmann, I. E. (2002). Review of nonlinear mixed-integer and disjunctive programming techniques. *Optimization and Engineering*, 3(3):227–252.

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

Gupta, R. and Roughgarden, T. (2020). Data-driven algorithm design. *Communications of the ACM*, 63(6):87–94.

Haber, E. and Ruthotto, L. (2017). Stable architectures for deep neural networks. *Inverse problems*, 34(1):014004.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

Hoos, H. H. (2011). Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer.

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2021). Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5149–5169.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2010). Automated configuration of mixed-integer programming solvers. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 186–202. Springer.

Jin, P., Zhang, Z., Zhu, A., Tang, Y., and Karniadakis, G. E. (2020). Sympnets: Intrinsic structure-preserving symplectic networks for identifying hamiltonian systems. *Neural Networks*, 132:166–179.

Kelley, C. T. (1995). *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics.

Kelley, C. T. (2003). *Solving nonlinear equations with Newton's method*. SIAM.

Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G. L., and Dilkina, B. (2016). Learning to branch in mixed integer programming. In Schuurmans, D. and Wellman, M. P., editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 724–731. AAAI Press.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2016). SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence*, 232:20–42.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Knoll, D. A. and Keyes, D. E. (2004). Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397.

Krichene, W., Bayen, A. M., and Bartlett, P. L. (2015). Accelerated mirror descent in continuous and discrete time. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2845–2853.

Larsson, G., Maire, M., and Shakhnarovich, G. (2017). Fractalnet: Ultra-deep neural networks without residuals. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Li, L., Khodak, M., Balcan, N., and Talwalkar, A. (2021a). Geometry-aware gradient algorithms for neural architecture search. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Li, Z., Kovachki, N. B., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A. M., and Anandkumar, A. (2021b). Fourier neural operator for parametric partial differential equations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Liao, R., Xiong, Y., Fetaya, E., Zhang, L., Yoon, K., Pitkow, X., Urtasun, R., and Zemel, R. S. (2018). Reviving and improving recurrent back-propagation. In Dy, J. G. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3088–3097. PMLR.

Liu, D. C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Lorraine, J., Vicol, P., and Duvenaud, D. (2020). Optimizing millions of hyperparameters by implicit differentiation. In Chiappa, S. and Calandra, R., editors, *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, volume 108 of *Proceedings of Machine Learning Research*, pages 1540–1552. PMLR.

Lovelett, R. J., Avalos, J. L., and Kevrekidis, I. G. (2020). Partial observations and conservation laws: Gray-box modeling in biotechnology and optogenetics. *Industrial & Engineering Chemistry Research*, 59(6):2611–2620.

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

Lu, Y., Zhong, A., Li, Q., and Dong, B. (2018). Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In Dy, J. G. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3282–3291. PMLR.

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J.-B., Ahern, A., et al. (2023). Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263.

Mencarelli, L., Chen, Q., Pagot, A., and Grossmann, I. E. (2020). A review on superstructure optimization approaches in process system engineering. *Computers & Chemical Engineering*, 136:106808.

Metz, L., Maheswaranathan, N., Freeman, C. D., Poole, B., and Sohl-Dickstein, J. (2020). Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*.

Mezzadri, F. (2006). How to generate random matrices from the classical compact groups. *arXiv preprint math-ph/0609050*.

Mishra, S. (2018). A machine learning framework for data driven acceleration of computations of differential equations. *arXiv preprint arXiv:1807.09519*.

Mitsos, A., Najman, J., and Kevrekidis, I. G. (2018). Optimal deterministic algorithm generation. *Journal of Global Optimization*, 71(4):891–913.

Nascimento, R. G., Fricke, K., and Viana, F. A. (2020). A tutorial on solving ordinary differential equations using python and hybrid physics-informed neural network. *Engineering Applications of Artificial Intelligence*, 96:103996.

Negrinho, R., Gormley, M. R., Gordon, G. J., Patil, D., Le, N., and Ferreira, D. (2019). Towards modular and programmable architecture search. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13715–13725.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

Qiao, L., Erban, R., Kelley, C. T., and Kevrekidis, I. G. (2006). Spatially distributed stochastic systems: Equation-free and equation-assisted preconditioned computations. *The Journal of Chemical Physics*, 125(20):204108.

Rajeswaran, A., Finn, C., Kakade, S. M., and Levine, S. (2019). Meta-learning with implicit gradients. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 113–124.

Redman, W. T., Fonoberova, M., Mohr, R., Kevrekidis, I. G., and Mezić, I. (2022). Algorithmic (semi-) conjugacy via Koopman operator theory. *arXiv preprint arXiv:2209.06374*.

Rice, J. R. (1976). The algorithm selection problem. In *Advances in Computers*, volume 15, pages 65–118. Elsevier.

Rico-Martinez, R., Anderson, J. S., and Kevrekidis, I. G. (1994). Continuous-time nonlinear signal processing: A neural network based approach for gray box identification. In *Proceedings of IEEE Workshop on Neural Networks for Signal Processing*, pages 596–605.

Rico-Martinez, R., Kevrekidis, I. G., and Krischer, K. (1995). Nonlinear system identification using neural networks: Dynamics and instabilities. In *Neural Networks for Chemical Engineers*, pages 409–442. Elsevier Amsterdam, The Netherlands.

Rico-Martinez, R., Krischer, K., Kevrekidis, I. G., Kube, M., and Hudson, J. (1992). Discrete- vs. continuous-time nonlinear signal processing of Cu electrodissolution data. *Chemical Engineering Communications*, 118(1):25–48.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. SIAM.

Saad, Y. and Schultz, M. H. (1986). GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

Saad, Y. and Van Der Vorst, H. A. (2000). Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33.

Schiessler, E. J., Aydin, R. C., Linka, K., and Cyron, C. J. (2021). Neural network surgery: Combining training with topology optimization. *Neural Networks*, 144:384–393.

Schrodi, S., Stoll, D., Ru, B., Sukthanker, R., Brox, T., and Hutter, F. (2022). Towards discovering neural architectures from scratch. *arXiv preprint arXiv:2211.01842*.

Schwarzschild, A., Borgnia, E., Gupta, A., Huang, F., Vishkin, U., Goldblum, M., and Goldstein, T. (2021). Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 6695–6706.

Scieur, D., Roulet, V., Bach, F. R., and d'Aspremont, A. (2017). Integration methods and optimization algorithms. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1109–1118.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144.

Stuart, A. and Humphries, A. R. (1998). *Dynamical systems and numerical analysis*, volume 2. Cambridge University Press.

Su, W., Boyd, S. P., and Candès, E. J. (2014). A differential equation for modeling Nesterov's accelerated gradient method: Theory and insights. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2510–2518.

Tang, H., Huang, Z., Gu, J., Lu, B., and Su, H. (2020). Towards scale-invariant graph-related problem solving by iterative homogeneous GNNs. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Tawarmalani, M. and Sahinidis, N. V. (2005). A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103(2):225–249.

Tsitouras, C. (2002). Neural networks with multidimensional transfer functions. *IEEE Transactions on Neural Networks*, 13(1):222–228.

