SIAM J. SCI. COMPUT.
Vol. 44, No. 4, pp. A2322–A2348

SLIMTRAIN—A STOCHASTIC APPROXIMATION METHOD FOR TRAINING SEPARABLE DEEP NEURAL NETWORKS*

ELIZABETH NEWMAN†, JULIANNE CHUNG‡, MATTHIAS CHUNG‡, AND LARS RUTHOTTO§

Abstract. Deep neural networks (DNNs) have shown their success as high-dimensional function approximators in many applications; however, training DNNs can be challenging in general. DNN training is commonly phrased as a stochastic optimization problem whose challenges include nonconvexity, nonsmoothness, insufficient regularization, and complicated data distributions. Hence, the performance of DNNs on a given task depends crucially on tuning hyperparameters, especially learning rates and regularization parameters. In the absence of theoretical guidelines or prior experience on similar tasks, this requires solving a series of repeated training problems which can be time-consuming and demanding on computational resources. This can limit the applicability of DNNs to problems with nonstandard, complex, and scarce datasets, e.g., those arising in many scientific applications. To remedy the challenges of DNN training, we propose `slimTrain`, a stochastic optimization method for training DNNs with reduced sensitivity to the choice of hyperparameters and fast initial convergence. The central idea of slimTrain is to exploit the separability inherent in many DNN architectures; that is, we separate the DNN into a nonlinear feature extractor followed by a linear model. This separability allows us to leverage recent advances made for solving large-scale, linear, ill-posed inverse problems. Crucially, for the linear weights, `slimTrain` does not require a learning rate and automatically adapts the regularization parameter. In our numerical experiments using function approximation tasks arising in surrogate modeling and dimensionality reduction, slimTrain outperforms existing DNN training methods with the recommended hyperparameter settings and reduces the sensitivity of DNN training to the remaining hyperparameters. Since our method operates on mini-batches, its computational overhead per iteration is modest and savings can be realized by reducing the number of iterations (due to quicker initial convergence) or the number of training problems that need to be solved to identify effective hyperparameters.

Key words. deep learning, iterative methods, stochastic approximation, learning rates, variable projection, inverse problems

MSC codes. 68T07, 65K99, 65C20

DOI. 10.1137/21M1452512

1. Introduction.

Deep neural networks (DNNs) provide a powerful framework for approximating complex mappings, possessing universal approximation properties [15], and flexible architectures composed of simple functions parameterized by weights. Numerous studies have shown that excellent performance can be obtained using state-of-the-art DNNs in numerous applications including image processing,

*Submitted to the journal's Methods and Algorithms for Scientific Computing section October 14, 2021; accepted for publication (in revised form) April 4, 2022; published electronically August 4, 2022.

https://doi.org/10.1137/21M1452512

Funding: This work was partially supported by the NSF through grants DMS-1654175, DMS-1723005, and DMS-1751636. This work was also supported by the Air Force Office of Scientific Research through grant 20RT0237, and by the U.S. Department of Energy's Office of Advanced Scientific Computing Research Field Proposal 20-023231.

†Department of Mathematics, Emory University, Atlanta, GA 30322 USA (elizabeth.newman@emory.edu, http://math.emory.edu/~enewma5).

‡Department of Mathematics, Virginia Tech, Blacksburg, VA 24061 USA; Current address: Department of Mathematics, Emory University, Atlanta, GA 30322 USA (jmchung@emory.edu, http://www.math.emory.edu/~jmchung/, matthias.chung@emory.edu, http://www.math.emory.edu/~mchun45/).

§Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322 USA (lruthotto@emory.edu, http://www.mathcs.emory.edu/~lruthot).

speech recognition, surrogate modeling, and dimensionality reduction [23, 45, 49]. However, getting such results *in practice* may be a computationally expensive and cumbersome task. The process of training DNNs, or finding the optimal weights, is rife with challenges, e.g., the optimization problem is nonconvex, expressive networks require a very large number of weights, and, perhaps most critically, appropriate regularization is needed to ensure the trained network generalizes well to unseen data. Due to these challenges, it can be difficult to train a network efficiently and to sufficient accuracy. Training becomes even more difficult for large, high-dimensional datasets and complex mappings and in the absence of experience on similar learning tasks. The latter is a particular challenge in scientific applications that often involve unique training data sets, which limits the use of standard architectures and established hyperparameters.

While the literature on effective solvers for training DNNs is vast (see, e.g., the recent survey [7]), the most popular approaches are stochastic approximation (SA) methods. SA methods are computationally appealing since only a small, randomly chosen sample (i.e., mini-batch) from the training data is needed at each iteration to update the DNN parameters. Also, SA methods tend to exhibit good generalization properties. The most extensively studied and utilized SA method is the stochastic gradient descent (SGD) method [43] and its many popular variants such as AdaGrad [18] and ADAM [29]. Despite the popularity of SGD variants, major disadvantages include slow convergence and, most notoriously, the need to select a suitable learning rate (step size). Stochastic Newton and stochastic quasi-Newton methods have been proposed to accelerate convergence of SA methods [6, 24, 9, 52, 12], but including curvature information in SA methods is not trivial. Contrary to deterministic methods, which are known to benefit from the use of second-order information (consider, e.g., the natural step size of one and local quadratic convergence of Newton's method), noisy curvature estimates in stochastic methods may have harmful effects on the robustness of the iterations [12]. Furthermore, SA methods cannot achieve a convergence rate that is faster than sublinear [1], and additional care must be taken to handle nonlinear, nonconvex problems arising in DNN training. The performance and convergence properties of SA methods depend heavily on the properties of the objective function and on the choice of the learning rate.

In this paper, we seek to simplify the training of DNNs by exploiting the *separability* inherent in most common DNN architectures. We assume that the network, G, is parameterized by two blocks of weights, $\mathbf{W}$ and $\boldsymbol{\theta}$, and is of the form

$$G(\cdot, \mathbf{W}, \boldsymbol{\theta}) = \mathbf{W} F(\cdot, \boldsymbol{\theta}), \tag{1.1}$$

where F, also referred to as a feature extractor, is a parameterized, nonlinear function. The important observation here is that the DNN is nonlinear in $\boldsymbol{\theta}$ and, crucially, is *linear in* $\mathbf{W}$. Any DNN whose last layer does not contain a nonlinear activation function can be written in this form, so our definition includes many state-of-the-art DNNs; see, e.g., [27, 41, 31, 30, 44] and following works like [46, 49, 34]. In a supervised learning framework, the goal is to find a set of network weights, $(\mathbf{W}, \boldsymbol{\theta})$, such that $\mathbf{W} F(\mathbf{y}, \boldsymbol{\theta}) \approx \mathbf{c}$ for all input-target pairs $(\mathbf{y}, \mathbf{c})$ in a data space. Training the network means learning the network weights by minimizing an expected loss or discrepancy of the DNN approximation over all input-target pairs $(\mathbf{y}, \mathbf{c})$ in a training set, while generalizing well to unobserved input-target pairs.

