© 2020 Society for Industrial and Applied Mathematics

Layer-Parallel Training of Deep Residual Neural Networks[*]

Stefanie Günther[†], Lars Ruthotto[‡], Jacob B. Schroder[§], Eric C. Cyr[¶], and Nicolas R. Gauger[†]

Abstract. Residual neural networks (ResNets) are a promising class of deep neural networks that have shown excellent performance for a number of learning tasks, e.g., image classification and recognition. Mathematically, ResNet architectures can be interpreted as forward Euler discretizations of a nonlinear initial value problem whose time-dependent control variables represent the weights of the neural network. Hence, training a ResNet can be cast as an optimal control problem of the associated dynamical system. For similar time-dependent optimal control problems arising in engineering applications, parallel-in-time methods have shown notable improvements in scalability. This paper demonstrates the use of those techniques for efficient and effective training of ResNets. The proposed algorithms replace the classical (sequential) forward and backward propagation through the network layers with a parallel nonlinear multigrid iteration applied to the layer domain. This adds a new dimension of parallelism across layers that is attractive when training very deep networks. From this basic idea, we derive multiple layer-parallel methods. The most efficient version employs a simultaneous optimization approach where updates to the network parameters are based on inexact gradient information in order to speed up the training process. Using numerical examples from supervised classification, we demonstrate that the new approach achieves a training performance similar to that of traditional methods, but enables layer-parallelism and thus provides speedup over layer-serial methods through greater concurrency.

Key words. deep learning, residual networks, supervised learning, optimal control, layer-parallelization, parallel-in-time, simultaneous optimization

AMS subject classifications. 49K15, 68T01, 68W10

DOI. 10.1137/19M1247620

[†]Scientific Computing Group, TU Kaiserslautern, Kaiserslautern 67663, Germany (stefanie.guenther@scicomp.uni-kl.de, nicolas.gauger@scicomp.uni-kl.de).
[‡]Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322 (lruthotto@emory.edu).
[§]Department of Mathematics and Statistics, University of New Mexico, Albuquerque, NM 87131 (jbschroder@unm.edu).
[¶]Computational Mathematics Department, Sandia National Laboratories, Albuquerque, NM 87185-1320 (eccyr@sandia.gov).

1. Introduction. One of the most promising areas in artificial intelligence is deep learning, a form of machine learning that uses neural networks containing many hidden layers [4, 45]. Deep neural networks (DNNs) and, in particular, deep residual networks (ResNets) [37], have been breaking human records in various contests and are now central to technologies such as image recognition [39, 44, 45] and natural language processing [6, 15, 42].

The abstract goal of machine learning is to model a function $f : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^m$ and train its parameter $\boldsymbol{\theta} \in \mathbb{R}^p$ such that

$$(1.1) \qquad\qquad\qquad f(\mathbf{y}, \boldsymbol{\theta}) \approx \mathbf{c}$$

for input-output pairs $(\mathbf{y}, \mathbf{c})$ from a certain data set $\mathcal{Y} \times \mathcal{C}$. Depending on the nature of the inputs and outputs, the task can be regression or classification. When outputs are available for all samples, or parts of the samples, or when they are not available, this formulation describes supervised, semisupervised, and unsupervised learning, respectively. The function f can be thought of as an interpolation or approximation function.

In deep learning, the function f involves a DNN that aims at transforming the input data using many layers. The layers successively apply affine transformations and element-wise nonlinearities that are parametrized by the network parameters $\boldsymbol{\theta}$. The training problem consists of finding the parameters $\boldsymbol{\theta}$ such that (1.1) is satisfied for data elements from a training data set but also holds for previously unseen data from a validation data set, which has not been used during training. The former objective is commonly modeled as an expected loss, and optimization techniques are used to find the parameters that minimize the loss.

Despite rapid methodological developments, compute times for training state-of-the-art DNNs can still be prohibitive, measured in the order of hours or days, involving hundreds or even thousands of layers and millions or billions of network parameters [16, 43]. There is thus a great interest in increasing parallelism to reduce training runtimes. The most common approach involves *data-parallelism*, where elements of the training data set are distributed onto multiple compute units. Synchronous and asynchronous data-parallel training algorithms have been developed to coordinate the network parameter updates [40, 1]. Another approach is referred to as *model-parallelism*, which aims at partitioning different layers of the network and its parameters to different compute units. Model-parallelism has traditionally been used when the network dimension exceeds available memory of a single compute unit. Often, a combination of both approaches is employed [36, 16].

However, none of the above approaches to parallelism tackles the scalability barrier created by the intrinsically serial propagation of data through the network itself. In either of the above approaches, each subsequent layer can process accurate information only after the previous layer has finished its computation. As a result, training runtimes typically scale linearly with the number of layers. As current state-of-the-art networks tend to increase complexity by adding more and more layers (see, e.g., the ResNet-1001 with 1001 layers and 10.2 million weights in [38]), the serial layer propagation creates a serious bottleneck for fast and scalable training algorithms seeking to leverage modern high-performance computing (HPC) facilities.

In this paper, we address the above scalability barrier by introducing concurrency across the network layers. To this end, we replace the serial data propagation through the network layers with a nonlinear multigrid method that treats layers, or layer chunks, simultaneously and thus enables full layer-parallelism. Our goal is to have a training methodology that

is scalable in the number of layers, e.g., doubling the number of layers and the number of compute resources should result in a nearly constant runtime. To achieve this, we leverage recent advances in parallel-in-time integration methods for unsteady differential equations.[1]

The forward propagation through a ResNet can be seen as a discretization of a time-dependent ordinary differential equation (ODE), which was first observed in [34, 18, 35]. Interpreting the network propagation as a nonlinear dynamical system has since attracted increasing attention (see, e.g., [48] and references therein, or [14]).

Based on this interpretation, we employ a multigrid reduction in time (MGRIT) approach [19] that divides the time domain—which, in this interpretation, corresponds to the layer domain—into multiple time chunks that can be processed in parallel on different compute units. Coupling of the chunks is achieved through a coarse-grid correction scheme that propagates information across chunk interfaces on a coarser time- (i.e., layer-) grid. The method can be interpreted as a parallelization of the model, processing layer chunks simultaneously within the iterative multigrid scheme, thus breaking the traditional layer-serial propagation. At convergence, the iterative multigrid scheme solves the same problem as a layer-serial method, and thus it can be utilized in any common gradient-based optimization technique to update the network parameters, such as stochastic gradient descent (SGD) or other batch approaches, without loss of accuracy. Further, it can be applied in addition to any data-parallelism across the data set elements, thus multiplying data-parallel runtime speedup. Runtime speedup over traditional layer-serial methods is achieved through the new dimension of parallelization across layers, enabling greater concurrency.

The addition of layer-parallelism allows for ResNets to take advantage of large machines currently programmed with message-passing style parallelism. The use of such large machines in conjunction with a multilevel training algorithm scalable in the number of layers opens the door to training networks with thousands, or possibly even millions, of layers. We demonstrate the feasibility of such an approach by using the parallel-in-time package XBraid [53] with a ResNet on large clusters. Additionally, the nonintrusive approach of XBraid would allow for any node-level optimizations (such as those utilizing GPUs) to be used.

The iterative nature of the multigrid approach further enables the use of simultaneous optimization algorithms for training the network. Simultaneous optimization methods have been widely used for optimization problems that are constrained by partial differential equations (PDEs), where they show promise for reducing the runtime overhead of the optimization when compared to a pure simulation of the underlying PDE (see, e.g., [7, 54, 5] and references therein). They aim at solving the optimization problem in an all-at-once fashion, updating the optimization parameters simultaneously while solving for the time-dependent system state. Here, we apply the *one-shot* method [9, 33] to solve the training problem simultaneously for the network state and parameters. In this approach, network parameter updates are based on inexact gradient information resulting from early stopping of the layer-parallel multigrid iteration.

The paper is structured as follows. Section 2 gives an introduction to the deep learning optimization problem and its interpretation as an optimal control problem. Further, it discusses

---

[1]For an introduction and overview on various parallel-in-time integration schemes for unsteady differential equations, we refer the reader to the review paper [22] and to more recent developments, such as [23, 29].

numerical discretization of the optimal control problem and summarizes necessary conditions for optimality. Section 3 introduces the layer-parallel multigrid approach replacing the forward and backward propagation through the network. Section 4 focuses on the integration of the layer-parallel multigrid scheme into a simultaneous optimization algorithm. Numerical results demonstrating the feasibility and runtime benefits of the proposed layer-parallel scheme are presented in section 5.

2. Deep learning as a dynamic optimal control problem. In this section, we present an optimal control formulation of a supervised classification problem using a deep residual network. Limiting the discussion to this specific task allows us to provide a self-contained mathematical description. We note that the layer-parallel approach can be extended to other learning tasks, e.g., semisupervised learning, autoregression, or recurrent learning; see [28, 2] for a general introduction and a comprehensive overview of deep learning techniques.

2.1. Optimal control formulation. In supervised classification, the given data set consists of s feature, or example, vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_s \in \mathbb{R}^{n_f}$ and associated class probability vectors $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_s \in \Delta_{n_c}$, where Δ_{n_c} denotes the unit simplex in $\mathbb{R}^{n_c}$, and $n_f, n_c \in \mathbb{N}$ denote the number of features and classes in the given data set, respectively. The jth component of $\mathbf{c}_k$ represents the probability of example $\mathbf{y}_k$ belonging to the jth class. The learning problem aims at training a network and its classifier that approximate the feature-to-class mapping for all data elements.