Venkataraman, S. and Amos, B. (2021). Neural fixed-point acceleration for convex optimization. *arXiv preprint arXiv:2107.10254*.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van

der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Wibisono, A., Wilson, A. C., and Jordan, M. I. (2016). A variational perspective on accelerated methods in optimization. *Proceedings of the National Academy of Sciences*, 113(47):E7351–E7358.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Yeomans, H. and Grossmann, I. E. (1999). A systematic modeling framework of superstructure optimization in process synthesis. *Computers & Chemical Engineering*, 23(6):709–731.

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W. L., and Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4805–4815.

Zhang, J., Mokhtari, A., Sra, S., and Jadbabaie, A. (2018). Direct Runge–Kutta discretization achieves acceleration. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3904–3913.

Zhao, J. and Mau, J. (2020). Discovery of governing equations with recursive deep neural networks. *arXiv preprint arXiv:2009.11500*.

Zhu, A., Bertalan, T., Zhu, B., Tang, Y., and Kevrekidis, I. G. (2023). Implementation and (inverse modified) error analysis for implicitly-templated ode-nets. *arXiv preprint arXiv:2303.17824*.

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

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710. Computer Vision Foundation / IEEE Computer Society.

A Recursively Recurrent Neural Network (R2N2) Architecture for Learning Iterative Algorithms – Supplementary Information

Danimir T. Doncevic[a,b] Alexander Mitsos[c,a,d] Yue Guo[e] Qianxiao Li[e] Felix Dietrich[f] Manuel Dahmen[a,*] Ioannis G. Kevrekidis[g,*]

[a] Institute of Energy and Climate Research, Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, Jülich 52425, Germany
[b] RWTH Aachen University Aachen 52062, Germany
[c] JARA-ENERGY, Jülich 52425, Germany
[d] RWTH Aachen University, Process Systems Engineering (AVT.SVT), Aachen 52074, Germany
[e] Department of Mathematics, National University of Singapore, Singapore
[f] Department of Informatics, Technical University of Munich, Boltzmannstr. 3, 85748 Garching b. Munich, Germany
[g] Departments of Applied Mathematics and Statistics & Chemical and Biomolecular Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA

SM1. Forward-differencing in the R2N2 superstructure

SM1.1. Black-box function evaluations

In order to promote discovery of time-proven algorithms, the superstructure can induce some prior structure. For instance, the RK-NN developed in our previous work (Guo et al., 2022) essentially sets

$$\mathbf{v}_j = \boldsymbol{f}(\mathbf{x}_k + h \sum_{l=0}^{j-1} a_{j,l} \mathbf{v}_l)$$

to promote RK methods with lower-triangular Butcher tableaus. In the R2N2 superstructure, this corresponds to Equation (3.1) with $\theta_j := (a_{j,0}, \ldots, a_{j,j-1})$ followed by a direct function evaluation, $\boldsymbol{f}(\mathbf{x}'_j)$.

We can also hard-wire some of the operations of the superstructure to add yet another implicit bias. For example, we can encode forward-differencing, i.e.,

$$\mathbf{v}_j = \frac{1}{\epsilon} \left(\boldsymbol{f}(\mathbf{x}_k + \epsilon \mathbf{x}'_j) - \boldsymbol{f}(\mathbf{x}_k) \right), \qquad (1)$$

inside the superstructure for estimation of directional derivatives. Equation (1) estimates the derivative of $\boldsymbol{f}$ at $\mathbf{x}_k$ in the direction of some $\mathbf{x}'_j$. That is, the j-th layer then always approximates a directional derivative at the current iterate $\mathbf{x}_k$, where only the direction is determined by the trainable weights θ_j in $\boldsymbol{N}_j$ according to Equation (3.1). The corresponding pass through a layer of the superstructure is sketched in Figure 1. While Equation (1) and Figure 1 appear to deviate from Equation (2.7b) and Figure 2, respectively, we next illustrate that forward-differencing is contained as a special case in the superstructure.

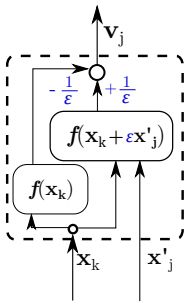


Fig. 1. The dashed cell indicates forward-differencing for estimation of directional derivatives with a small ϵ in the superstructure. The directional derivative at $\mathbf{x}_k$ requires reusing the first layer output $\boldsymbol{f}(\mathbf{x}_k)$ to explicitly construct finite differences between the outputs of the two function evaluations.

SM1.2. Forward-differencing can be expressed by regular function evaluations in the R2N2 superstructure

Figure 2 a) illustrates the hard-wiring of forward-differencing operations (Equation (1)) in the layers of the R2N2 superstructure, such that the layer output estimates a directional derivative at $\mathbf{x}_k$. The layer architecture shown in Figure 2 a) results from a specific parametrization of the general R2N2 superstructure, see Figure 2 b). If the optimal R2N2 parameters for the forward-differencing configuration are given by $\tilde{\theta}$, there exists a corresponding set of parameters θ for the R2N2 structure using

*M. Dahmen, Institute of Energy and Climate Research, Energy Systems Engineering (IEK-10), Forschungszentrum Jülich GmbH, Jülich 52425, Germany
E-mail: m.dahmen@fz-juelich.de
I. Kevrekidis, Departments of Applied Mathematics and Statistics & Chemical and Biomolecular Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA
E-mail: yannisk@jhu.edu

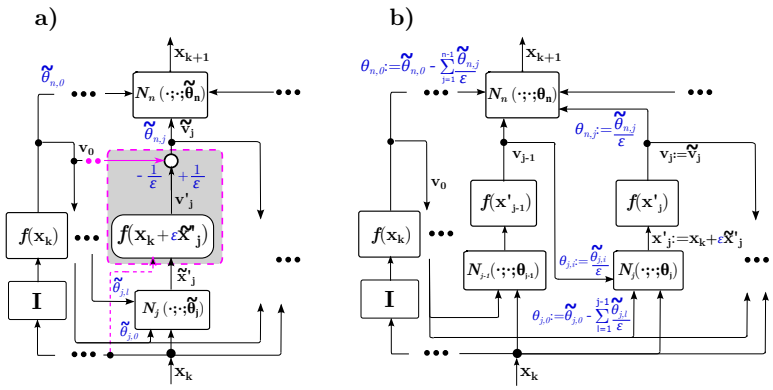


Fig. 2. Topological equivalence between one forward-differencing layer a) and a regular function evaluation layer b). The gray box with dashed pink surrounding encapsulates the encoding of a forward differencing. Black symbols are input and output quantities inside the superstructure. The quantities belonging to the forward-differencing encoding are marked by a tilde. Note that two additional inputs are needed for the forward-differencing layer. First, $\mathbf{x}_k$ is directly passed to the function evaluation such that $\mathbf{x}_k + \epsilon\tilde{\mathbf{x}}'$ acts as the function argument where the linear module $\boldsymbol{N}_j\left(\tilde{\boldsymbol{\theta}}_j\right)$ outputs an *intermediate* $\tilde{\mathbf{x}}'$. Second, $\boldsymbol{f}(\mathbf{x}_k) = \mathbf{v}_0$ is subtracted from that $\tilde{\mathbf{x}}'$ and a division by ϵ yields the final output $\tilde{\mathbf{v}}_j$ of the forward-differencing layer. Scalar weights are shown as blue symbols and are related to the trainable parameters. In b), on the other hand, $\boldsymbol{N}_j\left(\boldsymbol{\theta}_j\right)$ must directly return $\mathbf{x}'$ such that the layer output $\mathbf{v}_j$ equals $\tilde{\mathbf{v}}_j$ from a). Additionally, in b) the transformations from $\tilde{\boldsymbol{\theta}}$ to $\boldsymbol{\theta}$ are shown that result in equivalent output $\mathbf{x}_{k+1}$.