Main contributions. In this paper, we describe `slimTrain`, a *sampled limited-memory training* method that exploits the separability of the DNN architecture to

leverage recently developed sampled Tikhonov methods for automatic regularization parameter tuning [34, 13]. For the linear weights in a regression framework, we obtain a stochastic linear least-squares problem, and we use recent work on sampled limited-memory methods to approximate the global curvature of the underlying least-squares problem. Such methods can be viewed as row-action or SA methods and can speed up the initial convergence and improve the accuracy of iterates [13]. As discussed above, applying a second-order SA method to the entire problem is not trivial and obtaining curvature information for the nonlinear weights is computationally expensive, particularly for deep networks. As our approach only incorporates curvature in the final layer of the network, where we have a linear structure, its computational overhead is minimal. In doing so, not only can we improve initial convergence of DNN training, but we also can select the regularization parameter automatically by exploiting connections between the learning rate of the linear weights and the regularization parameter for Tikhonov regularization [11]. Thus, `slimTrain` is an efficient, practical method for training separable DNNs that is memory-efficient (i.e., working only on mini-batches), exhibits faster initial convergence compared to standard SA approaches (e.g., ADAM), produces networks that generalize well, and incorporates automatic hyperparameter selection.

This paper is organized as follows. In section 2, we describe separable DNN architectures and review various approaches to train such networks, with special emphasis on variable projection. Notably, we provide new theoretical analysis to support a variable projection stochastic approximation method. In section 3, we introduce our new `slimTrain` approach that incorporates sampled limited-memory Tikhonov (`slimTik`) methods within the nonlinear learning problem. Here, we describe cross-validation-based techniques to automatically and adaptively select the regularization parameter. Numerical results are provided in section 4, and conclusions follow in section 5.

2. Exploiting separability with variable projection. Given the space of input features $\mathcal{Y} \subseteq \mathbb{R}^{n_{\text{in}}}$ and the space of target features $\mathcal{C} \subseteq \mathbb{R}^{n_{\text{target}}}$, let $\mathcal{D} \subseteq \mathcal{Y} \times \mathcal{C}$ be the data space containing input-target pairs $(\mathbf{y}, \mathbf{c}) \in \mathcal{D}$. We focus on separable DNN architectures that consist of two separate phases: a nonlinear feature extractor $F : \mathcal{Y} \times \mathbb{R}^{n_\theta} \to \mathbb{R}^{n_{\text{out}}}$ parametrized by $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$ followed by a linear model $\mathbf{W} \in \mathbb{R}^{n_{\text{target}} \times n_{\text{out}}}$. In general, the goal is to learn the network weights, $(\mathbf{W}, \boldsymbol{\theta})$, by solving the stochastic optimization problem

$$\min_{\mathbf{W},\boldsymbol{\theta}} \ \mathbb{E}\ L(\mathbf{W}F(\mathbf{y}, \boldsymbol{\theta}), \mathbf{c}) + R(\boldsymbol{\theta}) + S(\mathbf{W}), \tag{2.1}$$

where $L : \mathbb{R}^{n_{\text{target}}} \times \mathcal{C} \to \mathbb{R}$ is a loss function, and $R : \mathbb{R}^{n_\theta} \to \mathbb{R}$ and $S : \mathbb{R}^{n_{\text{target}} \times n_{\text{out}}} \to \mathbb{R}$ are regularizers. Here, $\mathbb{E}$ denotes the expected value over a distribution of input-target pairs in $\mathcal{D}$.

Choosing an appropriate loss function L is task-dependent. For example, a least-squares loss function promotes data-fitting and is well suited for function approximation tasks whereas a cross-entropy loss function is preferred for classification tasks where the network outputs are interpreted as a discrete probability distribution [28]. In this work, we focus on exploiting separability to improve DNN training for function approximation or data fitting tasks such as partial differential equation (PDE) surrogate modeling [49, 55] and dimensionality reduction such as autoencoders [23]. Hence, we restrict our focus to a stochastic least-squares loss function with Tikhonov regularization

$$\min_{\mathbf{W},\boldsymbol{\theta}} \ \Phi(\mathbf{W}, \boldsymbol{\theta}) \equiv \mathbb{E}\ \tfrac{1}{2} \|\mathbf{W}F(\mathbf{y}, \boldsymbol{\theta}) - \mathbf{c}\|_2^2 + \tfrac{\alpha}{2} \|\mathbf{L}\boldsymbol{\theta}\|_2^2 + \tfrac{\lambda}{2} \|\mathbf{W}\|_{\text{F}}^2, \tag{2.2}$$

where $\Phi : \mathbb{R}^{n_{\text{target}} \times n_{\text{out}}} \times \mathbb{R}^{n_{\boldsymbol{\theta}}} \to \mathbb{R}$ is the objective function, $\mathbf{L}$ is a user-defined operator, $\|\cdot\|_{\text{F}}$ is the Frobenius norm, and $\alpha, \lambda \geq 0$ are the regularization parameters for $\boldsymbol{\theta}$ and $\mathbf{W}$, respectively.

2.1. SA methods that exploit separability. A standard, and the current state-of-the-art, approach to solve (2.2) is stochastic optimization over both sets of weights $(\mathbf{W}, \boldsymbol{\theta})$ simultaneously (i.e., joint estimation). While generic and straightforward, this fully coupled approach can suffer from slow convergence (e.g., due to ill-conditioning) and does not attain potential benefits that can be achieved by treating the separate blocks of weights differently (e.g., exploiting the structure of the arising subproblems). We seek computational methods for training DNNs that exploit separability, i.e., we treat the two parameter sets $\boldsymbol{\theta}$ and $\mathbf{W}$ differently and exploit linearity in $\mathbf{W}$. Three general approaches to numerically tackle the optimization problem (2.2) while taking advantage of the separability are as follows.