A powerful class of networks are residual neural networks (ResNets) [37]. In an abstract form, the network transformation to a generic input data example $\mathbf{y}$ using an N-layer ResNet can be written as

$$(2.1) \qquad \mathbf{u}^{n+1} = \mathbf{u}^n + hF(\mathbf{u}^n, \boldsymbol{\theta}^n) \quad \text{for } n = 0, 1, \ldots, N-1, \quad \text{with} \quad \mathbf{u}_0 = \mathbf{L}_{\text{in}}\mathbf{y},$$

with $\mathbf{u}^n \in \mathbb{R}^q$, q being the network width. The transformations in F typically consist of affine linear and elementwise nonlinear transformations that are parametrized by the entries in the layer weights $\boldsymbol{\theta}^0, \ldots, \boldsymbol{\theta}^{N-1} \in \mathbb{R}^d$, respectively. For simplicity, we consider the single-layer perceptron model

$$(2.2) \qquad F(\mathbf{u}, \boldsymbol{\theta}) = \sigma(\mathbf{K}(\boldsymbol{\theta}^{(1)})\mathbf{u} + \mathbf{B}\boldsymbol{\theta}^{(2)}),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is a nonlinear activation function that is applied componentwise, e.g., $\sigma(x) = \tanh(x)$ or $\sigma(x) = \max\{x, 0\}$. Here, each weight vector $\boldsymbol{\theta} \in \{\boldsymbol{\theta}^0, \ldots, \boldsymbol{\theta}^{N-1}\}$ is partitioned into one part that defines a linear operator $\mathbf{K} \in \mathbb{R}^{q \times q}$ and another part that represents coefficients of a bias with respect to columns of the given matrix $\mathbf{B} \in \mathbb{R}^{q \times n_b}$ (e.g., $n_b = 1$ and $\mathbf{B} = \mathbf{e}_q$, a vector of all ones, to add a constant shift to all features). In this work, we assume that the linear operators $\mathbf{K}(\cdot)$ either are dense matrices or correspond to convolutional operators (see [46]) parametrized by $\boldsymbol{\theta}^{(1)}$, whose entries we determine in the training. However, our method can be extended to other parametrizations (e.g., the layer used in [37], which features two affine transformations, or the layer based on an antisymmetric matrix suggested in [34]). While $\mathbf{K}(\cdot)$ needs to be a square matrix, we use a nonsquare model for the operator $\mathbf{L}_{\text{in}} \in \mathbb{R}^{q \times n_f}$ to map the data set elements to the network width.

Considering a small but positive h in (2.1), it is intuitive to interpret the ResNet propagation (2.1) as a forward Euler discretization of the initial value problem

$$(2.3) \qquad \partial_t \mathbf{u}(t) = F(\mathbf{u}(t), \boldsymbol{\theta}(t)), \quad t \in [0, T], \quad \text{with} \quad \mathbf{u}(0) = \mathbf{L}_{\text{in}} \mathbf{y}.$$

In this formulation, t is an artificial time that refers to the propagation of the input features through the neural network, going from the input layer with $\mathbf{u}(0)$ to the network output $\mathbf{u}(T)$ being the solution of the initial value problem evaluated at some final time T. In contrast to the discrete ResNet propagation, the dynamical system continuously transforms the network state $\mathbf{u}(t)$ by prescribing its time-derivative with the vector field F, whose parameters $\boldsymbol{\theta}(t)$ will be learned during training.

In order to classify the network output into a specific class, a hypothesis function is required that predicts the class probabilities. Here, we limit ourselves to multinomial regression models, which are common in deep learning. To this end, we consider the *softmax hypothesis function* given by

$$(2.4) \qquad S(\mathbf{u}(T), \mathbf{W}, \boldsymbol{\mu}) = \frac{1}{\mathbf{e}_{n_c}^{\top} \exp(\mathbf{z})} \exp(\mathbf{z}), \quad \mathbf{z} = \mathbf{W} \mathbf{u}(T) + \boldsymbol{\mu},$$

where $\mathbf{e}_{n_c} \in \mathbb{R}^{n_c}$ is a vector of all ones, exp is the exponential function applied elementwise, and $\mathbf{W} \in \mathbb{R}^{n_c \times q}$, $\boldsymbol{\mu} \in \mathbb{R}^{n_c}$ denote a weight matrix and a bias vector, both of whose entries need to be learned in training alongside the network parameters $\boldsymbol{\theta}(t)$.

The performance of the network transformation and classification can then be measured by comparing the predicted class probabilities to the given ones in $\mathbf{c}$. To this end, we use the *cross entropy loss function*

$$(2.5) \qquad \ell(\mathbf{u}(T), \mathbf{c}, \mathbf{W}, \boldsymbol{\mu}) = -\mathbf{c}^{\top} \log(S(\mathbf{u}(T), \mathbf{W}, \boldsymbol{\mu})).$$

The training problem consists of minimizing the average cross entropy loss function over many examples with respect to $\boldsymbol{\theta}(t), \mathbf{W}$, and $\boldsymbol{\mu}$. It can thus be cast as the following continuous-in-time optimal control problem:

$$(2.6) \qquad \min \quad \frac{1}{s} \sum_{k=1}^{s} \ell(\mathbf{u}_k(T), \mathbf{c}_k, \mathbf{W}, \boldsymbol{\mu}) + \int_0^T R(\boldsymbol{\theta}(t), \mathbf{W}, \boldsymbol{\mu}) \, dt$$

$$(2.7) \qquad \text{subject to} \quad \partial_t \mathbf{u}_k(t) = F(\mathbf{u}_k(t), \boldsymbol{\theta}(t)) \quad \forall t \in [0, T],$$

$$(2.8) \qquad \qquad \qquad \quad \mathbf{u}_k(0) = \mathbf{L}_{\text{in}} \mathbf{y}_k \qquad \forall k = 1, \ldots, s.$$

The optimal control problem aims at finding control variables $\boldsymbol{\theta}(t)$ (the network weights) and corresponding state variables $\mathbf{u}_k(t)$ (the network states) that minimize the objective function, while satisfying the constraints (2.7)–(2.8) (the network dynamics) for all data set elements $k = 1, \ldots, s$. The objective function consists of the empirical cross entropy loss function evaluated at the final time T, and an additional regularization term denoted by R. In conventional deep learning approaches, R typically applies a Tikhonov regularization penalizing "large" network and classification parameters, measured in a chosen norm. Within the time-continuous optimal control interpretation, we additionally penalize the time-derivative of $\boldsymbol{\theta}(t)$ in order to ensure weights that vary smoothly in time. This is an important ingredient for stability analysis [34].

2.2. Discretization of the optimal control problem. We solve the time-continuous optimal control problem in a first-discretize-then-optimize fashion. We discretize the control $\boldsymbol{\theta}(t)$ and the states $\mathbf{u}(t)$ at regularly spaced time points $t_n = n \cdot h$, where $h = T/N$ and $n = 0, 1, \ldots, N$. In this setting, each discrete state $\mathbf{u}^n$ and control $\boldsymbol{\theta}^n$ corresponds to the nth layer of the network. This leads to the discrete control problem

$$(2.9) \qquad \min \quad \frac{1}{s} \sum_{k=1}^{s} \ell(\mathbf{u}_k^N, \mathbf{c}_k, \mathbf{W}, \boldsymbol{\mu}) + \sum_{n=0}^{N-1} R(\boldsymbol{\theta}^n, \mathbf{W}, \boldsymbol{\mu})$$

$$(2.10) \qquad \text{subject to} \quad \mathbf{u}_k^{n+1} = \Phi(\mathbf{u}_k^n, \boldsymbol{\theta}^n) \quad \forall n = 0, \ldots, N-1,$$

$$(2.11) \qquad \qquad \mathbf{u}_k^0 = \mathbf{L}_{\text{in}} \mathbf{y}_k \qquad \forall k \in 1, \ldots, s.$$

In this general description, Φ can denote any layer-to-layer propagator which maps data $\mathbf{u}^n$ to the next layer. In the case of a forward Euler discretization, it reads

$$(2.12) \qquad \Phi(\mathbf{u}^n, \boldsymbol{\theta}^n) = \mathbf{u}^n + hF(\mathbf{u}^n, \boldsymbol{\theta}^n),$$

giving the ResNet propagation as in (2.1). However, the time-continuous interpretation of the network propagation permits to employ other, possibly more stable, discretization schemes of the initial value problem (2.3) (see [34]) and thus opens the door to new network architecture designs. It also allows for discretization of the controls and states at different time points, which can improve the efficiency and will be the subject of future research. Further, numerical advances for solving the corresponding optimal control problem can be leveraged, such as the time-parallel approach, which is discussed in this paper.

2.3. Necessary optimality conditions. The necessary conditions for optimality of the discrete, equality-constrained optimization problem (2.9)–(2.11) can be derived from the associated Lagrangian function

$$(2.13) \qquad L := J + \sum_{k=1}^{s} \left[\sum_{n=0}^{N-1} \left(\bar{\mathbf{u}}_k^{n+1} \right)^T \left(\Phi(\mathbf{u}_k^n, \boldsymbol{\theta}^n) - \mathbf{u}_k^{n+1} \right) + \left(\bar{\mathbf{u}}_k^0 \right)^T \left(\mathbf{L}_{\text{in}} \mathbf{y}_k - \mathbf{u}_k^0 \right) \right],$$

where J denotes the objective function in (2.9) and $\bar{\mathbf{u}}_k^n$ are called adjoint variables for layer $n = 0, \ldots, N$ and example $k = 1, \ldots, s$. Optimal points of the problem are saddle points of the Lagrangian function (see, e.g., [49]); thus equating its partial derivatives with respect to all state, adjoint, and control variables to zero yields the following necessary conditions for optimality:
  1. *State equations*

$$(2.14) \qquad \mathbf{u}_k^{n+1} = \Phi(\mathbf{u}_k^n, \boldsymbol{\theta}^n) \quad \forall n = 0, \ldots, N-1,$$

$$(2.15) \qquad \text{with} \quad \mathbf{u}_k^0 = \mathbf{L}_{\text{in}} \mathbf{y}_k \qquad \forall k = 1, \ldots, s.$$

  2. *Adjoint equations*

$$(2.16) \qquad \bar{\mathbf{u}}_k^n = \left(\partial_{\mathbf{u}} \Phi(\mathbf{u}_k^n, \boldsymbol{\theta}^n) \right)^T \bar{\mathbf{u}}_k^{n+1} \quad \forall n = 0, \ldots, N-1,$$

$$(2.17) \qquad \text{with} \quad \bar{\mathbf{u}}_k^N = \frac{1}{s} \left(\partial_{\mathbf{u}} \ell(\mathbf{u}_k^N, \mathbf{c}_k, \mathbf{W}, \boldsymbol{\mu}) \right)^T \quad \forall k = 1, \ldots, s.$$

3. *Design equations*

$$(2.18) \qquad 0 = \sum_{k=1}^{s} (\partial_{\boldsymbol{\theta}^n} \Phi(\mathbf{u}_k^n, \boldsymbol{\theta}^n))^T \bar{\mathbf{u}}_k^{n+1} + (\partial_{\boldsymbol{\theta}^n} R)^T \quad \forall n = 0, \dots, N-1$$

$$(2.19) \qquad 0 = \frac{1}{s} \sum_{k=1}^{s} \left(\partial_{\mathbf{W}, \boldsymbol{\mu}} \ell(\mathbf{u}_k^N, \mathbf{c}_k, \mathbf{W}, \boldsymbol{\mu}) \right)^T + (\partial_{\mathbf{W}, \boldsymbol{\mu}} R)^T .$$

Here, subscripts denote partial derivatives, $\partial_x = \frac{\partial}{\partial x}$. Training a residual network corresponds to the attempt of solving the above set of equations for the special choice of Φ being the forward Euler time-integration scheme. However, the above equations, as well as the discussions in the remainder of this paper, are general, in the sense that any other layer-to-layer propagator Φ can be utilized that corresponds to the discretization of the dynamical system (2.3).