regular function calls, which follows directly from $\tilde{\boldsymbol{\theta}}$. For example, the weighting of the first function evaluation, $\mathbf{v}_0$ is transformed by

$$\theta_{j,0} := \tilde{\theta}_{j,0} - \sum_{l=1}^{j-1} \frac{\tilde{\theta}_{j,l}}{\epsilon}, \tag{2}$$

where ϵ corresponds to the value used for calculating finite differences in the finite-differencing module, e.g., $\epsilon = 10^{-8}$. Moreover, the forward-differencing module in Figure 2 a) can be thought of as having a fixed bypass with weight of 1 from the current iterate $\mathbf{x}_k$ since it computes a function of $\mathbf{x}_k + \epsilon\tilde{\mathbf{x}}'$, where $\tilde{\mathbf{x}}'$ is the output of $\boldsymbol{N}_j\left(\tilde{\boldsymbol{\theta}}_j\right)$. To have identical layer outputs $\tilde{\mathbf{v}}_j$ and $\mathbf{v}_j$, the module $\boldsymbol{N}_j\left(\boldsymbol{\theta}_j\right)$ in Figure 2 b) must directly return $\mathbf{x}' := \mathbf{x}_k + \epsilon\tilde{\mathbf{x}}'$. Note that the subtracted summation over preceding layer indices in Equation (2) is due to the construction in Figure 2 a) and Equation (1) that anchors all finite differences at $\mathbf{x}_k$. In theory, $\boldsymbol{\theta}_j$ could also express finite differences evaluated at arbitrary positions in the image of $\boldsymbol{N}_j$. However, the output layer weights $\boldsymbol{\theta}_n$ must change in a complementary fashion in this case, such that the overall inputs to $\boldsymbol{N}_n$ are still computed by a forward difference as demanded by Equation (1).

Hard-wiring an operation like forward-differencing may be useful to guide the training procedure. To achieve forward-differencing in the limit, certain trainable parameters must tend to zero. Choi et al. (2019) have analyzed a similar problem in the hyperparameter spaces of optimizers and noted a difficulty in learning parameters in different orders of magnitude. Hence, fixing a routine like forward-differencing *a priori* may be beneficial.

SM1.3. Implementation of forward-differencing modules

To fix forward-differencing (Equation (1)) in the R2N2 superstructure also requires a backward pass. We wrapped Equation (1) as a PyTorch *Autograd* function and manually implemented the backward passes. Backpropagating through the regular function evaluation, $\boldsymbol{f}$, is straightforward:

$$\frac{\partial \mathbf{v}_j}{\partial \mathbf{x}'_j}(\mathbf{x}'_j) = \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}(\mathbf{x}'_j). \tag{3}$$

Accordingly, for the backward pass through forward-differencing, we require the derivatives for

both of its terms:

$$\frac{\partial \mathbf{v}_j}{\partial \mathbf{x}_k}(\mathbf{x}'_k, \mathbf{x}'_j) = \frac{1}{\epsilon}\left(\frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}(\mathbf{x}_k + \epsilon \mathbf{x}'_j) - \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}(\mathbf{x}_k)\right), \quad \frac{\partial \mathbf{v}_j}{\partial \mathbf{x}'_j}(\mathbf{x}'_k, \mathbf{x}'_j) = \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}(\mathbf{x}'_j) \tag{4}$$

Note that the derivative with respect to $\mathbf{x}_k$ only impacts the weight update for $k > 1$, i.e., when computing at least two iterations with the R2N2 during training. $\mathbf{x}_0$ itself does not depend on trainable parameters. An alternative to the implementation described herein is to express the forward functions only in terms of PyTorch built-in operations or using other libraries that support automatic differentiation.

SM2. Detailed description of the generated test problems

SM2.1. Linear equation test problems

In this section, we describe the detailed generation of problem data for linear equation solving problems used in this study (paper and supplementary material). We consider linear problems of the form

$$\mathbf{A}\mathbf{x} - \mathbf{b} = 0,$$

where $\mathbf{A} \in \mathbb{R}^{m \times m}$, $\mathbf{x} \in \mathbb{R}^m$, and $\mathbf{b} \in \mathbb{R}^m$ and we choose $m = 5$ as the problem dimensionality. Matrices $\mathbf{A}$ are generated as $\mathbf{A} = \tilde{\mathbf{A}}^T\tilde{\mathbf{A}} + \mathbf{I}\boldsymbol{\lambda}$, where $\tilde{\mathbf{A}}$ is a matrix with random entries $a_{ij} \sim \mathcal{N}(0, 0.1)$ and $\boldsymbol{\lambda}$ can be used to manipulate the spectrum of $\mathbf{A}$. Here, we use $\boldsymbol{\lambda} = (1, 0.75, 0.5, 0.1, 0.1)^T$. These choices guarantee that $\mathbf{A}$ is symmetric positive definite with eigenvalues fairly suited for Krylov methods (Kelley, 1995), and, in extension, that Problem (3.3) has a unique solution. Different problem instances are completed by a right-hand side $\mathbf{b}_i$, which is sampled by adding uniform noise to a fixed randomly-chosen mean $\tilde{\mathbf{b}}$, i.e., $\mathbf{b}_i = \tilde{\mathbf{b}} + \mathbf{b}'_i$, where $\tilde{\mathbf{b}} \sim \mathcal{N}(0, 5 \cdot \mathbf{I}_m)$ and $\mathbf{b}' \sim \mathcal{U}(-\mathbb{1}^m, \mathbb{1}^m)$. The initial input to the R2N2 (and to GMRES) is fixed as $\mathbf{x}_0 = \mathbf{0}$. The matrices $\mathbf{A}_1$–$\mathbf{A}_3$ that are used for the training set are generated according to the procedure described above.

Further, we have have used the following matrices for additional experiments in Section SM3.1. $\mathbf{A}_4$–$\mathbf{A}_7$ are defined, using $\Delta\boldsymbol{\lambda} = (0.5, 0.4, 0.3, 0.2, 0.1)^T$, as:

$$\mathbf{A}_4 = \mathbf{A}_1 - diag(\Delta\boldsymbol{\lambda}),$$

$$\mathbf{A}_5 = \mathbf{A}_1 + 0.5 \cdot diag(\Delta\boldsymbol{\lambda}),$$

$$\mathbf{A}_6 = \mathbf{A}_1 + 1.3 \cdot diag(\Delta\boldsymbol{\lambda}),$$

$$\mathbf{A}_7 = \mathbf{A}_1 + 2.5 \cdot diag(\Delta\boldsymbol{\lambda}).$$

$\mathbf{A}_8$–$\mathbf{A}_{11}$ are randomly generated symmetric matrices. Each of these matrices has a random scalar, $\sigma_l \sim \mathcal{U}(0, 5)$, for the uniform range and then entries $\tilde{a}_{ij} \sim \mathcal{U}(0, \sigma_l)$. Finally, $\mathbf{A}$ is obtained as a symmetric matrix by computing $\tilde{\mathbf{A}}\tilde{\mathbf{A}}^T$.