Alternating directions. One approach that exploits separability of the variables $\boldsymbol{\theta}$ and $\mathbf{W}$ is alternating optimization [3]. For (2.2), this corresponds to alternating between two stochastic optimization problems. Note for simplicity of presentation we assume that each of following optimization problems has a unique minimizer. Suppose we initialize $\boldsymbol{\theta}_0$. Then, at the kth iteration, we embark on

$$\mathbf{W}_k = \arg\min_{\mathbf{W}} \ \Phi(\mathbf{W}, \boldsymbol{\theta}_{k-1}) \tag{2.3}$$

and

$$\boldsymbol{\theta}_k = \arg\min_{\boldsymbol{\theta}} \ \ \Phi(\mathbf{W}_k, \boldsymbol{\theta}). \tag{2.4}$$

Notice that convergence of this approach can be slow when variables are tightly coupled [2, 53]. Furthermore, this approach is not practical in our settings, since minimization problems (2.3) and (2.4) are computationally expensive, particularly the nonconvex, high-dimensional, often nonsmooth optimization problem for $\boldsymbol{\theta}$.

Block coordinate descent. A practical alternative for alternating directions is block coordinate descent. The general idea of a block coordinate descent approach for (2.2) is to approximate the alternating optimization of (2.3) and (2.4) via iterative update schemes (e.g., one iteration of an iterative optimization step) for each set of variables [53]. Note that under certain assumptions, a block coordinate descent method applied to two sets of parameters has been shown to converge [33, 42]. Although a block coordinate descent approach provides a computationally appealing alternative to the fully coupled and alternating directions approaches, this approach, like alternating directions, suffers from slow convergence when the blocks are tightly coupled.

Variable projection (VarPro). A compromise between alternating directions and block coordinate descent is to solve (2.3) with respect to $\mathbf{W}$ while performing an iterative update method for (2.4) with respect to $\boldsymbol{\theta}$. This can be seen as a stochastic approximation version of a variable projection approach [21]. Formally, we can write the iteration in terms of the *reduced* stochastic optimization problem

$$\min_{\boldsymbol{\theta}} \ \Phi^{\text{red}}(\boldsymbol{\theta}) \equiv \Phi(\widehat{\mathbf{W}}(\boldsymbol{\theta}), \boldsymbol{\theta}), \tag{2.5}$$

where

$$\widehat{\mathbf{W}}(\boldsymbol{\theta}) = \arg\min_{\mathbf{W}} \mathbb{E} \ \tfrac{1}{2} \|\mathbf{W} F(\mathbf{y}, \boldsymbol{\theta}) - \mathbf{c}\|_2^2 + \tfrac{\lambda}{2} \|\mathbf{W}\|_{\text{F}}^2. \tag{2.6}$$

Notice that (2.6) is a *stochastic* Tikhonov-regularized *linear* least-squares problem and, under the assumption that the order of expectation and differentation is interchangeable, there exists a closed form solution, i.e.,

$$\widehat{\mathbf{W}}(\boldsymbol{\theta}) = \mathbb{E}\mathbf{c}F(\mathbf{y},\boldsymbol{\theta})^\top \left(\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta}) + \boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})^\top + \lambda\mathbf{I}\right)^{-1}. \tag{2.7}$$

Here, $\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta}) = \mathbb{E}F(\mathbf{y},\boldsymbol{\theta})$ and $\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta}) = \mathbb{E}(F(\mathbf{y},\boldsymbol{\theta}) - \boldsymbol{\mu}_{\mathbf{y}})(F(\mathbf{y},\boldsymbol{\theta}) - \boldsymbol{\mu}_{\mathbf{y}})^\top$. Details of the derivation can be found in Appendix A.

2.2. Theoretical justification for VarPro in SA methods. After solving for $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ in (2.6), VarPro uses an iterative scheme, typically an SGD variant, to update $\boldsymbol{\theta}$. The key is to ensure that the mini-batch gradients used to update $\boldsymbol{\theta}$ are unbiased. To the best of our knowledge, we provide the first theoretical analysis demonstrating that VarPro in an SA setting produces an unbiased estimate of the gradient. We note that the derivation, presented for stochastic Tikhonov-regularized least-squares problems, can be extended to any objective function which is convex with respect to the linear weights, such as when using a cross-entropy loss function.

In the context of the DNN training problem, let $\mathcal{T} \subseteq \mathcal{D}$ be a finite training set. At the kth training iteration, we select a mini-batch of the training set, $\mathcal{T}_k \subset \mathcal{T}$. For the $\mathcal{T}_k$ we seek to minimize the function

$$\Phi_k(\mathbf{W},\boldsymbol{\theta}) = \frac{1}{|\mathcal{T}_k|} \sum_{(\mathbf{y},\mathbf{c})\in\mathcal{T}_k} \tfrac{1}{2}\left\|\mathbf{W}F(\mathbf{y},\boldsymbol{\theta}) - \mathbf{c}\right\|_2^2 + \tfrac{\alpha}{2}\left\|\mathbf{L}\boldsymbol{\theta}\right\|_2^2 + \tfrac{\lambda}{2}\left\|\mathbf{W}\right\|_{\mathrm{F}}^2. \tag{2.8}$$

A VarPro SA method applied to (2.5) considers the reduced functional at the kth iteration,

$$\Phi_k^{\mathrm{red}}(\boldsymbol{\theta}) = \Phi_k(\widehat{\mathbf{W}}(\boldsymbol{\theta}),\boldsymbol{\theta}), \tag{2.9}$$

where $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ is obtained from (2.6), i.e., the solution to the stochastic Tikhonov-regularized linear least-squares problem *over the entire data space*.