The state equations correspond to the forward propagation of input examples $\mathbf{y}_k$ through the network layers. The adjoint equations propagate partial derivatives with respect to the network states backward through the network layers, starting from a terminal condition at N equal to the local derivative of the loss function. In a time-continuous setting, the adjoint equations correspond to the discretization of an additional adjoint dynamical system for propagating network state derivatives backward in time.[2] We note that solving the above adjoint equations backward in time is equivalent to the *backpropagation* method that is established within the deep learning community for computing the network gradient [46]. It further corresponds to the reverse mode of automatic differentiation [30]. The adjoint variables are utilized in the right-hand side of the design equations, which then form the so-called reduced gradient. For feasible state and adjoint variables, the reduced gradient holds the total derivative, i.e., the sensitivity, of the objective function with respect to the controls. It is thus used within gradient-based optimization methods for updating the network controls.

3. Layer-parallel multigrid approach. In order to achieve concurrency across all of the network layers, we replace the sequential propagation through the residual network (forward and backward) with an iterative multigrid scheme.

Based on the time-continuous nonlinear ODE interpretation of ResNets as in (2.3), and its time-discretization as in (2.10)–(2.11), we employ the MGRIT [19] method to parallelize across the time domain of the network. While the discussion in this section revolves around time grids, here each time point is considered a layer in the network. Thus, the multigrid approach constructs a multilevel hierarchy, where each level is a network containing fewer layers (i.e., fewer time points). The coarsest level will contain only a handful of layers, while the finest level could contain thousands of layers (or more). When run in parallel, each compute unit will own only a few fine-grid layers, thus allowing for massive parallelism to be applied to the learning algorithm.

The MGRIT scheme was introduced in [19] and first applied to neural networks in [50], although that work considered parallelism over epochs of the training algorithm, rather than over layers. We refer the reader to the works [19, 20, 32] for the details of the method, but here we provide a self-contained overview of the MGRIT scheme.

---

[2]The adjoint approach is a common and well-established method in optimal control that provides gradient information at computational costs that are independent of the design space dimension; see, e.g., [26]

3.1. Multigrid across layers for forward propagation. Consider the network states to be collected in a vector $\mathbf{U} = (\mathbf{u}^0, \mathbf{u}^1, \ldots, \mathbf{u}^N)$. The forward propagation through the network (2.14)–(2.15) can then be written as the space-time system

$$(3.1) \qquad A(\mathbf{U}, \boldsymbol{\theta}) := \begin{pmatrix} \mathbf{u}^0 \\ \mathbf{u}^1 - \Phi(\mathbf{u}^0, \boldsymbol{\theta}^0) \\ \vdots \\ \mathbf{u}^N - \Phi(\mathbf{u}^{N-1}, \boldsymbol{\theta}^{N-1}) \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{\text{in}}\mathbf{y} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{pmatrix} =: \mathbf{G},$$

where each block row corresponds to a time step, which in turn corresponds to a layer in the network. Here, the $\mathbf{u}^n$ denote the network states at each time step for either a single generic input vector $\mathbf{y}$ or for a batch, i.e., a subset, of input vectors $\mathbf{y}_k$, $k \in \mathcal{S} \subset \{1, \ldots, s\}$.

Sequential time-stepping solves (3.1) through forward substitution, i.e., forward propagation of input data through the network layers. In contrast, MGRIT solves (3.1) iteratively, beginning with some initial solution guess for $\mathbf{U}$, by using the full approximation storage (FAS) nonlinear multigrid method [11]; see section 3.1.1. In both cases, the exact same equations are solved, and thus the same solution is reached (in the case of MGRIT, to within a user tolerance). Regarding cost, sequential time-stepping is $O(N)$ but sequential. Instead, MGRIT solves this system with an $O(N)$ multigrid method with a larger computational constant but with parallelism in the layer dimension. This parallelism allows for a distributed workload, processing multiple layers in parallel on multiple compute units. Typically, a certain number of processors is needed for MGRIT to show a speedup over layer-serial forward propagation. This is referred to as the cross-over point. However, the speedups observed can be large; e.g., the work [33] showed a speedup of $19\times$ for a model optimization problem while using an additional 256 processors in time.

3.1.1. MGRIT using full approximation scheme. Similar to linear multigrid methods, the nonlinear FAS method computes coarse-grid error corrections to fine-grid approximations of the solution. Each iteration of the nonlinear MGRIT scheme consists of three steps. First, a relaxation scheme is employed to cheaply compute an approximation to the true solution on the fine grid. Second, the current error is approximated on a coarser grid by solving a coarse-grid residual equation. Third, the interpolated coarse-grid error approximation is used to correct the current fine-grid solution approximation. This idea is based on the fact that low frequency error components can be reduced with relaxation much faster on coarser grids. While a general introduction to linear and nonlinear multigrid methods can be found in [12], here we explain each of the algorithmic components of MGRIT, starting with the coarse-grid residual equation.

Let $\mathbf{U}$ denote an approximation to the true solution $\mathbf{U}_*$ of (3.1) such that $\mathbf{U}_* = \mathbf{U} + \mathbf{E}$, with $\mathbf{E}$ denoting the current error. Then this error can be expressed in terms of the residual $\mathbf{R}$ as

$$(3.2) \qquad \mathbf{R} := \mathbf{G} - A(\mathbf{U}, \boldsymbol{\theta}) = A(\mathbf{U}_*, \boldsymbol{\theta}) - A(\mathbf{U}, \boldsymbol{\theta})$$
$$(3.3) \qquad = A(\mathbf{U} + \mathbf{E}, \boldsymbol{\theta}) - A(\mathbf{U}, \boldsymbol{\theta}).$$

In a multigrid setting, the residual equation (3.3) is solved on a coarser grid such that an approximation to the error $\mathbf{E}$ can be computed more cheaply than on the fine grid. In
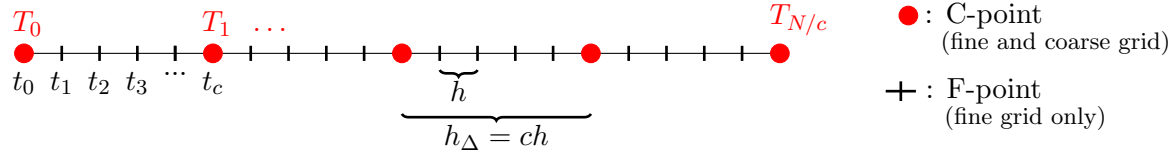
Figure 1. *Fine grid (t_i) and coarse grid (T_j) for coarsening factor $c = 5$. MGRIT eliminates the fine points (black vertical lines, F-points) to yield a coarse level composed of the red circles (C-points).*

linear cases, i.e., when A is linear in $\mathbf{U}$, the residual equation reduces to $A\mathbf{E} = \mathbf{R}$ and thus can be solved for the error $\mathbf{E}$ directly. In the nonlinear case, the residual equation $A(\mathbf{V}, \boldsymbol{\theta}) = A(\mathbf{U}, \boldsymbol{\theta}) + \mathbf{R}$ is solved for $\mathbf{V}$ on the coarse grid before the error can be extracted with $\mathbf{E} = \mathbf{V} - \mathbf{U}$.

For a given time-grid discretization $t_n = nh, n = 0, \ldots, N$, and $h = T/N$, the coarse grid is defined by choosing a coarsening factor $c > 1$ and assigning every cth time point to the next coarser time grid with $T_n = nh_\Delta, n = 0, \ldots, N_\Delta = N/c$, and coarse-grid spacing $h_\Delta = ch$. An example of two grid levels using a coarsening factor of $c = 5$ is given in Figure 1. The residual $\mathbf{R}$, as well as the current approximation $\mathbf{U}$ and controls $\boldsymbol{\theta}$, are restricted to the coarse grid with injection by choosing every cth time point; i.e., the restriction of $\mathbf{U}$ is

$$(3.4) \qquad \mathbf{U}_\Delta = (\mathbf{u}_\Delta^0, \mathbf{u}_\Delta^1, \ldots, \mathbf{u}_\Delta^{N_\Delta}), \quad \text{where } \mathbf{u}_\Delta^n = \mathbf{u}^{nc},$$

with $\mathbf{R}_\Delta, \boldsymbol{\theta}_\Delta$ defined analogously. Consequently, the residual equation that is to be solved on the coarse grid reads

$$(3.5) \qquad A_\Delta(\mathbf{V}_\Delta, \boldsymbol{\theta}_\Delta) = A_\Delta(\mathbf{U}_\Delta, \boldsymbol{\theta}_\Delta) + \mathbf{R}_\Delta.$$

Here, A_Δ denotes a rediscretization of A on the coarse grid utilizing a coarse-grid propagator Φ_Δ, i.e.,

$$(3.6) \qquad A_\Delta(\mathbf{U}_\Delta, \boldsymbol{\theta}_\Delta) := \begin{pmatrix} \mathbf{u}_\Delta^0 \\ \mathbf{u}_\Delta^1 - \Phi_\Delta(\mathbf{u}_\Delta^0, \boldsymbol{\theta}_\Delta^0) \\ \vdots \\ \mathbf{u}_\Delta^{N_\Delta} - \Phi_\Delta(\mathbf{u}_\Delta^{N_\Delta - 1}, \boldsymbol{\theta}_\Delta^{N_\Delta - 1}) \end{pmatrix}.$$

An obvious choice for Φ_Δ is a rediscretization of the problem on the coarse grid, such as by using the same propagator as that used on the fine grid, but with a bigger time step size $h_\Delta = ch$, thus skipping the fine-grid time points and updating only the coarse-grid points. For instance, Φ could be a forward or backward Euler discretization with time-step size h, and Φ_Δ could be a forward or backward Euler discretization with time step size $h_\Delta = ch$.[3] In

---

[3]In general, the time-grid hierarchy and the corresponding coarse-grid operator Φ_Δ should be chosen such that stability of the time-stepping method on each coarse time grid is ensured. In this work, the chosen hierarchy encountered no stability issues on coarse time grids for the chosen time-stepping method. In fact, we never observed stability issues, even on very coarse time grids. However, a more thorough treatment of stability for ResNets and MGRIT is beneficial and will be the topic of future research.
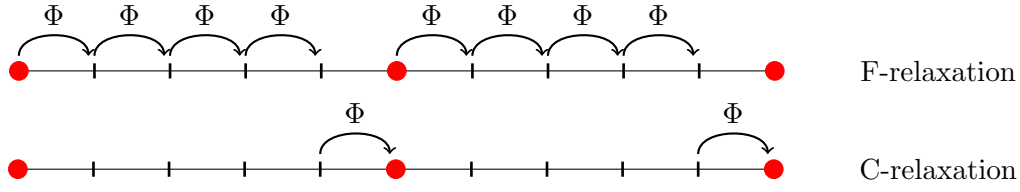
Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Figure 2. *F-relaxation and C-relaxation for a coarsening by a factor of $c = 5$.*

the case of forward Euler (i.e., ResNet architecture), the coarse-grid propagator Φ_Δ is given by

$$(3.7) \qquad \Phi_\Delta(\mathbf{u}_\Delta^n, \boldsymbol{\theta}_\Delta^n) = \mathbf{u}_\Delta^n + h_\Delta F(\mathbf{u}_\Delta^n, \boldsymbol{\theta}_\Delta^n).$$

On the coarse grid, residual equation (3.5) is solved exactly with forward substitution. Afterward, the error approximation on the coarse grid is extracted with $\mathbf{E}_\Delta = \mathbf{V}_\Delta - \mathbf{U}_\Delta$. This coarse-grid error approximation is then used to correct the fine-grid approximation $\mathbf{U}$ at coarse-grid points with $\mathbf{U}^{nc} \leftarrow \mathbf{U}^{nc} + \mathbf{E}_\Delta^n$.