Finally, $\mathbf{A}_{12}$–$\mathbf{A}_{19}$ are generated with the same procedure as $\mathbf{A}_1$ – $\mathbf{A}_3$. However, the random entries now use $a_{ij} \sim \mathcal{N}(0, \sigma_l)$ where 2 matrices are generated for each $\sigma_l \in \{0.3, 0.5, 0.7, 1\}$, i.e., $\mathbf{A}_{12}$ and $\mathbf{A}_{13}$ for $\sigma_l = 0.3$, $\mathbf{A}_{14}$ and $\mathbf{A}_{15}$ for $\sigma_l = 0.5$, $\mathbf{A}_{16}$ and $\mathbf{A}_{17}$ for $\sigma_l = 0.7$, and $\mathbf{A}_{18}$ and $\mathbf{A}_{19}$ for $\sigma_l = 1.0$.

$$\tilde{\mathbf{b}} = (2.483570, -0.691321, 3.238442, 7.615149, -1.170766)^T .$$

$$\mathbf{A}_1 = \begin{pmatrix} 1.392232 & 0.152829 & 0.088680 & 0.185377 & 0.156244 \\ 0.152829 & 1.070883 & 0.020994 & 0.068940 & 0.141251 \\ 0.088680 & 0.020994 & 0.910692 & -0.222769 & 0.060267 \\ 0.185377 & 0.068940 & -0.222769 & 0.833275 & 0.058072 \\ 0.156244 & 0.141251 & 0.060267 & 0.058072 & 0.735495 \end{pmatrix},$$

$$\mathbf{A}_2 = \begin{pmatrix} 1.122760 & -0.040031 & 0.113992 & 0.068578 & 0.089329 \\ -0.040031 & 0.920757 & 0.085742 & 0.089300 & 0.158474 \\ 0.113992 & 0.085742 & 0.896851 & 0.150485 & 0.044783 \\ 0.068578 & 0.089300 & 0.150485 & 0.729516 & 0.070168 \\ 0.089329 & 0.158474 & 0.044783 & 0.070168 & 1.163038 \end{pmatrix},$$

$$\mathbf{A}_3 = \begin{pmatrix} 1.037577 & 0.120230 & -0.149775 & 0.099841 & 0.169390 \\ 0.120230 & 1.095856 & 0.180211 & 0.120029 & 0.133797 \\ -0.149775 & 0.180211 & 0.781548 & 0.241405 & 0.320369 \\ 0.099841 & 0.120029 & 0.241405 & 0.877185 & 0.040910 \\ 0.169390 & 0.133797 & 0.320369 & 0.040910 & 0.602205 \end{pmatrix},$$

$$\mathbf{A}_8 = \begin{pmatrix} 9.801337 & -4.563474 & 2.196806 & -5.154676 & 5.063176 \\ -4.563474 & 48.751049 & -26.335994 & -3.910831 & 17.380485 \\ 2.196806 & -26.335994 & 31.887071 & 1.215492 & -12.532923 \\ -5.154676 & -3.910831 & 1.215492 & 4.0743960 & -5.876128 \\ 5.063176 & 17.380485 & -12.532923 & -5.876128 & 25.900849 \end{pmatrix},$$

$$\mathbf{A}_9 = \begin{pmatrix} 19.582102 & -1.721533 & 5.067191 & 20.194875 & 1.561468 \\ -1.721533 & 38.090555 & -11.445662 & -22.832142 & -0.152421 \\ 5.067191 & -11.445662 & 17.191893 & 14.784228 & -3.889048 \\ 20.194875 & -22.832142 & 14.784228 & 49.221081 & 22.059518 \\ 1.561468 & -0.152421 & -3.889048 & 22.059518 & 31.461613 \end{pmatrix},$$

$$\mathbf{A}_{10} = \begin{pmatrix} 1.543741 & -1.708336 & -0.855255 & 1.180115 & -0.606022 \\ -1.708336 & 7.993454 & 1.813288 & -0.855154 & -0.375811 \\ -0.855255 & 1.813288 & 2.131294 & -2.223852 & -0.808170 \\ 1.180115 & -0.855154 & -2.223852 & 3.296235 & 1.148258 \\ -0.606022 & -0.375811 & -0.8081702 & 1.148258 & 2.018821 \end{pmatrix},$$

$$\mathbf{A}_{11} = \begin{pmatrix} 0.554750 & 0.192700 & -0.030087 & -0.173792 & 0.078237 \\ 0.192700 & 0.134709 & 0.005420 & 0.156018 & -0.081507 \\ -0.030087 & 0.005420 & 0.491319 & -0.087115 & -0.068497 \\ -0.173792 & 0.156018 & -0.087115 & 0.923782 & -0.356224 \\ 0.078237 & -0.081507 & -0.068497 & -0.356224 & 0.197102 \end{pmatrix}.$$

$$\mathbf{A}_{12} = \begin{pmatrix} 1.803328 & 0.200759 & -0.355809 & -0.098682 & -0.037251 \\ 0.200759 & 1.243347 & 0.088843 & 0.263899 & 0.195536 \\ -0.355809 & 0.088843 & 1.495596 & 0.093483 & 0.383077 \\ -0.098682 & 0.263899 & 0.093483 & 1.295673 & 0.091526 \\ -0.037251 & 0.195536 & 0.383077 & 0.091526 & 1.171966 \end{pmatrix},$$

$$\mathbf{A}_{13} = \begin{pmatrix} 1.373797 & 0.029822 & 0.291240 & -0.06804 & -0.122712 \\ 0.029822 & 1.352286 & 0.213403 & 0.259224 & 0.113595 \\ 0.291240 & 0.213403 & 1.145153 & 0.260138 & -0.256945 \\ -0.068040 & 0.259224 & 0.260138 & 1.044292 & 0.023357 \\ -0.122712 & 0.113595 & -0.256945 & 0.023357 & 1.493027 \end{pmatrix},$$

$$\mathbf{A}_{14} = \begin{pmatrix} 1.875641 & 0.369074 & -0.254450 & 0.011282 & 0.086120 \\ 0.369074 & 1.438546 & 0.165303 & 0.330450 & 0.326974 \\ -0.254450 & 0.165303 & 1.578616 & 0.135095 & 0.435910 \\ 0.011282 & 0.330450 & 0.135095 & 1.407443 & 0.175663 \\ 0.086120 & 0.326974 & 0.435910 & 0.175663 & 1.302648 \end{pmatrix},$$

$$\mathbf{A}_{15} = \begin{pmatrix} 1.496302 & 0.069012 & 0.466847 & -0.023807 & -0.07450 \\ 0.069012 & 1.356091 & 0.304412 & 0.368689 & 0.278316 \\ 0.466847 & 0.304412 & 1.273936 & 0.359224 & -0.193665 \\ -0.023807 & 0.368689 & 0.359224 & 1.096486 & 0.099104 \\ -0.0745018 & 0.278316 & -0.193665 & 0.099104 & 1.661732 \end{pmatrix}.$$

$$\mathbf{A}_{16} = \begin{pmatrix} 1.947954 & 0.537389 & -0.153091 & 0.121248 & 0.209492 \\ 0.537389 & 1.633745 & 0.241763 & 0.397001 & 0.458412 \\ -0.153091 & 0.241763 & 1.661637 & 0.176707 & 0.488742 \\ 0.121248 & 0.397001 & 0.176707 & 1.519214 & 0.259799 \\ 0.209492 & 0.458412 & 0.488742 & 0.259799 & 1.433331 \end{pmatrix},$$