To update the nonlinear weights, we select a "descent" direction $\mathbf{p}_k$ with respect to $\boldsymbol{\theta}$ and compute the next iterate,

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + \gamma\mathbf{p}_k(\boldsymbol{\theta}_{k-1}; \widehat{\mathbf{W}}(\boldsymbol{\theta}_{k-1})). \tag{2.10}$$

Here, γ denotes an appropriate learning rate and $\mathbf{p}_k$ is a direction that is computed based on the current estimate of $\boldsymbol{\theta}_{k-1}$ with respect to the current batch $\mathcal{T}_k$. The selection of $\mathbf{p}_k$ depends on the chosen stochastic optimization method and requires knowing information about the derivative of (2.8). Explicitly, we compute the derivative of Φ_k^{red} with respect to $\boldsymbol{\theta}$ by

$$\begin{aligned} \mathrm{D}_{\boldsymbol{\theta}}\Phi_k^{\mathrm{red}}(\boldsymbol{\theta}) &= \mathrm{D}_{\boldsymbol{\theta}}\Phi_k(\widehat{\mathbf{W}}(\boldsymbol{\theta}),\boldsymbol{\theta}) \\ &= \left[\mathrm{D}_{\mathbf{W}}\Phi_k(\mathbf{W},\boldsymbol{\theta})\right]_{\mathbf{W}=\widehat{\mathbf{W}}(\boldsymbol{\theta})} \cdot \mathrm{D}_{\boldsymbol{\theta}}\widehat{\mathbf{W}}(\boldsymbol{\theta}) + \left[\mathrm{D}_{\widetilde{\boldsymbol{\theta}}}\Phi_k(\widehat{\mathbf{W}}(\boldsymbol{\theta}),\widetilde{\boldsymbol{\theta}})\right]_{\widetilde{\boldsymbol{\theta}}=\boldsymbol{\theta}}. \end{aligned} \tag{2.11}$$

Note that, contrary to VarPro derivations in deterministic settings [21, 14, 34], the first term in (2.11) does not vanish. This is because $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ satisfies the optimality conditions for Φ, the objective function for expected value minimization problem (2.6), but may not be optimal for Φ_k, the objective function for the current batch. However,

we observe that the term vanishes in expectation over all samples, that is,

$$\begin{aligned}\mathbb{E}\left(\left[\mathrm{D}_{\mathbf{W}}\Phi_k(\mathbf{W},\boldsymbol{\theta})\right]_{\mathbf{W}=\widehat{\mathbf{W}}(\boldsymbol{\theta})}\cdot \mathrm{D}_{\boldsymbol{\theta}}\widehat{\mathbf{W}}(\boldsymbol{\theta})\right) &= \left[\mathrm{D}_{\mathbf{W}}\mathbb{E}\,\Phi_k(\mathbf{W},\boldsymbol{\theta})\right]_{\mathbf{W}=\widehat{\mathbf{W}}(\boldsymbol{\theta})}\cdot \mathrm{D}_{\boldsymbol{\theta}}\widehat{\mathbf{W}}(\boldsymbol{\theta})\\ &= \left[\mathrm{D}_{\mathbf{W}}\Phi(\mathbf{W},\boldsymbol{\theta})\right]_{\mathbf{W}=\widehat{\mathbf{W}}(\boldsymbol{\theta})}\cdot \mathrm{D}_{\boldsymbol{\theta}}\widehat{\mathbf{W}}(\boldsymbol{\theta})\\ &= \mathbf{0}.\end{aligned} \tag{2.12}$$

Since SA methods can handle unbiased noisy gradients, one could define a VarPro SGD approach using the following unbiased estimator for the gradient:

$$\mathbf{p}_k(\boldsymbol{\theta};\widehat{\mathbf{W}}(\boldsymbol{\theta})) = -\left[\mathrm{D}_{\widetilde{\boldsymbol{\theta}}}\Phi_k(\widehat{\mathbf{W}}(\boldsymbol{\theta}),\widetilde{\boldsymbol{\theta}})\right]^{\top}_{\widetilde{\boldsymbol{\theta}}=\boldsymbol{\theta}}, \tag{2.13}$$

where the derivative is

$$\begin{aligned}\mathrm{D}_{\boldsymbol{\theta}}\Phi_k(\mathbf{W},\boldsymbol{\theta}) &= \mathrm{D}_{\boldsymbol{\theta}}\left(\frac{1}{|\mathcal{T}_k|}\sum_{(\mathbf{y},\mathbf{c})\in\mathcal{T}_k}\tfrac{1}{2}\left\|\mathbf{W}F(\mathbf{y},\boldsymbol{\theta})-\mathbf{c}\right\|_2^2+\tfrac{\alpha}{2}\left\|\mathbf{L}\boldsymbol{\theta}\right\|_2^2\right)\\ &= \frac{1}{|\mathcal{T}_k|}\sum_{(\mathbf{y},\mathbf{c})\in\mathcal{T}_k}\left(\mathbf{W}F(\mathbf{y},\boldsymbol{\theta})^{\top}-\mathbf{c}\right)\mathbf{W}\mathrm{D}_{\boldsymbol{\theta}}F(\mathbf{y},\boldsymbol{\theta})+\alpha\boldsymbol{\theta}^{\top}\mathbf{L}^{\top}\mathbf{L}.\end{aligned} \tag{2.14}$$

Note that $\mathrm{D}_{\boldsymbol{\theta}}F(\mathbf{y},\boldsymbol{\theta})$ can be obtained through back propagation which can be parallelized over samples. Because (2.11) is equal to the gradient of the full objective function Φ in expectation, we say the update for $\boldsymbol{\theta}$ in (2.10) using (2.13) is unbiased.

2.3. Challenges of VarPro in stochastic optimization. The appeal of a VarPro approach is marred by the impracticality of computing $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ in (2.6). For each mini-batch update of $\boldsymbol{\theta}$, one would need to recompute $\widehat{\mathbf{W}}(\boldsymbol{\theta})$, which requires propagating many samples through the network. Since a computation is costly, in terms of time and storage, we can only obtain an approximation of $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ in practice.

One way to approximate $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ is to replace the vector $\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})$ and the matrix $\mathbb{E}\mathbf{c}F(\mathbf{y},\boldsymbol{\theta})^{\top}$ with sample mean approximations and the covariance matrix $\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta})$ with a sample covariance matrix. The accuracy of the approximation, and hence the expected bias of the gradients for the nonlinear weights, will depend on the size of the sample. However, these quantities still depend on $\boldsymbol{\theta}$, and hence for any iterative process where $\boldsymbol{\theta}$ is being updated, these values need to be recomputed at each iteration.