Complementing the coarse time-grid error correction is the fine-grid relaxation process. Here, block Jacobi relaxation alternates between the fine-grid and the coarse-grid points. More precisely, relaxation on the fine points (called F-relaxation) corresponds to updating each fine point concurrently over each time chunk interval, thus propagating each coarse point value $\mathbf{u}^{kc}$ through the corresponding fine point interval (T^k, T^{k+1}) as in

$$(3.8) \qquad \mathbf{u}^n \leftarrow \Phi(\mathbf{u}^{n-1}, \boldsymbol{\theta}^{n-1}) \quad \text{for each } n = kc + 1, \ kc + 2, \ldots, (k+1)c - 1.$$

Importantly, each kth interval of fine points can be computed independently in parallel. Relaxation on the coarse points (called C-relaxation) is analogous and updates each coarse point concurrently by propagating the nearest left neighboring value. For the kth coarse point, the update is given by

$$(3.9) \qquad \mathbf{u}^{kc} \leftarrow \Phi(\mathbf{u}^{kc-1}, \boldsymbol{\theta}^{kc-1}) \quad \text{for } k = 1, 2, \ldots, N_\Delta.$$

The actions of F- and C-relaxations are described in Figure 2. Unless otherwise noted, we use FCF-relaxation, which is an application of F-relaxation (3.8), followed by an application of C-relaxation (3.9), and then F-relaxation (3.8) again. We note that such F/C orderings in relaxation are common for multigrid methods.

Taken together, the coarse-grid error correction and the fine-grid relaxation form the two-grid MGRIT cycle depicted in Algorithm 3.1.[4] Typically, the MGRIT Algorithm 3.1 is carried out recursively, with successively coarser time grids, until a coarsest time grid of trivial size is reached, and step 3 is solved exactly using forward substitution. If the levels are traversed in order, going down to the coarsest time grid and then back up to the finest time grid, this is called a V-cycle. It corresponds to the "Solve" in step 3 being implemented as a single

---

[4]The F-relaxation two-grid version of nonlinear MGRIT is equivalent to the Parareal algorithm [24].

recursive call. However, more powerful cycles can be applied that visit coarse time grids more frequently (such as F-cycles; see, e.g., [51, 12] for more information on multigrid cycling).

Note that the main work carried out on a given time grid is the parallel relaxation process. Thus the work on each MGRIT level is highly parallel. Only when a coarsest time grid of trivial size is reached is the level solved sequentially by forward substitution. Thus, the algorithm simultaneously computes all time steps in parallel, reducing the serial propagation component to the size of the coarsest grid plus the traversal through each level.

Algorithm 3.1. MGRIT(A, $\mathbf{U}$, $\boldsymbol{\theta}$, $\mathbf{G}$) for two grid levels.

1: Apply F- or FCF-relaxation to $A(\mathbf{U}, \boldsymbol{\theta}) = \mathbf{G}$ ▷ eq. (3.8)–(3.9)
2: Restrict the fine-grid approximation and residual $\mathbf{R}$ to the coarse grid:
    $\mathbf{U}_\Delta^i \leftarrow \mathbf{U}^{ic}, \mathbf{R}_\Delta^i \leftarrow (\mathbf{G} - A(\mathbf{U}, \boldsymbol{\theta}))^{ic}$ for $i = 0, \dots, N_\Delta$
3: Solve $A_\Delta(\mathbf{V}_\Delta, \boldsymbol{\theta}_\Delta) = A_\Delta(\mathbf{U}_\Delta, \boldsymbol{\theta}_\Delta) + \mathbf{R}_\Delta$.   ▷ eq. (3.5)
4: Compute the coarse-grid error approximation: $\mathbf{E}_\Delta = \mathbf{V}_\Delta - \mathbf{U}_\Delta$.
5: Correct $\mathbf{U}$:
    $\mathbf{U}^{ic} \leftarrow \mathbf{U}^{ic} + \mathbf{E}_\Delta^{ic}$ for $i = 0, \dots, N_\Delta$, and apply F-relaxation
6: **If** $\|\mathbf{R}\| \le tol$: halt.
    Else: go to step 1.

The MGRIT iterations can be mathematically considered a fixed-point method for solving the forward problem (2.14)–(2.15). Using the iteration index m, it reads

$$(3.10) \qquad \text{for } m = 0, 1, \dots : \quad \mathbf{U}_{m+1} = \text{MGRIT}(A, \mathbf{U}_m, \boldsymbol{\theta}, \mathbf{G}).$$

The MGRIT iterator has been shown to be a contraction in many settings for linear, nonlinear, parabolic, and hyperbolic problems, although hyperbolic problems tend to be more difficult (e.g., [17, 19, 33, 21]). Upon convergence, the limit fixed-point $\mathbf{U} = \text{MGRIT}(A, \mathbf{U}, \boldsymbol{\theta}, \mathbf{G})$ will satisfy the discrete network state equations as in (2.14)–(2.15), since MGRIT solves the same underlying problem.

Before starting the multigrid iterations, an initial solution guess for $\mathbf{U}$ must be set. Typically, the coarse-grid points are initialized using the best current solution estimate. This is often either some generic initial condition or an interpolated solution from a cheaper, coarser time grid.

3.2. Multigrid across layers for backpropagation. The same nonlinear multigrid scheme as described in section 3.1 can also be utilized to solve the adjoint equations (2.16)–(2.17) in layer-parallel. The adjoint equations are linear in the adjoint variables $\bar{\mathbf{u}}^n$, and those are propagated backward through the network. The adjoint space-time system thus reads

$$(3.11) \quad \underbrace{\begin{pmatrix} \mathbf{I} & & & & \\ -(\partial_\mathbf{u}\Phi^{N-1})^T & \mathbf{I} & & & \\ & -(\partial_\mathbf{u}\Phi^{N-2})^T & \mathbf{I} & & \\ & & \ddots & \ddots & \\ & & & -(\partial_\mathbf{u}\Phi^0)^T & \mathbf{I} \end{pmatrix}}_{=:A_\mathbf{U}(\bar{\mathbf{U}}, \boldsymbol{\theta})} \begin{pmatrix} \bar{\mathbf{u}}^N \\ \bar{\mathbf{u}}^{N-1} \\ \vdots \\ \bar{\mathbf{u}}^1 \\ \bar{\mathbf{u}}^0 \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{1}{s}(\partial_{\mathbf{u}^N}\ell^N)^T \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{=:\mathbf{G}_\mathbf{U}},$$

where again $\bar{\mathbf{u}}^n$ denotes the adjoint variable at layer n for a general example $\mathbf{y}$ or for a batch of examples $\mathbf{y}_k$, $k \in \mathcal{S} \subset \{1, \ldots, s\}$. Further, $(\partial_\mathbf{u}\Phi^n)^T$ denotes the partial derivative $\partial_\mathbf{u}\Phi(\mathbf{u}^n, \boldsymbol{\theta}^n)^T = \left(\frac{\partial\mathbf{u}^{n+1}}{\partial\mathbf{u}^n}\right)^T$. It corresponds to the backward layer propagation of adjoint sensitivities, which in the case of a forward Euler discretization for Φ (i.e., ResNet architecture) reads

$$(3.12) \qquad \partial_\mathbf{u}\Phi(\mathbf{u}^n, \boldsymbol{\theta}^n)^T\bar{\mathbf{u}}^{n+1} = \bar{\mathbf{u}}^{n+1} + h\partial_\mathbf{u}F(\mathbf{u}^n, \boldsymbol{\theta}^n)^T\bar{\mathbf{u}}^{n+1}.$$

Each backward propagator at layer n depends on the primal state $\mathbf{u}^n$, and hence the system matrix and right-hand side of (3.11) depend on the current state $\mathbf{U}$, which is reflected in the subscripts in $A_\mathbf{U}$ and $\mathbf{G}_\mathbf{U}$. The structure of the adjoint system (3.11), however, is the same as that of the state system (3.1). Hence the same MGRIT approach as presented in Algorithm 3.1 can be utilized to solve the adjoint equations with the layer-parallel multigrid scheme by applying the iteration

$$(3.13) \qquad \bar{\mathbf{U}}_{m+1} = \text{MGRIT}(A_\mathbf{U}, \bar{\mathbf{U}}_m, \boldsymbol{\theta}, \mathbf{G}_\mathbf{U})$$

for the adjoint vector $\bar{\mathbf{U}} := (\bar{\mathbf{u}}^N, \ldots, \bar{\mathbf{u}}^0)$.

Remark 3.1. The adjoint equations depend on the primal states $\mathbf{u}_k^n$. Therefore, those states need to be either stored during forward propagation or recomputed while solving the adjoint equations. Hybrid approaches, such as the check-pointing method, have been developed and compromise memory consumption with computational complexity (see, e.g., [52]). Memory-free methods using reversible networks were first proposed for general dynamics in [27]. However, as shown in [13], not all architectures that are algebraically reversible are numerically forward and backward stable. This motivates limiting the forward propagation to stable dynamics, e.g., dynamics inspired by hyperbolic systems.