$$\mathbf{A}_{17} = \begin{pmatrix} 1.618807 & 0.108202 & 0.642453 & 0.020426 & -0.026291 \\ 0.108202 & 1.359896 & 0.395421 & 0.478153 & 0.443036 \\ 0.642453 & 0.395421 & 1.402719 & 0.458310 & -0.130384 \\ 0.020426 & 0.478153 & 0.458310 & 1.148681 & 0.174850 \\ -0.026291 & 0.443036 & -0.130384 & 0.174850 & 1.830438 \end{pmatrix}.$$

$$\mathbf{A}_{18} = \begin{pmatrix} 2.056424 & 0.789862 & -0.001053 & 0.286197 & 0.394551 \\ 0.789862 & 1.926544 & 0.356453 & 0.496827 & 0.655569 \\ -0.001053 & 0.356453 & 1.786167 & 0.239125 & 0.567990 \\ 0.286197 & 0.496827 & 0.239125 & 1.686871 & 0.386004 \\ 0.394551 & 0.655569 & 0.567990 & 0.386004 & 1.629354 \end{pmatrix},$$

$$\mathbf{A}_{19} = \begin{pmatrix} 1.802565 & 0.166987 & 0.905863 & 0.086776 & 0.046025 \\ 0.166987 & 1.365604 & 0.531934 & 0.642350 & 0.690117 \\ 0.905863 & 0.531934 & 1.595893 & 0.606940 & -0.035464 \\ 0.086776 & 0.642350 & 0.606940 & 1.226972 & 0.288470 \\ 0.046025 & 0.690117 & -0.035464 & 0.288470 & 2.083496 \end{pmatrix}.$$

SM2.2. Nonlinear equation test problems

As a case study for nonlinear problems we use a discretization of Chandrasekhar's H-function in the conservative case, which has also served as a test problem for nonlinear methods in Kelley (2003). We give the derivation by Kelley (2003) in the following. The H-function reads:

$$F(H)(\mu) = H(\mu) - \left(1 - \frac{c}{2}\int_0^1 \frac{\mu H(\nu)d\nu}{\mu+\nu}\right)^{-1} = 0. \tag{6}$$

A discretization of (6) uses the composite midpoint rule, i.e., (Kelley, 2003)

$$\int_0^1 f(\mu)d\mu = \frac{1}{N}\sum_{\iota=1}^{m} f(\mu_\iota), \tag{7}$$

to approximate the integral over $\iota = 1, \ldots, m$ discretization points (and $j = 1, \ldots, m$ likewise as a secondary index for the discretization points). With setting $\mu_\iota = \frac{(\iota-\frac{1}{2})}{m}$ and $x_j = H(\mu_j)$, the resulting discretized form for each component j of F can be written as (Kelley, 2003)

$$F(\mathbf{x})_j = x_j - \left(1 - \frac{c}{2m}\sum_{\iota=1}^{m}\frac{\mu_j x_\iota}{\mu_j+\mu_\iota}\right)^{-1}. \tag{8}$$

By defining a matrix $\mathbf{A}_c \in \mathbb{R}^{m\times m}$ with entries that depend on m and c, i.e.,

$$A_{c,j\iota} = \frac{c\mu_j}{2m\,(\mu_j+\mu_\iota)}, \tag{9}$$

we can simplify the above discretization to (Kelley, 2003)

$$(F(\mathbf{x}))_j = \mathbf{x}_j - (1 - (\mathbf{A}_c\mathbf{x})_j)^{-1}. \tag{10}$$

In one of Kelley's examples, $m = 10$ and $c = 0.9$ were used. We select these values as a basis for the experiments in our work. We fix $m = 10$ discretization points but we vary the parameter c of Equation (10) in $\{0.875, 0.905, 0.935\}$ in order to generate a set of training problems. $c \in \{0.85, 0.95\}$ are used for extrapolation. We further sample initial values from a normal distribution with mean $\tilde{\mathbf{x}}_0 = \mathbb{1}^{(m)}$, where $\mathbb{1}^{(m)}$ is an m-dimensional vector of ones, and variance of $\boldsymbol{\sigma} = 0.2 \cdot \mathbb{1}^{(m)}$, such that the entries of $\tilde{\mathbf{x}}_0$ are uncorrelated.

SM3. Additional numerical experiments

In this section, we present additional numerical experiments. These include a more detailed examination of the extrapolation studies done in Sections 4.1 and 4.2 (Section SM3.1), a comparison of the R2N2 to vanilla neural networks (Section SM3.2), and an ablation study for an R2N2 with iteration-dependent parameters (Section SM3.3).

SM3.1. Detailed generalization and extrapolation experiments for linear equation solving

We analyze the extrapolation results of the linear R2N2 shown in Figure 5 in more detail. First, we consider extrapolation in the space of RHS for the fixed matrix $\mathbf{A}_1$, i.e., problems $(\mathbf{A}_1, \mathbf{b}_i)$ with $\mathbf{b}_i$ from a range not used in training. Given that $\mathbf{b}_i$ were constructed by adding noise around a specific mean $\tilde{\mathbf{b}}$, straightforward experiments are to increase the noise or to generate samples around a different base $\tilde{\mathbf{b}}'$. However, as we already observed empirically (and will further show in Section SM4), the R2N2 is globally convergent towards solutions for all $\mathbf{b}$ given $\mathbf{A}_1$. We thus only analyze the performance for a set $\{\mathbf{b}'\}_i$ of 500 samples, where $\tilde{\mathbf{b}} = \mathbf{0}$ and the additive noise is $\mathbf{b}' \sim \mathcal{U}(-5 \cdot \mathbb{1}^m, 5 \cdot \mathbb{1}^m)$. We choose this interval, since the initial training set had $\|\tilde{\mathbf{b}}\| = 5$, and we want $\mathbf{r}_0$ to be of similar magnitude for all samples. The results, shown in Figure 3, show that the R2N2 converges for all of these problem instances. The spread between the minimum and the maximum at each iteration is tolerable too, with a difference in convergence rate of less than a factor of 2. Further, the mean of the training samples, plotted in orange, is only slightly better than that of the test sample of arbitrary RHS.

Next, we analyze application of the R2N2 to problems, where matrix $\mathbf{A}$ has a higher base noise than $\mathbf{A}_1$–$\mathbf{A}_3$. That is, we deploy the R2N2 on a test set generated from the original set of $\mathbf{b}_i$ and a pair of matrices with base entries $a_{ij} \sim \mathcal{N}(0, \sigma_l)$ for each $\sigma_l \in \{0.3, 0.5, 0.7, 1\}$, i.e., we generate a total of 8 new matrices, listed as $\mathbf{A}_{12}$–$\mathbf{A}_{19}$ in Section SM2.1. Figure 4 shows the performance of the R2N2 (compared to GMRES(r)). The green line with $\sigma_l = 0.1$ corresponds to the training settings. For smaller levels of noise, the performance of the R2N2 seems to decrease proportionally. However, towards $\sigma_l = 0.7$, there is a larger deterioration, where the R2N2 is still convergent but with a very slow convergence rate. Finally, at $\sigma_l = 1$, the R2N2 diverges.