A practical strategy to approximate $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ is to use a sample average approximation (SAA) approach. In SAA methods, one first approximates the expected loss using a (large and representative) sample. The resulting optimization problem is deterministic and a wide range of optimization methods with proven theoretical guarantees can be used. For example, inexact Newton methods may be utilized to obtain fast convergence [5, 36, 54]. Solving a deterministic SAA optimization problem with an efficient solver guarantees the linear model fits the sampled data optimally at each training iteration. Note that if an SAA approach were used to solve both (2.5) and (2.6) with the same (fixed) sample set, then this would be equivalent to the variable projection SAA approach described in [34]. Indeed, there are various recent works [34, 40, 16] that exploit the separable structures (1.1) of neural networks in SAA settings in order to accelerate convergence. However, the disadvantage of SAA methods is that very large batch sizes are needed to obtain sufficient accuracy of the

approximation and to prevent overfitting. Although parallel computing tools (e.g., GPU and distributed computing) and strategies such as repeated sampling may be used, the storage requirements for SAA methods remain prohibitively large.

To summarize section 2, the widely used, fully coupled approach (optimizing over $\boldsymbol{\theta}$ and $\mathbf{W}$ simultaneously) and the alternating minimization approach represent two extremes: the former is a tractable approach, but ignores the separable structure while the latter exploits separability, but is computationally intractable in the stochastic setting. Although a block coordinate descent approach decouples the parameters and replaces expensive optimization solves with iterative updates, a VarPro approach can mathematically eliminate the linear weights, thereby reducing the problem to a stochastic optimization problem in $\boldsymbol{\theta}$ only. The resulting noisy gradient estimates for $\boldsymbol{\theta}$ are unbiased when $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ is computed exactly, making VarPro compatible with SGD variants to update $\boldsymbol{\theta}$. However, computing $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ when also updating $\boldsymbol{\theta}$ is intractable and poor approximations may lead to a large bias in the gradients for $\boldsymbol{\theta}$. Hence, providing an effective and efficient way to approximate $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ is crucial to obtain a practical implementation of VarPro stochastic optimization.

3. Sampled limited-memory DNN training with `slimTrain`. We present `slimTrain` as a tractable variant of VarPro in the SA setting, which adopts a sampled limited-memory Tikhonov scheme to approximate the linear weights and to estimate an effective regularization parameter for the linear weights. The key idea is to approximate the linear weights using the output features obtained from recent mini-batches and nonlinear weight iterates. By storing the output features from the most recent iterates, `slimTrain` avoids additional forward and backward propagations through the neural network which, especially for deep networks, is computationally the most expensive part of training, and hence adds only a small computational overhead to the training.

3.1. Sampled Tikhonov methods to approximate $\widehat{\mathbf{W}}(\boldsymbol{\theta})$. As described in section 2, approximating $\widehat{\mathbf{W}}(\boldsymbol{\theta})$ well is challenging, but important for reducing bias in the gradient for $\boldsymbol{\theta}$; see (2.12). This motivates us to use state-of-the-art iterative sampling approaches to solve stochastic, Tikhonov-regularized, *linear* least-squares problems. For exposition purposes, we first reformulate (2.6) as

$$\widehat{\mathbf{w}}(\boldsymbol{\theta}) = \arg\min_{\mathbf{w}} \ \mathbb{E}\ \tfrac{1}{2}\,\|\mathbf{A}(\mathbf{y},\boldsymbol{\theta})\mathbf{w} - \mathbf{c}\|_2^2 + \tfrac{\lambda}{2}\|\mathbf{w}\|_2^2, \tag{3.1}$$

where $\mathbf{w} = \mathrm{vec}(\mathbf{W}) \in \mathbb{R}^{n_{\mathrm{target}} n_{\mathrm{out}}}$ concatenates the columns of $\mathbf{W}$ in a single vector, $\mathbf{A}(\mathbf{y},\boldsymbol{\theta}) = F(\mathbf{y},\boldsymbol{\theta})^\top \otimes \mathbf{I}_{n_{\mathrm{target}}}$ with $\otimes$ denoting the Kronecker product. This Kronecker structure extends to a mini-batch $\mathcal{T}_k$. Suppose we order the samples $(\mathbf{y}_i, \mathbf{c}_i) \in \mathcal{T}_k$ for $i = 1, \ldots, |\mathcal{T}_k|$. Then, the final layer can be expressed for vectorized linear weights as

$$\mathbf{W}\mathbf{Z}_k(\boldsymbol{\theta}) \approx \mathbf{C}_k \qquad \underset{\mathrm{mat}}{\overset{\mathrm{vec}}{\longleftrightarrow}} \qquad \mathbf{A}_k(\boldsymbol{\theta})\mathbf{w} \approx \mathbf{b}_k,$$

where

$$\begin{aligned}
\mathbf{Z}_k(\boldsymbol{\theta}) &= \begin{bmatrix} F(\mathbf{y}_1, \boldsymbol{\theta}) & \cdots & F(\mathbf{y}_{|\mathcal{T}_k|}, \boldsymbol{\theta}) \end{bmatrix} && \in \mathbb{R}^{n_{\text{out}} \times |\mathcal{T}_k|}, \\
\mathbf{C}_k &= \begin{bmatrix} \mathbf{c}_1 & \cdots & \mathbf{c}_{|\mathcal{T}_k|} \end{bmatrix} && \in \mathbb{R}^{n_{\text{target}} \times |\mathcal{T}_k|}, \\
\mathbf{A}_k(\boldsymbol{\theta}) &= \mathbf{Z}_k(\boldsymbol{\theta})^\top \otimes \mathbf{I}_{n_{\text{target}}} && \in \mathbb{R}^{|\mathcal{T}_k| n_{\text{target}} \times n_{\text{out}} n_{\text{target}}}, \quad \text{and} \\
\mathbf{b}_k &= \operatorname{vec}(\mathbf{C}_k) = \begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_{|\mathcal{T}_k|} \end{bmatrix} && \in \mathbb{R}^{n_{\text{target}} |\mathcal{T}_k|}.
\end{aligned}$$

Henceforth, in this section, since $\boldsymbol{\theta}$ is fixed in (3.1), we use $\mathbf{A}_k = \mathbf{A}_k(\boldsymbol{\theta})$ for presentation purposes.