3.3. Nonintrusive implementation. The MGRIT algorithm relies on the action of the layer-to-layer forward and backward propagators, Φ and $\partial_\mathbf{u}\Phi^T$, and their respective rediscretizations, Φ_Δ and $\partial_\mathbf{u}\Phi_\Delta^T$, on coarser grid levels. However, it does not access or "know" the internals of these functions. Hence, MGRIT can be applied in a fully nonintrusive way with respect to any existing discretization of the nonlinear dynamics describing the network forward and backward propagation. A user can wrap existing sequential evolution operators according to an MGRIT software interface, and then the MGRIT code iteratively computes the solution to (2.14)–(2.15) and (2.16)–(2.17) in parallel.

Our chosen MGRIT implementation for time-parallel computations (forward and backward) is XBraid [53]. One particular advantage of XBraid is its generic and flexible user-interface that requires relatively straightforward user-routines (which likely already exist), such as how to take inner-products and norms with vectors $\mathbf{u}^n$, how to take a time step with Φ and $\partial_\mathbf{u}\Phi^T$, etc.

Since the user defines the action of Φ, any existing implementation of layer computations can continue to be used, including accelerator code, e.g., for GPUs. However, since Φ takes a single time step, any use of GPU kernels for Φ implies memory movement to and from the CPU at every time step. This is because current architectures largely rely on the CPU to handle the message-passing layer of parallelism, and it is over this layer that XBraid provides

temporal parallelism. However, future implementations could move the message-passing layer to occur solely on the GPU, thus removing this memory movement overhead. Additionally, the bandwidth and latency between CPUs and accelerators will continue to improve, also ameliorating this issue.

Remark 3.2. The state and adjoint MGRIT iterations recover at convergence the same reduced gradient as that of a layer-serial forward and backward propagation through the network. Thus they can be integrated into any gradient-based training algorithm for updating the network control parameters $\boldsymbol{\theta}, \mathbf{W}, \boldsymbol{\mu}$. Subgradient methods, such as SGD or other batch approaches, can also be utilized by choosing the corresponding subset $\mathcal{S} \subset \{1, \ldots, s\}$. Regarding speedup and parallelism, the layer-parallel computations are particularly attractive in the small-batch mode when options for data parallelism are limited. Overall, we expect a runtime speedup through the greater concurrency within state and adjoint solves, if the computational resources are large enough.

4. Simultaneous layer-parallel training. The iterative nature of the layer-parallel multigrid scheme allows for a simultaneous training approach that solves the network state and adjoint equations inexactly during training. To this end, we reduce the accuracy of the state and adjoint MGRIT solver during training and update the network control parameters utilizing inexact gradient information. This corresponds to an early stopping of the MGRIT iterations in each outer optimization cycle. The theoretical background of this early-stopping approach of the inner state and adjoint fixed-point iterations is based on the *one-shot* method [9], which has been successful for reducing runtimes of many PDE-constrained optimization problems in aerodynamics applications (e.g., [41, 25, 8]).

Algorithm 4.1. Simultaneous layer-parallel training.

1: Perform m_1 state updates: ▷ section 3.1
    for $m = 1, \ldots, m_1:$ $\mathbf{U}_m \leftarrow \mathrm{MGRIT}(A, \mathbf{U}_{m-1}, \boldsymbol{\theta}, \mathbf{G})$
2: Perform m_2 adjoint updates: ▷ section 3.2
    for $m = 1, \ldots, m_2:$ $\bar{\mathbf{U}}_m \leftarrow \mathrm{MGRIT}(A_{\mathbf{U}_{m_1}}, \bar{\mathbf{U}}_{m-1}, \boldsymbol{\theta}, \mathbf{G}_{\mathbf{U}_{m_1}})$
3: Assemble reduced gradient $\nabla_{\boldsymbol{\theta}} J, \nabla_{\mathbf{W}} J, \nabla_{\boldsymbol{\mu}} J$ ▷ (2.18),(2.19)
4: Approximate Hessians $\mathbf{B}_{\boldsymbol{\theta}}, \mathbf{B}_{\mathbf{W}}, \mathbf{B}_{\boldsymbol{\mu}}$ and select a step size $\alpha > 0$
5: Network control parameter update:
    $\boldsymbol{\theta} \;\leftarrow \boldsymbol{\theta} \;\; - \alpha \mathbf{B}_{\boldsymbol{\theta}}^{-1} \nabla_{\boldsymbol{\theta}} J$
    $\mathbf{W} \leftarrow \mathbf{W} - \alpha \mathbf{B}_{\mathbf{W}}^{-1} \nabla_{\mathbf{W}} J$
    $\boldsymbol{\mu} \;\leftarrow \boldsymbol{\mu} \;\; - \alpha \mathbf{B}_{\boldsymbol{\mu}}^{-1} \nabla_{\boldsymbol{\mu}} J$
6: **If** converged: halt
    Else: go to step 1.

The simultaneous layer-parallel training approach is summarized in Algorithm 4.1. We clarify the details of the method in the following points:

- *Number of state and adjoint updates m_1, m_2:* For "large" m_1, m_2, the algorithm recovers the same gradient, and hence the same scheme, as that of a conventional layer-serial gradient-based training approach but with the addition of enabled layer-parallelism, providing runtime reductions through greater concurrency. Considering

smaller numbers of inner MGRIT iterations, e.g., $m_1, m_2 \in \{1, 2\}$, further reduces the runtime of each iteration and yields the simultaneous optimization approach. In that case, control parameter updates in step 5 are based on inexact gradient information utilizing the most recent state and adjoint variables $(\mathbf{u}_{m_1}^n)$ and $(\bar{\mathbf{u}}_{m_2}^n)$.

For the extreme case $m_1 = m_2 = 1$ (and appropriate Hessian approximation of quasi-Newton type; see below), the resulting optimization iteration can be mathematically interpreted as an approximate, reduced sequential quadratic programming (rSQP) method with convergence analysis as presented in [41]. In [10], theoretical considerations on the choice of m_1, m_2 are presented, which rely on the state and adjoint residuals by searching for descent on an augmented Lagrangian function. In practice, choosing m_1, m_2 to be as small as 2 has proven successful in our experience.

- *Hessian approximation*: In order to prove convergence of the simultaneous one-shot method on a theoretical level, the preconditioners $\mathbf{B}_{\boldsymbol{\theta}}, \mathbf{B}_{\mathbf{W}}, \mathbf{B}_{\boldsymbol{\mu}}$ should approximate the Hessian of an augmented Lagrangian function that involves the residual of the state and adjoint equations (see [9] and references therein). Numerically, we approximate the Hessian through consecutive limited-memory BFGS updates based on the current reduced gradient (thus assuming that the residual term is small). Alternatively, one might try to approximate the Hessian with a scaled identity matrix, which drastically reduces computational complexity and has already proven successful in various applications of the one-shot method. It should be noted that the Hessian with respect to $\mathbf{W}, \boldsymbol{\mu}$ can be computed directly, as it involves only the second derivative of the loss function ℓ in (2.5) and the regularizer R.

- *Step size selection*: The step size α is selected through a standard line-search procedure based on the current value of the objective function, e.g., a backtracking line-search satisfying the (strong) Wolfe condition (see, e.g., [49]).

- *Stopping criterion*: Since the one-shot method targets optimality and feasibility of the state, adjoint, and control variables simultaneously, the stopping criterion should involve not only the norm of the reduced gradient but also the norm of the state and adjoint residuals. In the context of network training, however, solving the optimization problem to high accuracy is typically not desired in order to prevent overfitting. We therefore compute a validation accuracy in each iteration of the above algorithm by applying the current network controls to a separate validation data set. We terminate the training, if the current network controls produce a high validation accuracy, rather than focusing on the current residuals of the state, adjoint, and gradient norms.

5. Numerical results. We investigate the computational benefits of the simultaneous layer-parallel training approach on three test cases. For all test cases, our focus is on the ability to achieve speedup in training runtimes for very deep neural networks by introducing parallelism between the layers. It is likely, though not explored here, that greater combined speedups are possible by additionally using data-parallelism or parallelizing inside of each layer. Further studies are required to better understand the trade-off of distributing parallel work between layer- and data-parallelism.
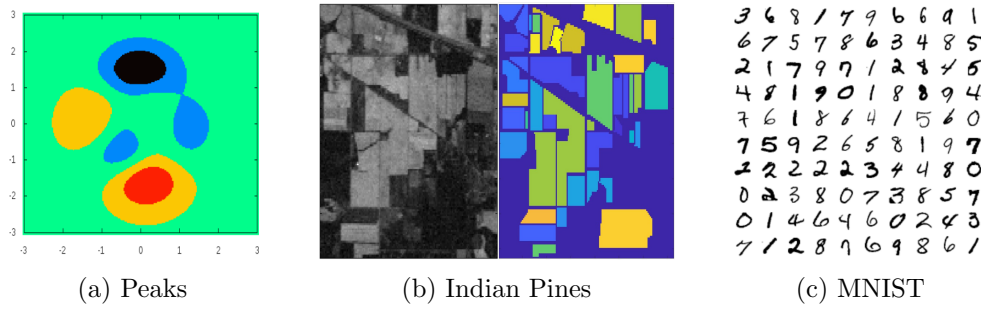
(a) Peaks   (b) Indian Pines   (c) MNIST

Figure 3. *Classes of the Peaks example (test case 1), sample band and true classes of the Indian Pines data set (test case 2), and examples from the MNIST data set (test case 3).*

5.1. Test cases.

1. Level set classification (*Peaks example*).

As a first step, we consider the test problem suggested in [34] for classifying grid points into five (nonconvex) level sets of a smooth nonlinear function $f : [-3, 3]^2 \to \mathbb{R}$ (Figure 3(a)). The training data set consists of $s = 5000$ randomly chosen points $\mathbf{y}_k \in [-3, 3]^2$, $k = 1, \ldots, s$, and standard basis vectors $c_k \in \mathbb{R}^5$ which represent the probability that a point $\mathbf{y}_k$ belongs to level set $i \in \{1, 2, 3, 4, 5\}$. The goal is to train a network to predict the correct level sets for new points in $[-3, 3]^2$ (validation points). We choose a ResNet architecture with smoothed ReLU activation defined as

$$(5.1) \qquad \sigma(x) = \begin{cases} \max\{x, 0\}, & |x| > 0.1, \\ 2\frac{1}{2}x^2 + \frac{1}{2}x + \frac{1}{40}, & |x| \le 0.1. \end{cases}$$

Also, we define the linear operations $\mathbf{K}(\cdot)$ at each layer to be a dense matrix representation of the weights $\boldsymbol{\theta}^n$. We choose a network depth of $T = 5$ discretized with up to $N = 2048$ layers and a network width of 8 such that $\mathbf{u}^n \in \mathbb{R}^8 \ \forall n = 0, \ldots, N$. In order to map the data set to the network width, we choose L_{in} to be a dense $\mathbb{R}^{8 \times 2}$ matrix, whose entries are learned alongside the network parameters, followed by an initial application of the activation function.