The test problems we use in this paper overlay the noisy base entries of $\mathbf{A}$ with a spectrum $\boldsymbol{\lambda}$. We now examine how the convergence behavior of the R2N2 changes when $\boldsymbol{\lambda}$ is amplified or attenuated, respectively. For this study, we use matrices $\mathbf{A}_4$–$\mathbf{A}_7$ combined with the original sample of RHSs $\{\mathbf{b}_i\}$ to form a test set for the R2N2. In Figure 5, we analyze the resulting performance of the R2N2 on that test set. Changes to the overlaid spectrum in either direction reduce the performance of the R2N2. For $\mathbf{A}_7$, we even find divergent behavior, see Section SM4 for an explanation.

Finally, we generate four random, symmetric matrices $\mathbf{A}_8$ – $\mathbf{A}_{11}$. For each matrix we have drawn a random scalar $\sigma_l \sim \mathcal{U}(0, 5)$ and entries $\tilde{a}_{ij} \sim \mathcal{U}(0, \sigma_l)$ and then computed $\tilde{\mathbf{A}}\tilde{\mathbf{A}}^T$ to obtain an symmetric matrix. We again use the original sample of RHSs $\{\mathbf{b}_i\}$. The performance of the R2N2 on these random symmetric problems is plotted in Figure 6. The first three of these matrices lead to problem instances, on which the R2N2 diverges. The final matrix, $\mathbf{A}_{11}$, happened to have properties such that the R2N2 is actually able to converge to the solutions of the associated test set. We explain that this is a (lucky) property of $\mathbf{A}_{11}$ in Section SM4.

SM3.2. Comparison to vanilla neural networks

In this subsection, we train a vanilla NN on the same problem data as the R2N2. The NN has two hidden layers with 20 neurons each and uses tanh as the activation function for the hidden layers and linear for the output layer.

We examine the linear equation solving problems presented in Section 4.1. The NN is trained to learn $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{x}^\star$. To this end, the input layer concate-

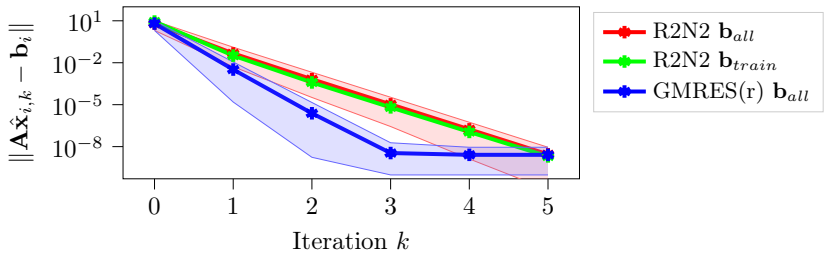


Fig. 3. Convergence of the R2N2 for linear problems with $\mathbf{A}_1$ and arbitrary right-hand sides from $[-5, 5]^m$ (denoted by $\mathbf{b}_{all}$). The solid line indicates the mean, whereas the area shows the spread between minimum and maximum performance. The R2N2 (mean) performance on the training distribution is given by the green line (denoted by $\mathbf{b}_{train}$), and is only slightly better than that for arbitrary RHS (red line).

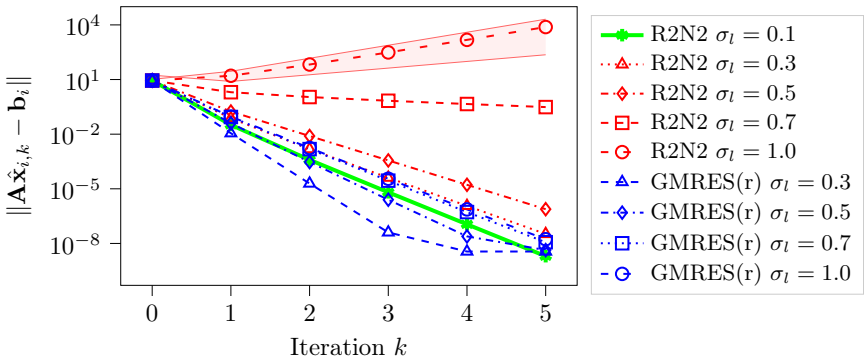


Fig. 4. Convergence of the R2N2 for linear problems with increased levels of noise in the base entries of $\mathbf{A}$, cf. Section SM2.1. Performance on the original training set is given by the green line. Compared to that, the noise level is gradually increased to a tenfold. The convergence of the R2N2, indicated by the red lines, decreases with increased noise until, eventually for noise level $\sigma_l = 1$, the R2N2 iterations lead to divergence on the test problems. We also plot the range between the minimum and maximum samples for $\sigma_l = 1$, highlighting that the divergence in mean is not caused by outliers.

nates $\mathbf{A}$ and $\mathbf{r}_0$, where $\mathbf{r}_0 = -\mathbf{b}$, to a $(m^2 + m)$-dimensional, i.e., 30-dimensional input vector. Owing to this large dimension, the NN has about one order of magnitude more trainable parameters and operations during inference than the R2N2. For training, we used *Adam* with default settings on 10,000 epochs. The training data set corresponds to the one used in Section 4.1. We have trained one NN, the feed-forward NN (*FFNN*), as a direct problem-to-solution mapping, i.e., on a single forward pass. Further, we trained a second NN, the recurrent (*RNN*), on 3 recurrent iterations using the same loss as for the R2N2, i.e., Equation (4.1) with $w_k = 4^k$. Here, the input to the first iteration uses $(\mathbf{A}, \mathbf{r}_0)$ with $\mathbf{r}_0 = -\mathbf{b}$. For subsequent iterations, we use the RNN output $\hat{\mathbf{x}}_k$ to compute $\mathbf{r}_k = \mathbf{A}\hat{\mathbf{x}}_k - \mathbf{b}$. The training of the FFNN has resulted in a final train loss of $1.28e^{-3}$ and test loss of $1.45e^{-3}$, whereas for the RNN we find a train loss of 4.669 coupled with a test loss of 4.688. Figure 7 compares the FFNN and the RNN to the R2N2 and GMRES on the test set of the training data, i.e., unseen samples of $\mathbf{b}_i$, and also for additional recurrences. The FFNN, which was only trained on one iteration, immediately incurs a drop in performance at the second iteration. The RNN performance, too, decreases when performing additional iterations $k > 3$. Clearly, neither the FFNN nor the RNN learn a behavior that can match the performance of the R2N2 despite possessing much more trainable parameters than the latter.