Introduced in [47, 13], sampled Tikhonov (`sTik`) and sampled limited-memory Tikhonov (`slimTik`) methods are specialized iterative methods developed for solving stochastic regularized linear least-squares problems. For an initial iterate $\mathbf{w}_0$, the kth `sTik` iterate is given by

$$\mathbf{w}_k(\Lambda) = \arg\min_{\mathbf{w}} \frac{1}{2} \left\| \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_{k-1} \\ \mathbf{A}_k \\ \sqrt{\Lambda + \sum_{i=1}^{k-1} \Lambda_i} \mathbf{I} \end{bmatrix} \mathbf{w} - \begin{bmatrix} \mathbf{A}_1 \mathbf{w}_{k-1} \\ \vdots \\ \mathbf{A}_{k-1} \mathbf{w}_{k-1} \\ \mathbf{b}_k \\ \frac{\sum_{i=1}^{k-1} \Lambda_i}{\sqrt{\Lambda + \sum_{i=1}^{k-1} \Lambda_i}} \mathbf{w}_{k-1} \end{bmatrix} \right\|_2^2, \tag{3.2}$$

where $\mathbf{w}_{k-1}$ is the previously computed estimate, $\mathbf{A}_1, \ldots, \mathbf{A}_{k-1}$ are matrices containing previously computed output features, $\Lambda + \sum_{i=1}^{k-1} \Lambda_i > 0$, and Λ is a regularization parameter estimate. The `sTik` iterates can also be expressed in update form as an SA method,

$$\mathbf{w}_k(\Lambda) = \mathbf{w}_{k-1} - \mathbf{B}_k(\Lambda) \mathbf{g}_k(\mathbf{w}_{k-1}, \Lambda), \tag{3.3}$$

with $\mathbf{g}_k(\mathbf{w}_{k-1}, \Lambda) = \mathbf{A}_k^\top (\mathbf{A}_k \mathbf{w}_{k-1} - \mathbf{b}_k) + \Lambda \mathbf{w}_{k-1}$ containing gradient information for the current mini-batch and $\mathbf{B}_k(\Lambda) = ((\Lambda + \sum_{i=1}^{k-1} \Lambda_i)\mathbf{I} + \sum_{i=1}^{k} \mathbf{A}_i^\top \mathbf{A}_i)^{-1}$ containing global curvature information of the least-squares problem. Note that contrary to standard SA methods, (3.3) does not require a learning rate or a line search parameter. The learning rate can be interpreted as one, which is optimal for Newton's method.

Importantly, the regularization parameter λ in (3.1), which is typically required to be set in advance, has been replaced with a new parameter estimate Λ which can be chosen adaptively at each iteration. Each Λ_k corresponds to a regularization parameter at iteration k and can change at each iteration (Λ_j, $j = 1, \ldots, k-1$, correspond to regularization parameters from previous iterations). In fact, the parameters λ and Λ_k's are directly connected. After one epoch (e.g., iterating through all training samples), the `sTik` iterate is identical to the Tikhonov solution of (3.1) with $\lambda = \sum_{i=1}^{k} \Lambda_i$, where k is the number of iterations required for one epoch. We exemplify the convergence of `sTik` in Figure 1 when approximating the MATLAB `peaks` function [25]. Moreover, it has been shown that `sTik` iterates converge asymptotically to a Tikhonov solution and subsequently adaptive parameter selection methods were developed in [47].

Since (3.2) and (3.6) correspond to standard Tikhonov problems, extensions of standard regularization parameters methods, such as the discrepancy principle

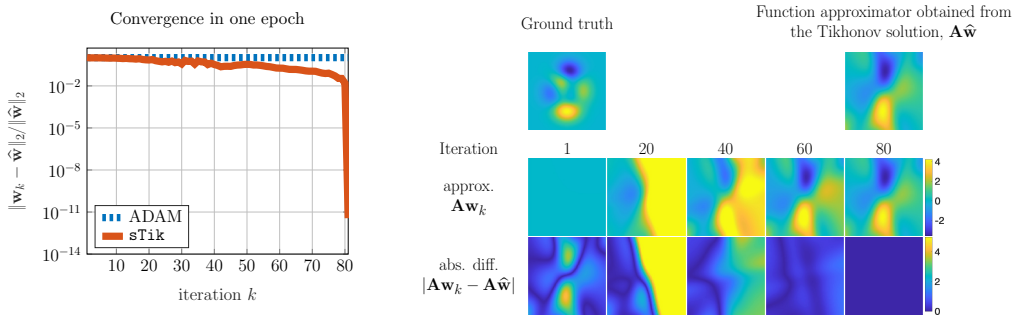


FIG. 1. *Illustration comparing convergence of* `sTik` *and ADAM with a fixed regularization parameter for solving* (3.1). *We consider approximating the* MATLAB `peaks` *function,* $f : \mathbb{R}^2 \to \mathbb{R}$, *using training data located on a uniform grid. We apply a fixed nonlinear transformation to each point in the domain to form the rows of* $\mathbf{A}_k \in \mathbb{R}^{|\mathcal{T}_k| \times |\mathbf{w}|}$ *and the corresponding true function values are stored in* $\mathbf{b}_k \in \mathbb{R}^{|\mathcal{T}_k|}$, *where* $|\mathbf{w}|$ *is the number of linear weights. The constant regularization parameters are* $\Lambda_k = \frac{\lambda}{80}$, *where* 80 *is the number of iterations in one epoch. Both* $|\mathbf{w}|$ *and* λ *are chosen arbitrarily and the number of iterations depends on the number of training points and the batch size. The best linear weights are given by the Tikhonov solution,* $\widehat{\mathbf{w}} = (\mathbf{A}^\top\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^\top\mathbf{b}$, *and the corresponding best function approximator is* $\mathbf{A}\widehat{\mathbf{w}}$. *To the left, we plot the convergence of the relative error* $\|\mathbf{w}_k - \widehat{\mathbf{w}}\|_2/\|\widehat{\mathbf{w}}\|_2$ *for each iteration* k *in a single epoch. By design,* `sTik` *converges to the least-squares solution in one epoch whereas ADAM makes little progress. To the right, the middle row shows the function approximations for different* `sTik` *iterates,* $\mathbf{A}\mathbf{w}_k$, *and the bottom row shows the absolute difference of the approximation with the best approximation. The top row depicts the true* `peaks` *function (left) and the best approximation obtained from the Tikhonov solution (right).*