2. Hyperspectral image segmentation (*Indian Pines*).

In this test case, we consider a soil segmentation problem based on a hyperspectral image data set. The input data consists of hyperspectral bands over a single landscape in Indiana, USA (Indian Pines data set [3]) with 145×145 pixels. For each pixel, the data set contains 220 spectral reflectance bands which represent different portions of the electromagnetic spectrum in the wavelength range $0.4 - 2.5 \cdot 10^{-6}$. The goal is to train a network to assign each pixel of the scene to one of 16 class labels that represent the type of land-cover present at that pixel (such as alfalfa, corn, soybean, wheat, etc.); see Figure 3(b).

We use the spectral bands of $s = 1000$ randomly chosen pixel points $\mathbf{y}_k \in \mathbb{R}^{220}$, $k = 1, \ldots, s$, together with their corresponding class probability vectors $\mathbf{c}_k \in \mathbb{R}^{16}$ (unit vectors) for training. The network architecture is a ResNet with smoothed ReLU activation (i.e., $\sigma(x) = \max\{0, x\}$, smoothed around zero) and defines the linear

operations $\mathbf{K}(\cdot)$ at each layer to be a dense matrix representation of the weights $\boldsymbol{\theta}^n$. We choose a network depth of $T = 20$ discretized with up to $N = 2048$ layers and a network width of 220 channels, corresponding to the 220 reflectance bands. The initial operator L_{in} is chosen to be the identity.

3. MNIST image classification (*MNIST*).

As a final example, we consider the now classic MNIST [47] test case for classification of handwritten digits encoded in a 28×28 grayscale image (Figure 3(c)). Our objective for this test case is to demonstrate the scalability of the layer-parallel approach over an increasing number of layers. While we obtain reasonable validation accuracy, the objective is not to develop an optimal ResNet to solve this problem. Further, we obtained the timings below with our own straightforward implementation of convolutions, ensuring compatible layer-to-layer propagators with XBraid for our initial tests. Future work will use a fast convolution library, which will provide a substantial speedup to both the serial and layer-parallel codes.

For the weak scaling runs below, we use a ResNet architecture with tanh activation and define internal layers by the linear operator $\mathbf{K}(\cdot)$ using 8 convolution kernels of width 3; we used similar architectures in [35, 34]. This yields a weight tensor of size $\mathbb{R}^{3\times3\times8\times8}$ at each layer. The parameters to be trained are in $\mathbb{R}^{28\times28}$ at each layer. The strong scaling training tests below used four convolutional kernels to reduce memory requirements. The network is defined to have a depth of $T = 5$ and is discretized with up to $N = 2048$ layers. The initial operator L_{in} is chosen to be the identity copied over the eight (or four) convolutional kernels.

The Peaks and Indian Pines computations were performed on the RHRK cluster Elwetritsch II at TU Kaiserslautern. Elwetritsch II has 485 nodes based on Haswell (2×8 cores, 64GB) and Skylake (2×12 cores, 96GB) architectures. The computations for the MNIST results were performed on the Skybridge capacity cluster at Sandia National Laboratories. Skybridge is a Cray cluster containing 1848 nodes with two 8-core Intel 2.6 GHz Sandy Bridge processors, 64GB of RAM per node, and an Infiniband interconnect. The source code is available online [31].

5.2. Layer-parallel scaling and performance validation. First, we investigate the performance of the layer-parallel MGRIT propagation for a single objective function and gradient evaluation. Here, we keep the network weights fixed and propagate a batch of examples of sizes $s = 5000, 1000, 500$ for the Peaks, Indian Pines, and MNIST test cases, respectively, through the network. We choose a coarsening factor of $c = 4$ to set up a hierarchy of ever coarser layer-grids to employ the multigrid scheme. This coarsening strategy did not encounter any stability issues for forward Euler on the coarser layer-grids.

Figure 4 shows the convergence history of the MGRIT iterations for two different problem sizes using $N = 256$ and $N = 2048$ layers. We monitor the relative drop of the state and adjoint residual norms and observe fast convergence for all test cases that is independent of the number of layers. Note that the performed multigrid iterations themselves are not dependent on the number of cores used for parallelization, making Figure 4 independent of the parallel distribution. Next, we report scaling results varying the number of cores.

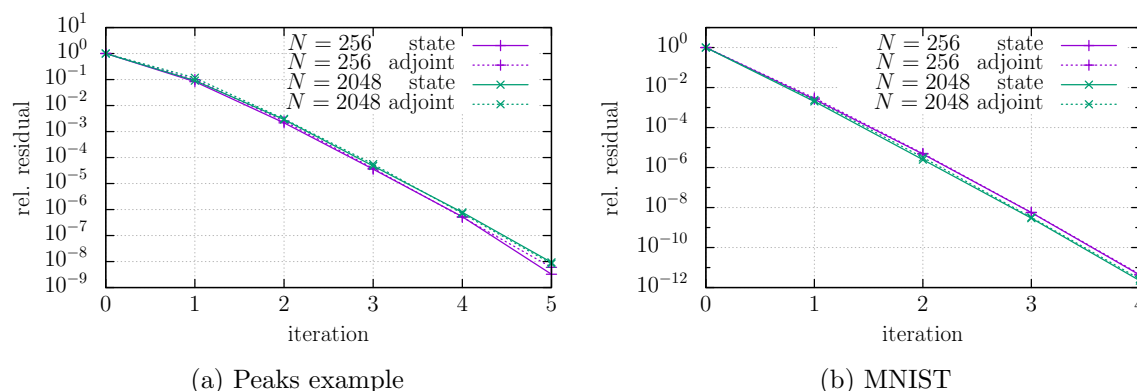We investigate scaling results for the layer-parallel MGRIT scheme and compare runtimes

(a) Peaks example   (b) MNIST

Figure 4. *Convergence history of MGRIT solving the state and adjoint equations for $N = 256$ and $N = 2048$ layers. The MGRIT scheme achieves fast convergence independent of the number of layers. (The corresponding figure for the Indian Pines test case shows the same quantitative behavior and hence is omitted here.)*

to conventional serial-in-layer forward and backward propagation. Figure 5 presents a weak-scaling study for the layer-parallel MGRIT scheme. Here, we double the number of layers as well as the number of compute cores while keeping the ratio $N/\#\text{cores} = 4$ fixed, such that each compute unit processes 4 layers. Runtimes are measured for one objective function and gradient evaluation, using a relative stopping criterion of five orders of magnitude for the MGRIT residual norms. Note that the layer-serial data points have been added for comparison, even though they are executed on only one core. For the layer-serial propagation, doubling the number of layers leads to a doubling in runtime. The layer-parallel MGRIT approach, however, yields nearly constant runtimes independent of the problem size. The resulting speedups are reported in Table 1. Since the layer-parallel MGRIT approach removes the linear runtime scale of the conventional serial-layer propagation, resulting speedups increase linearly, with the problem size yielding up to a factor of $16\times$ for the MNIST case using 2048 layers and 512 cores. Further speedup can be expected when considering ever more layers (and computational resources).

A strong scaling study is presented in Figure 6 for various numbers of layers. Here, we keep the problem sizes fixed and measure the time-to-solution for one gradient evaluation, with MGRIT for increasing numbers of computational resources. It shows good, strong scaling behavior for all test cases, independent of the numbers of layers. The cross-over point where the layer-parallel MGRIT approach shows speedup over the layer-serial propagation is around 16 cores for all cases.

5.3. Simultaneous layer-parallel training validation. Next, we investigate the simultaneous layer-parallel training, using $m_1 = m_2 = 2$ layer-parallel MGRIT iterations in each outer training iteration (see Algorithm 4.1). The Hessian approximations $B_{\boldsymbol{\theta}}, B_{\mathbf{W}}, B_{\boldsymbol{\mu}}$ are computed by successive limited-memory BFGS updates based on the current gradient $\nabla_{(\boldsymbol{\theta},\mathbf{W},\boldsymbol{\mu})}J$. We compare runtimes of the simultaneous layer-parallel training with a conventional layer-serial training approach, while choosing the same Hessian as well as the same initial network parameters for both approaches. However, we tune the optimization hyperparameters (such as regularization parameters, step size selection, etc.) separately for both schemes in order to
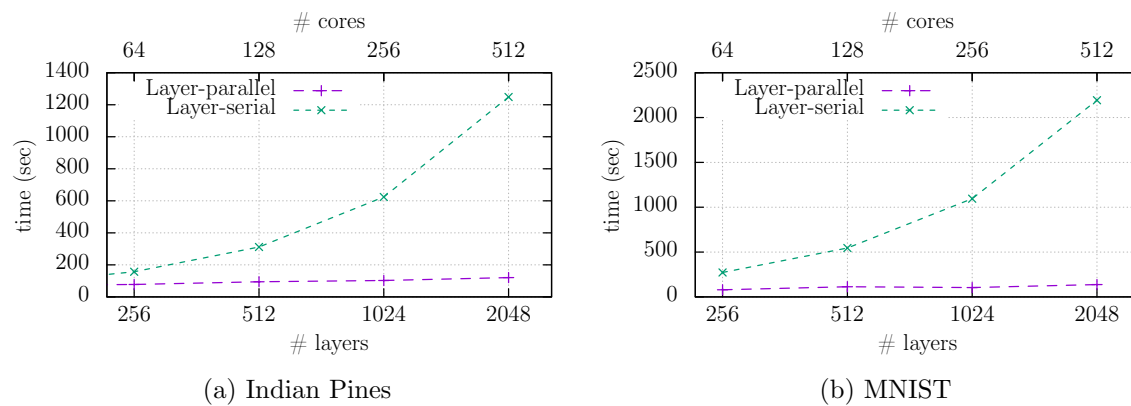
(a) Indian Pines   (b) MNIST

Figure 5. *Runtime comparison of a layer-parallel gradient evaluation with layer-serial forward and backward propagation. The layer-parallel approach yields nearly constant runtimes for increasing problem sizes and computational resources. (The corresponding figure for the Peaks test case shows the same quantitative behavior and hence is omitted here.)*

Table 1
Runtime and speedup of layer-parallel gradient evaluation over layer-serial propagation.