Next, we compare the vanilla NN models to the R2N2 on the discretized form of Chandrasekhar's H-function (Equation (4.4)). As a training set, we use the one generated for the R2N2 in Section 4.2. However, since the NN is not agnostic to the di-

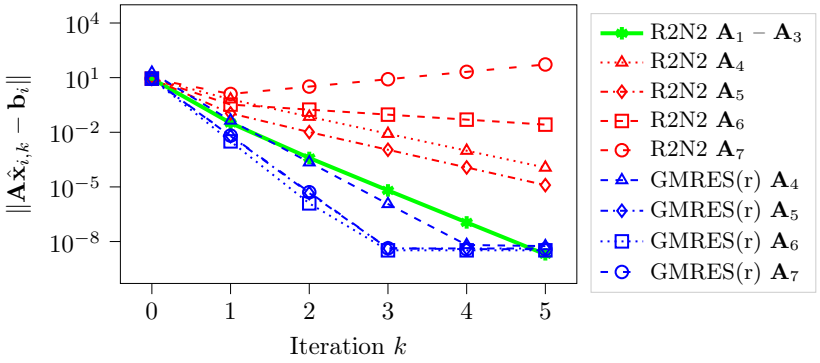


Fig. 5. Convergence of the R2N2 for linear problems when the diagonal entries of $\mathbf{A}$ are decreased or increased respectively. Performance on the original training set is given by the green line. For changes in either direction, the R2N2 performance (red lines) worsens. This underscores that the R2N2 is *adapted* to a certain problem class, from which we move away by the changes to the diagonals.

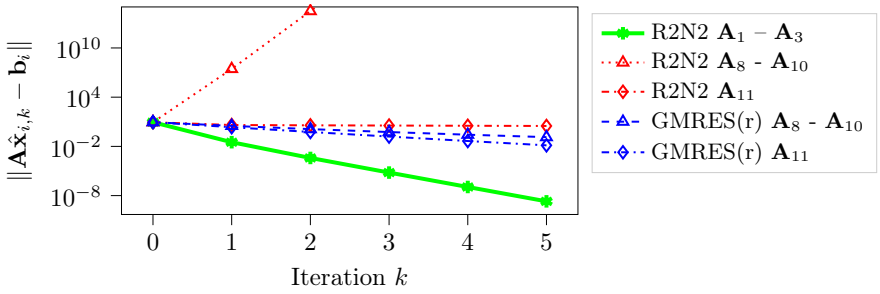


Fig. 6. Convergence of the R2N2 (shown by red lines) for linear problems formed from random symmetric matrices $\mathbf{A}_8$–$\mathbf{A}_{11}$. Performance on the original training set is given by the green line. In general, the R2N2 does not converge for such random problems that are not related to the training set. However, as $\mathbf{A}_{11}$ proves, coincidental convergence is possible.

mension of inputs $\mathbf{x}_0$, we only use the subset where $m = 10$ for training and comparison herein. We provide the NNs with the input $(\mathbf{x}_k, \boldsymbol{f}(\mathbf{x}_k))$, i.e., we allow for function evaluations to happen externally to the NNs. We train three different NNs: the *FFNN* is again trained on a single forward pass, the RNN_2 is trained on two recurrent iterations to match the R2N2, and the RNN_6 is trained on 6 consecutive iterations, such that it can learn to operate with the same amount of function evaluations as the R2N2 was allowed in this case. Owing to the comparatively large hidden layers, the NNs have about one or two orders of magnitude more trainable parameters and overall operations in a forward pass than the R2N2. Figure 8 compares the three NNs to the R2N2 and NK-GMRES on the test set of the training data, i.e., unseen samples of $\mathbf{x}_0$. Again, all three NNs achieve a partial reduction of the residual in accordance with the training objective, but cannot match the performance of the R2N2, especially over more than 2 iterations. The trained NNs are neither able to "predict" solutions to the test problems more accurately than the R2N2 or NK-GMRES, nor do they exhibit a behavior indicative of a solver, when applied recurrently. We therefore omit further studies and conclude that the R2N2 represents iterative solvers much better than vanilla-architecture neural networks that learn pure input-to-output mappings.

SM3.3. Multiple outer iterations of linear equation solving

In Section 4.1, we have explored the convergence of the R2N2 with a fixed $\boldsymbol{\theta}_n$ and a loss term summed over all outer iterations. With that configuration, the R2N2 was consistently outperformed by GMRES due to GMRES' minimization of the residual in the subspace. Here, we endow the R2N2 with varying $\boldsymbol{\theta}_{n,k}$ for iterations $k = 1, \ldots, T$ but the

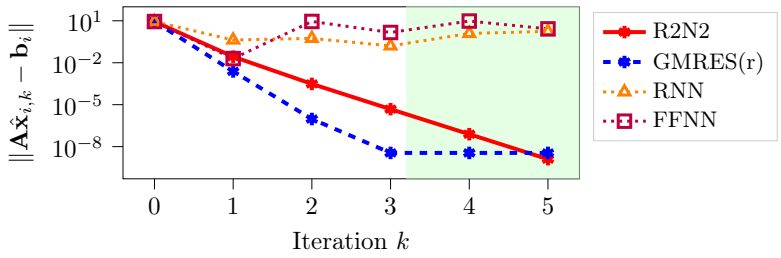


Fig. 7. Comparing the convergence behavior of the R2N2 (red, asterisk) against that of a FFNN (purple, square) and RNN (orange, triangle) trained on the same linear problem data, respectively. The FFNN is trained on a single forward pass, i.e., to minimize the residual at iteration $k = 1$. The RNN is trained on a weighted loss over iterations $k = \{1, 2, 3\}$. The green area denotes iterations for which the RNN, R2N2 (and FFNN) have not been trained. GMRES(r) (blue, asterisk) is added as a benchmark.

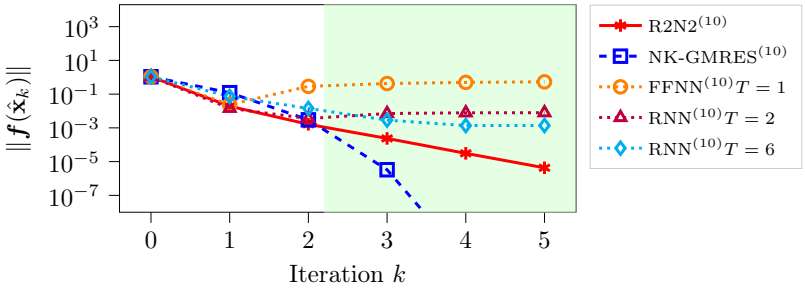


Fig. 8. Comparing the convergence behavior of the R2N2 (red, asterisk) against that of various NNs trained on the same nonlinear problem data. The FFNN (orange, circles) is trained on a single forward pass, i.e., to minimize the residual at iteration $k = 1$. RNN_2 (purple, triangles) is trained trained on a weighted loss over iterations $k = \{1, 2\}$, like the R2N2. RNN_6 (cyan, diamonds) is trained on a weighted loss over iterations $k = \{1, \dots, 6\}$, which allows to see (and operate on) the same number of function evaluations as the R2N2. NK-GMRES (blue, asterisk) is added as a benchmark.

training loss only considers the final iterate $\mathbf{x}_{i,T}$ of problem instances, i.e.,

$$MSE_f = \sum_{i=1}^{N} \left(\mathbf{A}\hat{\mathbf{x}}_{i,T} - \mathbf{b}_i\right)^2,$$

The results of this study are given by Figure 9. With the additional iteration-dependent parameters, the R2N2 appears to learn to combine consecutive iterations in order to capture the full space of the residual such that an abrupt step towards the true solution is facilitated in the final iteration. This is underscored by the change of behavior between $T = 2$ and $T = 3$. As soon as the aggregated dimension of the subspaces built during the iterations – controlled by $n \times T$ – exceeds the problem dimensionality $m = 5$, the R2N2 is able to achieve a sudden improvement in performance, compared to GMRES. The level of that improvement, however, stagnates for $T > \frac{m}{n}$. The remaining residual presumably originates from the variance of the training data, since the R2N2 cannot adapt to individual problem instances. Evaluating the R2N2 for further iterations, which we performed by reapplying the parameters of the final step $\boldsymbol{\theta}_{n,T}$, does not lead to convergence for the steps $k > T$. Therefore, by the addition of more trainable parameters, which corresponds to a relaxation of the prior structure of the R2N2, the R2N2 behaves more comparable to the NNs analyzed in Section SM3.2. That is, improvements over GMRES in the training objective become possible, at the cost of forfeiting the ability to extrapolate or converge to solutions.