(DP), unbiased predictive risk minimization (UPRE), and generalized cross validation (GCV) techniques can be utilized. Indeed, *sampled* regularization parameter selection methods sDP, sUPRE, and sGCV for `sTik` and `slimTik` and their connection to the overall regularization parameter λ can be found in [47]. In this work, we focus on regularization parameter selection via sGCV since this method does not require any further hyperparameters (e.g., noise level estimates for the mini-batch), and we have observed that sGCV provides favorable λ estimates. For details on the GCV function, see original works [22, 51] and books [26, 50]. The sGCV parameter at the kth `slimTik` iterate can be computed as

$$\Lambda_k = \arg\min_{\Lambda} \frac{|\mathcal{T}_k| \, \|\mathbf{A}_k\mathbf{w}_k(\Lambda) - \mathbf{b}_k\|_2^2}{\big(|\mathcal{T}_k| - \mathrm{tr}\big(\mathbf{A}_k\mathbf{T}_k(\Lambda)\mathbf{A}_k^\top\big)\big)^2}, \tag{3.4}$$

where

$$\mathbf{T}_k(\Lambda) = \left(\left(\Lambda + \sum_{i=1}^{k-1}\Lambda_i\right)\mathbf{I}_n + \sum_{i=k-r}^{k}\mathbf{A}_i^\top\mathbf{A}_i\right)^{-1}. \tag{3.5}$$

For some problems, e.g., inverse problems where $\mathbf{A}_k$ represent large-scale forward model matrices, `sTik` may not be practical since each iteration requires either solving a least-squares problem (3.2) whose coefficient matrix is growing at each iteration or updating matrix $\mathbf{B}_k$. To alleviate the memory burden, a variant of `sTik` called the sampled limited-memory Tikhonov (`slimTik`) method was proposed in [47]. Let

$r \in \mathbb{N}_0$ be a memory depth parameter. Then, the kth `slimTik` iterate has the form

$$\mathbf{w}_k(\Lambda) = \arg\min_{\mathbf{w}} \frac{1}{2} \left\| \begin{bmatrix} \mathbf{A}_{k-r} \\ \vdots \\ \mathbf{A}_{k-1} \\ \mathbf{A}_k \\ \sqrt{\Lambda + \sum_{i=1}^{k-1} \Lambda_i} \mathbf{I} \end{bmatrix} \mathbf{w} - \begin{bmatrix} \mathbf{A}_{k-r}\mathbf{w}_{k-1} \\ \vdots \\ \mathbf{A}_{k-1}\mathbf{w}_{k-1} \\ \mathbf{b}_k \\ \frac{\sum_{i=1}^{k-1} \Lambda_i}{\sqrt{\Lambda + \sum_{i=1}^{k-1} \Lambda_i}} \mathbf{w}_{k-1} \end{bmatrix} \right\|_2^2. \tag{3.6}$$

We provide a few remarks about the `slimTik` method. For linear least-squares problems, it can be shown that for the case $r = 0$, the `slimTik` method is equivalent to the stochastic block Kaczmarz method. Furthermore, for linear least-squares problems with a fixed regularization parameter, theoretical convergence results for `slimTik` with memory $r = 0$ were developed in [13]. We point out that limited memory methods like `slimTik` were initially developed to address problems where the size of $\mathbf{w}$ is massive, but this is not necessarily the case in DNN training where the number of weights in $\mathbf{w}$ may be modest. However, as we will see in subsection 3.2, a limited memory approach is suitable and can even be desirable in the context of solving nonlinear problems, where nonlinear parameters have direct impact on the model matrices $\mathbf{A}_k$. In this work, we are interested in incorporating extensions of `slimTik` with *adaptive* regularization parameter selection for nonlinear problems that exploit separability.

3.2. `slimTrain`. Our proposed SA algorithm, `slimTrain` takes advantage of the separable structure of many DNNs and integrates the `slimTik` method for efficiently updating the linear parameters and for automatic regularization parameter tuning. We consider the `slimTik` update of $\mathbf{W}$ to serve as an approximation of the eliminated linear weights in VarPro SA from (2.6). Specifically, at the kth iteration, $\widehat{\mathbf{W}}(\boldsymbol{\theta}) \approx \mathbf{W}_k = \mathrm{mat}(\mathbf{w}_k(\Lambda_k))$, where

$$\mathbf{w}_k(\Lambda_k) = \arg\min_{\mathbf{w}} \left\| \begin{bmatrix} \mathbf{M}_k \\ \mathbf{A}_k(\boldsymbol{\theta}_{k-1}) \\ \sqrt{\sum_{i=1}^{k} \Lambda_i} \mathbf{I} \end{bmatrix} \mathbf{w} - \begin{bmatrix} \mathbf{M}_k\mathbf{w}_{k-1} \\ \mathbf{b}_k \\ \frac{\sum_{i=1}^{k-1} \Lambda_i}{\sqrt{\sum_{i=1}^{k} \Lambda_i}} \mathbf{w}_{k-1} \end{bmatrix} \right\|_2^2, \tag{3.7}$$

with

$$\mathbf{M}_k = \begin{bmatrix} \mathbf{A}_{k-r}(\boldsymbol{\theta}_{k-r-1}) \\ \vdots \\ \mathbf{A}_{k-1}(\boldsymbol{\theta}_{k-2}) \end{bmatrix} \tag{3.8}$$

and Λ_k is computed using the sGCV method (cf. (3.4)). For the first r iterates, matrices with nonpositive indices are set to zero. Notice that this is not equivalent to the `slimTik` method for $\arg\min_{\mathbf{W}} \Phi(\mathbf{W}, \boldsymbol{\theta}_{k-1})$, since there is no inner iterative process and because of the dependence on previous $\boldsymbol{\theta}_j$. A summary of the algorithm is provided in Algorithm 3.1. We impose standard stopping criteria for supervised learning methods [23].

We note that an SA method that incorporates the `slimTik` method was considered for separable nonlinear inverse problems in [11], but there are some distinctions. First, the results in [11] use a fixed regularization parameter, but here we allow for adaptive parameter choice, which has previously only been considered for linear problems. We note that updating regularization parameters in nonlinear problems (especially

Algorithm 3.1 `slimTrain`: sampled limited-memory training for separable DNNs.