Test case	#Layers	#Cores	Serial	Parallel	Speedup
Peaks	256	64	1.8 sec	1.2 sec	1.5
	512	128	3.7 sec	1.5 sec	2.5
	1024	256	7.1 sec	1.6 sec	4.3
	2048	512	13.9 sec	1.8 sec	7.7
Indian Pines	256	64	157.1 sec	77.6 sec	2.0
	512	128	311.6 sec	94.5 sec	3.3
	1024	256	624.0 sec	102.6 sec	6.1
	2048	512	1248.0 sec	120.6 sec	10.3
MNIST	256	64	272.3 sec	79.5 sec	3.4
	512	128	545.3 sec	113.3 sec	4.8
	1024	256	1095.2 sec	104.0 sec	10.5
	2048	512	2193.5 sec	137.3 sec	16.0

find the best setting that reaches a prescribed validation accuracy with the least iterations and minimum runtime.

For the Peaks example, we train a network with $N = 1024$ layers distributed onto 256 compute cores, and for the Indian Pines data set and the MNIST case we choose $N = 512$ layers distributed onto 128 compute cores, giving 4 layers per processor in all cases. Figure 7 plots the training history over iteration counts (top) as well as runtime (bottom). We validate from the top figures that both approaches reach comparable performance in terms of training results (optimization iteration counts, training loss, and validation accuracy). Hence, reducing the accuracy of the inner multigrid iterations for solving the state and adjoint equations within a simultaneous training framework does not deteriorate the training behavior. But each iteration of the simultaneous layer-parallel approach is much faster than for the layer-serial approach due to the layer-parallelization and the reduced state and adjoint accuracies.
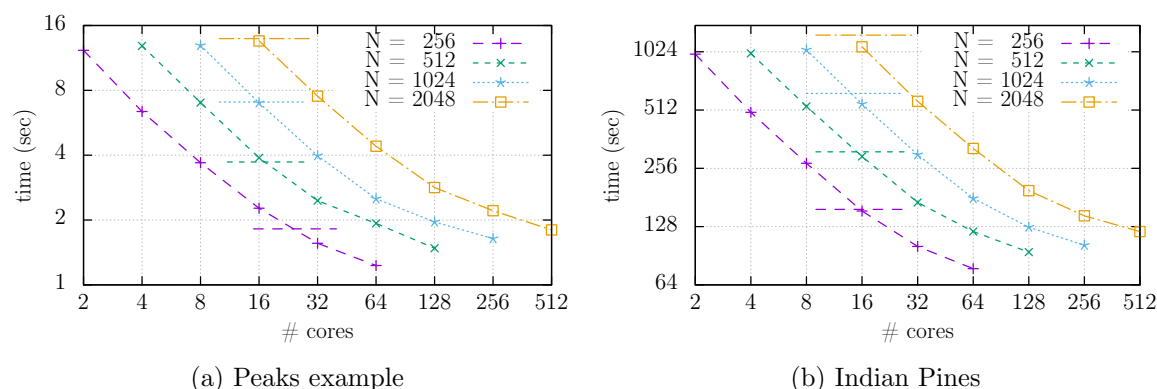
(a) Peaks example   (b) Indian Pines

Figure 6. *Strong scaling study for a layer-parallel gradient evaluation for various problem sizes from $N = 256$ to $N = 2048$ layers. Corresponding serial runtimes are indicated by horizontal dashed lines. The cross-over point where the layer-parallel approach yields speedup over the layer-serial propagation lies around 16 cores. (The corresponding figure for the MNIST test case shows the same quantitative behavior, and has hence been omitted here.)*

Therefore, the overall runtime for reaching the same final training result is reduced drastically (bottom figures). Runtime speedups are reported in Table 2. While these results have been computed for selected fixed N, we expect the speedup to scale linearly with increasing numbers of layers, similar to the observation in Table 1.

6. Conclusion. In this paper, we provide a proof-of-concept for layer-parallel training of deep residual neural networks (ResNets). The similarity between training ResNets and optimal control of nonlinear time-dependent differential equations motivates us to use parallel-in-time methods that are popular in many engineering applications. The method developed is based on nonlinear multigrid methods and introduces a new form of parallelism across layers.

We demonstrate two options to benefit from the layer-parallel approach. First, the nonlinear multigrid reduction in time (MGRIT) method can be used to replace forward and backward propagations in existing training algorithms, including for stochastic approximation methods such as stochastic gradient descent (SGD). In our experiments, this leads to speedup over serial implementations when using more than 16 compute cores. Second, additional savings can be obtained through the simultaneous layer-parallel training, which uses only inexact forward and backward propagations.

While the reported speedups might seem small in terms of parallel efficiency, these reductions can be of significant importance when considering large overall training runtimes. When bare training runtimes are in the order of days, any runtime reduction is appreciated, as long as computational resources are available. Further, since training a network typically involves a careful choice of hyperparameters, faster training runtimes will enable faster hyperparameter optimization and thus eventually lead to better training results in general. Lastly, we mention that such efficiencies for multigrid-in-time methods are not uncommon [20], where the nonintrusiveness of MGRIT contributes to the seemingly low efficiency, as does the fact that we are defining the efficiency of MGRIT with respect to an optimal serial algorithm. If the efficiency were defined with respect to MGRIT using one core, then the efficiencies would be much higher.
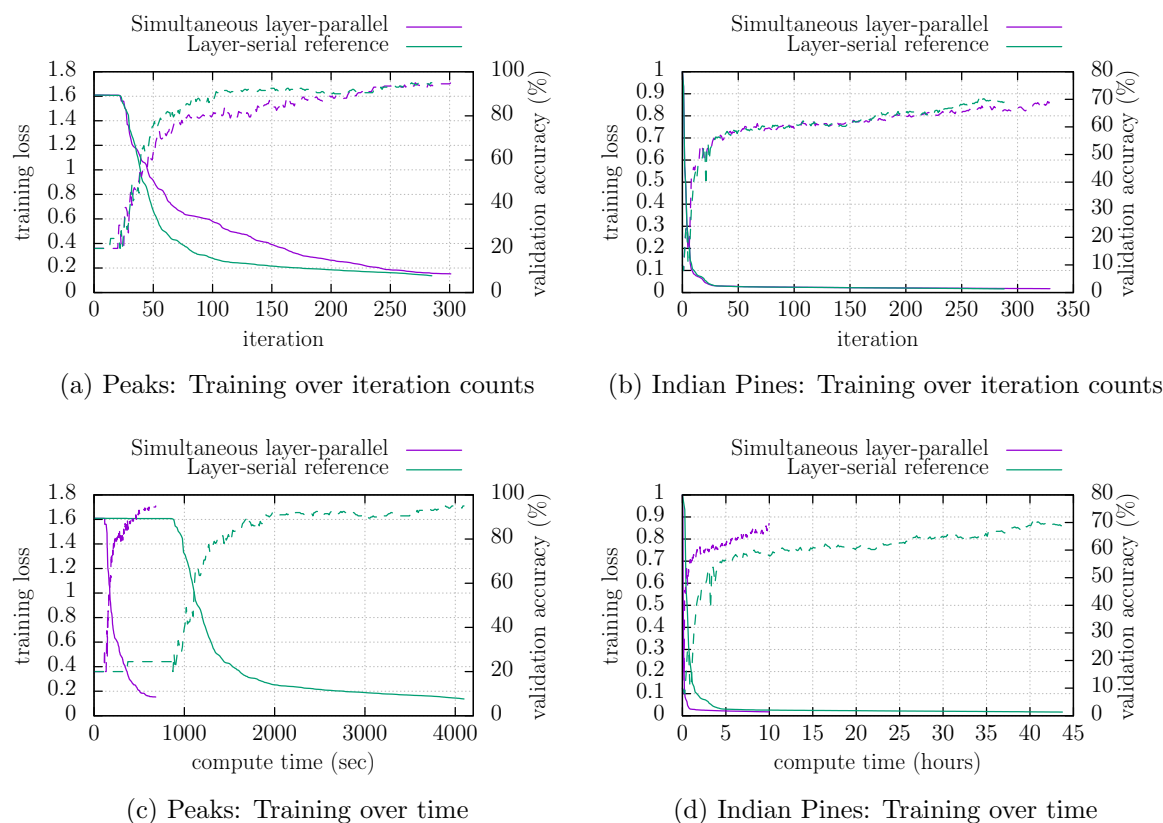
(a) Peaks: Training over iteration counts

(b) Indian Pines: Training over iteration counts

(c) Peaks: Training over time

(d) Indian Pines: Training over time

Figure 7. *Training loss (solid lines) and validation accuracy (dashed lines) over training iterations (top) and compute time (bottom). For the layer-parallel training, each core processes four layers. The simultaneous layer-parallel approach reaches training results comparable to a layer-serial approach within much less computational time. (The corresponding figure for the MNIST test case shows the same quantitative behavior, and has hence been omitted here.)*

Table 2
Runtime speedup of simultaneous layer-parallel training over layer-serial training.

Test case	N	#Cores	Layer-serial	Layer-parallel	Speedup
Peaks example	1024	256	4096 sec	683 sec	6.0
Indian Pines	512	128	2623 min	597 min	4.4
MNIST	512	128	619 min	71 min	8.5

Motivated by these first promising results, we will investigate the use of layer-parallel training for more challenging learning tasks, including more complex image-recognition problems. Further reducing the memory footprint of our algorithm in those applications motivates the use of reversible networks arising from hyperbolic systems [13]. A challenge arising here is the interplay of MGRIT and hyperbolic systems. Lastly, we note that while the current work focused on algorithmic development, it could nonetheless benefit greatly from integration with a more optimized code such as TensorFlow or Chainer. This is planned for future work.

An interesting topic concerning the interplay of existing codes with MGRIT is to develop strategies that handle more complicated layer architectures, such as pooling and connector layers. Here, we plan on leveraging previous work on adaptive spatial coarsening/refinement in MGRIT where the problem size at each time step can change.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., *Tensorflow: A system for large-scale machine learning*, in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, 2016, pp. 265–283.

[2] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data: A Short Course*, AML-Book, New York, NY, 2012.

[3] M. F. Baumgardner, L. L. Biehl, and D. A. Landgrebe, 220 *Band AVIRIS Hyperspectral Image Data Set: June* 12*, 1992, Indian Pine Test Site* 3, Purdue University Research Repository, 2015, https://doi.org/10.4231/R7RX991C.

[4] Y. Bengio, *Learning deep architectures for AI*, Found. Trends Mach. Learn., 2 (2009), pp. 1–127.

[5] G. Biros and O. Ghattas, *Parallel Lagrange–Newton–Krylov–Schur methods for PDE-constrained optimization: Part* I: *The Krylov–Schur solver*, SIAM J. Sci. Comput., 27 (2005), pp. 687–713, https://doi.org/10.1137/S106482750241565X.