SM4. Convergence properties of the R2N2 on linear problems

We analyze the convergence properties of the R2N2 in more detail. Results in Section 4.1, Figure 5, have demonstrated that, empirically, the R2N2 can

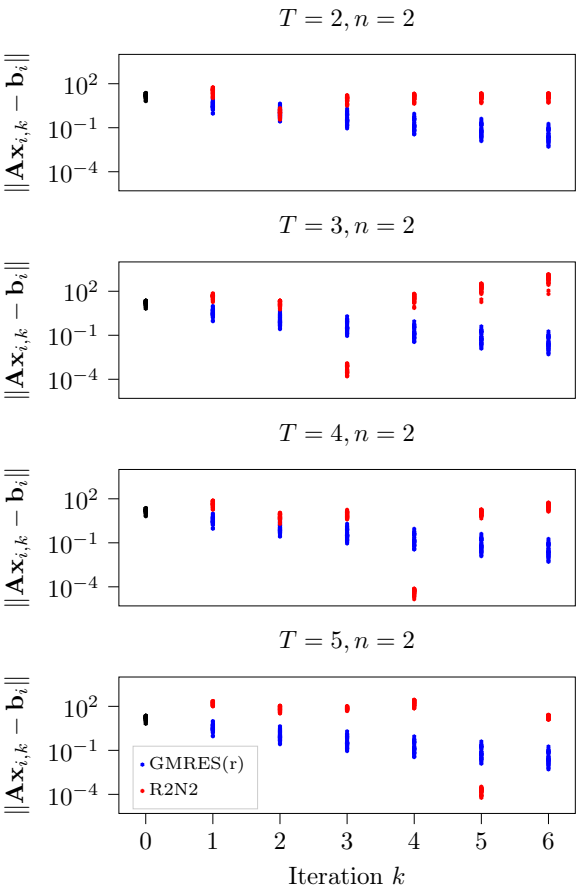


Fig. 9. Convergence of the R2N2 vs GMRES for 6 nonlinear iterations of solving $\mathbf{A}\mathbf{x} = \mathbf{b}_i$ for a random draw of samples $\mathbf{b}_i$. The generation of problem data follows the procedure described in (**SM2**). Both methods use 2 inner iterations. The different R2N2 used in the subplots were trained on the loss after $T = \{2, 3, 4, 5\}$ nonlinear iterations. The parameters in the final module, $\boldsymbol{\theta}_{n,k}$, were relaxed to take varying values across iterations $k \leq T$.

converge towards the solution of a linear problem for arbitrary right-hand sides $\mathbf{b}_i$ in some cases. Evidently, the property is then due to the matrix $\mathbf{A}$ and the coefficients $\boldsymbol{\theta}$ of the R2N2.

For linear problems, it is straightforward to write how the R2N2 acts on the (sequence of) residuals $\mathbf{r}_k$, given that $\mathbf{A}$ is a square and invertible matrix. We write the forward pass as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\zeta}\mathbf{V}, \tag{11}$$

where

$$\boldsymbol{\zeta}\mathbf{V} = \sum_{j=0}^{n-1} \mathbf{A}^j \mathbf{r}_k \zeta_{j+1}, \tag{12}$$

Note that the Krylov subspace in the R2N2 uses coefficients $\boldsymbol{\theta}$ instead of $\boldsymbol{\zeta}$ to express the linear combination. $\boldsymbol{\zeta}$ do not correspond to the final layer $\boldsymbol{\theta}_n$ in Equation (3.2). Instead, the whole of $\boldsymbol{\theta}$ can be transformed into $\boldsymbol{\zeta}$. We use the notation with $\boldsymbol{\zeta}$ here for more clarity.

Next, we also state the forward pass in terms of the residual inserting Equation (11), i.e.,

$$\mathbf{r}_{k+1} = \mathbf{A}\mathbf{x}_{k+1} - b = \mathbf{A}\mathbf{x}_k + \mathbf{A}\boldsymbol{\zeta}\mathbf{V}. \tag{13}$$

Further, by inserting for $\mathbf{x}_k = \mathbf{A}^{-1}(\mathbf{r}_k + \mathbf{b})$ and Equation (12), we obtain

$$\mathbf{r}_{k+1} = \mathbf{A}\mathbf{A}^{-1}(\mathbf{r}_k + \mathbf{b}) + \mathbf{A}\sum_{j=0}^{n-1} \mathbf{A}^j \mathbf{r}_k \zeta_{j+1} - \mathbf{b},$$

which, after canceling $\mathbf{A}\mathbf{A}^{-1}$ and $\mathbf{b}$, simplifies to

$$\mathbf{r}_{k+1} = \left[\mathbf{I} + \sum_{j=1}^{n} \mathbf{A}^j \zeta_j\right] \mathbf{r}_k =: \mathcal{A}^{iter}(\mathbf{r}_k), \tag{14}$$

where $\mathcal{A}^{iter}$ is the *algorithm operator* encoded by the R2N2 with parameters $\boldsymbol{\zeta}$ (or $\boldsymbol{\theta}$, respectively) for a specific input matrix $\mathbf{A}$. If the spectral norm of $\mathcal{A}^{iter}$ (largest singular value) is smaller than 1, the R2N2 will converge for all right-hand sides. For some $\mathbf{A}$, and given parameters $\boldsymbol{\theta}$, we compute this norm via singular value decomposition. For instance, the spectral norm of $\mathcal{A}^{iter}$ computed for matrix $\mathbf{A}_1$ and the trained parameters $\boldsymbol{\theta}$ of the R2N2 used for the results in Section 4.1 is 0.0163, which confirms that the R2N2 will converge for all RHS for $\mathbf{A}_1$. On the other hand, $\mathcal{A}^{iter}$ for matrix $\mathbf{A}_6$ (see Figure 5) has spectral norm 0.5315, which coincides with the slower convergence observed for $\mathbf{A}_6$. For $\mathbf{A}_7$ we even have spectral norm 2.5397 and the R2N2 does indeed diverge on problem instances featuring $\mathbf{A}_7$. Finally, when analyzing the random symmetric matrix $\mathbf{A}_{11}$ (see Figure 6), we discover that its associated operator coincidentally has spectral norm 0.9948, which again coincides with the observed slow convergence properties.

Equation (14) does not only allow to check for which problem instances a trained R2N2 is applicable. Instead, it further lets us derive conditions on $\boldsymbol{\theta}$ such that the resulting R2N2 is globally convergent on a linear problem class defined by one specific $\mathbf{A}$ or a set of $\mathbf{A}$. This should be explored in future work.

Bibliography

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., and Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*.

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

Kelley, C. T. (1995). *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics.

Kelley, C. T. (2003). *Solving nonlinear equations with Newton's method*. SIAM.