[6] A. Bordes, S. Chopra, and J. Weston, *Question Answering with Subgraph Embeddings*, preprint, https://arxiv.org/abs/1406.3676v3, 2014.

[7] A. Borzì and V. Schulz, *Computational Optimization of Systems Governed by Partial Differential Equations*, Comput. Sci. Eng. 8, SIAM, 2012, https://doi.org/10.1137/1.9781611972054.

[8] T. Bosse, N. Gauger, A. Griewank, S. Günther, L. Kaland, C. Kratzenstein, L. Lehmann, A. Nemili, E. Özkaya, and T. Slawig, *Optimal design with bounded retardation for problems with non-separable adjoints*, in Trends in PDE Constrained Optimization, Internat. Ser. Numer. Math. 165, Birkhäuser/Springer, 2014, pp. 67–84.

[9] T. Bosse, N. Gauger, A. Griewank, S. Günther, and V. Schulz, *One-shot approaches to design optimization*, in PDE Constrained Optimization, Internat. Ser. Numer. Math. 165, Birkhäuser/Springer, 2014, pp. 43–66.

[10] T. Bosse, L. Lehmann, and A. Griewank, *Adaptive sequencing of primal, dual, and design steps in simulation based optimization*, Comput. Optim. Appl., 57 (2014), pp. 731–760.

[11] A. Brandt, *Multi-level adaptive solutions to boundary-value problems*, Math. Comp., 31 (1977), pp. 333–390.

[12] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed., SIAM, 2000, https://doi.org/10.1137/1.9780898719505.

[13] B. Chang, L. Meng, E. Haber, L. Ruthotto, D. Begert, and E. Holtham, *Reversible architectures for arbitrarily deep residual neural networks*, in Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), New Orleans, LA, 2018, pp. 2811–2818.

[14] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural Ordinary Differential Equations*, preprint, https://arxiv.org/abs/1806.07366, 2018.

[15] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, *Natural language processing (almost) from scratch*, J. Mach. Learn. Res., 12 (2011), pp. 2493–2537, http://www.jmlr.org/papers/v12/collobert11a.html.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, *Large scale distributed deep networks*, in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., Curran Associates, Inc., 2012, pp. 1223–1231.

[17] V. A. Dobrev, Tz. Kolev, N. A. Petersson, and J. B. Schroder, *Two-level convergence theory for multigrid reduction in time (MGRIT)*, SIAM J. Sci. Comput., 39 (2017), pp. S501–S527, https://doi.org/10.1137/16M1074096.

[18] W. E, *A proposal on machine learning via dynamical systems*, Comm. Math. Statist., 5 (2017), pp. 1–11, https://doi.org/10.1007/s40304-017-0103-z.

[19] R. D. FALGOUT, S. FRIEDHOFF, TZ. V. KOLEV, S. P. MACLACHLAN, AND J. B. SCHRODER, *Parallel time integration with multigrid*, SIAM J. Sci. Comput., 36 (2014), pp. C635–C661, https://doi.org/10.1137/130944230.

[20] R. D. FALGOUT, S. FRIEDHOFF, T. V. KOLEV, S. P. MACLACHLAN, J. B. SCHRODER, AND S. VANDE-WALLE, *Multigrid methods with space–time concurrency*, Comput. Vis. Sci., 18 (2017), pp. 123–143.

[21] R. D. FALGOUT, T. A. MANTEUFFEL, B. O'NEILL, AND J. B. SCHRODER, *Multigrid reduction in time for nonlinear parabolic problems: A case study*, SIAM J. Sci. Comput., 39 (2017), pp. S298–S322, https://doi.org/10.1137/16M1082330.

[22] M. J. GANDER, 50 *years of time parallel time integration*, in Multiple Shooting and Time Domain Decomposition, T. Carraro, M. Geiger, S. Körkel, and R. Rannacher, eds., Springer, 2015, pp. 69–114.

[23] M. J. GANDER, S. GÜTTEL, AND M. PETCU, *A nonlinear ParaExp algorithm*, in Domain Decomposition Methods in Science and Engineering XXIV, P. E. Bjørstad, S. C. Brenner, L. Halpern, H. H. Kim, R. Kornhuber, T. Rahman, and O. B. Widlund, eds., Springer International Publishing, 2018, pp. 261–270.

[24] M. J. GANDER AND S. VANDEWALLE, *Analysis of the parareal time-parallel time-integration method*, SIAM J. Sci. Comput., 29 (2007), pp. 556–578, https://doi.org/10.1137/05064607X.

[25] E. ÖZKAYA AND N. GAUGER, *Single-step one-shot aerodynamic shape optimization*, in Optimal Control of Coupled Systems of Partial Differential Equations, Internat. Ser. Numer. Math. 158, Birkhäuser Basel, 2009, pp. 191–204.

[26] M. B. GILES AND N. A. PIERCE, *An introduction to the adjoint approach to design*, Flow, Turbulence and Combustion, 65 (2000), pp. 393–415.

[27] A. N. GOMEZ, M. REN, R. URTASUN, AND R. B. GROSSE, *The reversible residual network: Backpropagation without storing activations*, in Advances in Neural Information Processing Systems 30, I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds., Curran Associates, Inc., 2017, pp. 2211–2221.

[28] I. GOODFELLOW, Y. BENGIO, AND A. COURVILLE, *Deep Learning*, MIT Press, 2016.

[29] S. GÖTSCHEL AND M. L. MINION, *An Efficient Parallel-in-Time Method for Optimization with Parabolic PDEs*, preprint, http://arxiv.org/abs/1901.06850, 2019.

[30] A. GRIEWANK AND A. WALTHER, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2008.

[31] S. GÜNTHER ET AL., *Layer-Parallel Deep Residual Learning Package*, https://github.com/steffi7574/LayerParallelLearning.

[32] S. GÜNTHER, N. R. GAUGER, AND J. B. SCHRODER, *A non-intrusive parallel-in-time adjoint solver with the XBraid library*, Comput. Vis. Sci., 19 (2018), pp. 85–95.

[33] S. GÜNTHER, N. R. GAUGER, AND J. B. SCHRODER, *A non-intrusive parallel-in-time approach for simultaneous optimization with unsteady PDEs*, Optim. Methods Software, 34 (2019), pp. 1306–1321.

[34] E. HABER AND L. RUTHOTTO, *Stable architectures for deep neural networks*, Inverse Probl., 34 (2017), 014004, https://doi.org/10.1088/1361-6420/aa9a90.

[35] E. HABER, L. RUTHOTTO, E. HOLTHAM, AND S.-H. JUN, *Learning across scales—A multiscale method for convolution neural networks*, in Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), New Orleans, LA, 2018, pp. 3142–3148, https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16580.

[36] A. HARLAP, D. NARAYANAN, A. PHANISHAYEE, V. SESHADRI, N. DEVANUR, G. GANGER, AND P. GIBBONS, *Pipedream: Fast and Efficient Pipeline Parallel DNN Training*, preprint, https://arxiv.org/abs/1806.03377, 2018.

[37] K. HE, X. ZHANG, S. REN, AND J. SUN, *Deep residual learning for image recognition*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, 2016, pp. 770–778.

[38] K. HE, X. ZHANG, S. REN, AND J. SUN, *Identity mappings in deep residual networks*, in Computer Vision—ECCV 2016, Lecture Notes in Comput. Sci. 9908, B. Leibe, J. Matas, N. Sebe, M. Welling, eds., Springer, 2016, pp. 630–645.

[39] G. HINTON, L. DENG, D. YU, G. E. DAHL, A.-R. MOHAMED, N. JAITLY, A. SENIOR, V. VANHOUCKE, P. NGUYEN, T. N. SAINATH, ET AL., *Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups*, IEEE Signal Process. Mag., 29 (2012), pp. 82–97.

[40] F. N. IANDOLA, M. W. MOSKEWICZ, K. ASHRAF, AND K. KEUTZER, *Firecaffe: Near-linear acceleration of deep neural network training on compute clusters*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, 2016, pp. 2592–2600.

[41] K. ITO, K. KUNISCH, V. SCHULZ, AND I. GHERMAN, *Approximate nullspace iterations for KKT systems*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 1835–1847, https://doi.org/10.1137/080724952.

[42] S. JEAN, K. CHO, R. MEMISEVIC, AND Y. BENGIO, *On Using Very Large Target Vocabulary for Neural Machine Translation*, preprint, https://arxiv.org/abs/1412.2007v2, 2014.

[43] J. KEUPER AND F.-J. PREUNDT, *Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability*, in Proceedings of the 2nd Workshop on Machine Learning in High Performance Computing Environments, IEEE Press, 2016, pp. 19–26.

[44] A. KRIZHEVSKY, I. SUTSKEVER, AND G. HINTON, *Imagenet classification with deep convolutional neural networks*, in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[45] Y. LECUN, Y. BENGIO, AND G. HINTON, *Deep learning*, Nature, 521 (2015), pp. 436–444.

[46] Y. LECUN, B. E. BOSER, AND J. S. DENKER, D. HENDERSON, R. E. HOWARD, W. E. HUBBARD, AND L. D. JACKEL, *Handwritten digit recognition with a back-propagation network*, in Advances in Neural Information Processing Systems 2, Morgan-Kaufmann 1990, pp. 396–404.

[47] Y. LECUN, L. BOTTOU, Y. BENGIO, AND P. HAFFNER, *Gradient-based learning applied to document recognition*, Proc. IEEE, 86 (1998), pp. 2278–2324.

[48] Y. LU, A. ZHONG, Q. LI, AND B. DONG, *Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations*, preprint, https://arxiv.org/abs/1710.10121, 2017.

[49] J. NOCEDAL AND S. WRIGHT, *Numerical Optimization*, 2nd ed., Springer Science+Business Media, 2006.

[50] J. B. SCHRODER, *Parallelizing over Artificial Neural Network Training Runs with Multigrid*, Tech. report LLNL-JRNL-736173, Lawrence Livermore National Laboratory; preprint, https://arxiv.org/abs/1708.02276, 2017.

[51] U. TROTTENBERG, C. OOSTERLEE, AND A. SCHÜLLER, *Multigrid*, Academic Press, 2001.

[52] Q. WANG, P. MOIN, AND G. IACCARINO, *Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation*, SIAM J. Sci. Comput., 31 (2009), pp. 2549–2567, https://doi.org/10.1137/080727890.

[53] *XBraid: Parallel Time Integration with Multigrid*, http://llnl.gov/casc/xbraid.

[54] J. C. ZIEMS AND S. ULBRICH, *Adaptive multilevel inexact SQP methods for PDE-constrained optimization*, SIAM J. Optim., 21 (2011), pp. 1–40, https://doi.org/10.1137/080743160.