Training Data: $\mathcal{T} \subseteq \mathcal{D}$
Hyperparameters: memory depth $r \in \mathbb{N}_0$, mini-batch size n_{batch}, learning rate γ, regularization parameter α
Initialize: $\boldsymbol{\theta}_0 \in \mathbb{R}^{n_\theta}$, $\mathbf{W}_0 \in \mathbb{R}^{n_{\text{target}} \times n_{\text{out}}}$
while stopping criteria not satisfied **do**
  randomly partition $\mathcal{T}$ into mini-batches such that $\mathcal{T} = \bigsqcup_k \mathcal{T}_k$ and $|\mathcal{T}_k| = n_{\text{batch}}$
  for $k = 1, \ldots, \lfloor |\mathcal{T}|/n_{\text{batch}} \rfloor$ **do**
    select mini-batch $\mathcal{T}_k$
    forward propagate network to obtain $\mathbf{A}_k(\boldsymbol{\theta}_{k-1})$
    select Λ_k using sGCV ▷ (3.4)
    compute $\mathbf{W}_k = \text{mat}(\mathbf{w}_k(\Lambda_k))$ ▷ (3.7)
    compute derivatives of $\boldsymbol{\theta}$ via backpropagation ▷ (2.14)

$$[\mathrm{D}_{\boldsymbol{\theta}} \Phi_k(\mathbf{W}_k, \boldsymbol{\theta})]_{\boldsymbol{\theta}=\boldsymbol{\theta}_{k-1}} \equiv \left[\mathrm{D}_{\boldsymbol{\theta}}(\tfrac{1}{2}\|\mathbf{A}_k(\boldsymbol{\theta})\mathbf{w}_k(\Lambda_k), \boldsymbol{\theta})\|_2^2 + \tfrac{\alpha}{2}\|\mathbf{L}\boldsymbol{\theta}\|_2^2\right]_{\boldsymbol{\theta}=\boldsymbol{\theta}_{k-1}}$$

    select search direction $\mathbf{p}_k$
    update $\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + \gamma \mathbf{p}_k(\boldsymbol{\theta}_{k-1}; \mathbf{W}_k)$
    update memory matrix ▷ (3.8)

$$\mathbf{M}_{k+1} = \begin{bmatrix} \mathbf{A}_{k-r+1}(\boldsymbol{\theta}_{k-r})^\top & \ldots & \mathbf{A}_k(\boldsymbol{\theta}_{k-1})^\top \end{bmatrix}^\top$$

  end for
end while

stochastic ones) is a challenging task, and currently there are no theoretical justifications. Second, all forward matrices were recomputed for each new set of nonlinear parameters in [11]. That is, for updated estimate $\boldsymbol{\theta}_{k-1}$,

$$\mathbf{M}_k = \begin{bmatrix} \mathbf{A}_{k-r}(\boldsymbol{\theta}_{k-1}) \\ \vdots \\ \mathbf{A}_{k-1}(\boldsymbol{\theta}_{k-1}) \end{bmatrix}. \tag{3.9}$$

Such an approach would be computationally demanding for DNN learning problems, since this would require revisiting previous mini-batches and recomputing the forward propagation matrix for new parameters $\boldsymbol{\theta}_{k-1}$. Instead, we propose using (3.8), and we will show that these methods can perform well in practice.

4. Numerical results. We present a numerical study of training separable DNNs using `slimTrain` with automatic regularization parameter selection. In this section, we first provide a general discussion on numerical considerations of our proposed method in subsection 4.1. In subsection 4.2, we explore the relationship between various `slimTrain` hyperparameters (e.g., batch size, memory depth, regularization parameters) in a function approximation task. Our results show that automatic regularization parameter selection can mitigate poor hyperparameter selection. In subsection 4.3, we apply `slimTrain` to a PDE surrogate modeling task and show that it outperforms the state-of-the-art ADAM for the default hyperparameters. In subsection 4.4, we apply `slimTrain` to a dimensionality-reduction task in which the linear

weights are applied via a convolution. Notably, we observe faster convergence and, particularly with limited training data, improved results compared to ADAM.

4.1. Efficient implementation. Training separable DNNs with `slimTrain` adds some computational costs compared to existing SA methods like ADAM; however, those are modest in many cases and the overhead in computational time can be reduced by an efficient implementation. The additional costs stem from solving for the optimal linear weights in (3.6) and approximating the optimal regularization parameter using the sGCV function (3.4). The costs of these steps depend on the size of the nonlinear feature matrix, $\mathbf{A}_k \in \mathbb{R}^{|\mathcal{T}_k| n_{\text{target}} \times n_{\text{out}} n_{\text{target}}}$, the size of the memory matrix, $\mathbf{M}_k$, which contains r blocks of nonlinear features from previous batches, and the number of linear weights. In the case when the linear weights are applied via dense matrix, we can exploit the Kronecker structure in our problem; see subsection 3.1 for details. The Kronecker structure results in solving n_{target} least-squares problems simultaneously where each problem is moderate in size (typically, on the order of 10^2 or 10^3). Due to the modest problem size, we use a singular value decomposition (SVD) to solve the least-squares problem. We also reuse the SVD factors for efficiently adapting the regularization parameter. For the `peaks` and surrogate modeling experiments (subsections 4.2 and 4.3), we implement the Kronecker-structure framework in MATLAB. The code is available in the `Meganet.m` repository on https://github.com/XtractOpen/Meganet.m.

In the case when the linear weights parameterize a linear operator (most importantly, a convolution), efficient iterative solvers, such as LSQR [38] that only require matrix-vector products and avoid forming the matrix explicitly, can be used to find the optimal linear weights. Such methods were employed in [11] where the authors applied `slimTik` to massive, separable nonlinear inverse problems where the data matrix could not be represented all-at-once. Modifications of the sGCV function using stochastic trace estimators can then be used for estimating the regularization parameter efficiently; for more details, see [47].

In subsection 4.4, the linear weights parameterize a convolution layer with several input but only one output channel. Exploiting the separability between the different channels and the small number of weights per channel, we form the nonlinear feature matrix, $\mathbf{A}_k$, explicitly in our implementation. This allows us to use the same SVD-based automatic regularization parameter selection as in the dense case. To be precise, the columns of $\mathbf{A}_k$ are shifted copies of the batch data, which is large, but accessible (on the order of 10^5). Importantly, the number of columns (copies of the data) is small because the number of weights parameterizing the linear operator, denoted $|\mathbf{w}|$, is small (on the order of 10^2). We can construct the data matrix $\mathbf{A}_k$ efficiently by taking advantage of the structure of convolutional operators; each channel has its own linear weights and the samples share the same weights. For storage efficiency, we can form the smaller matrix $\mathbf{A}_k^\top \mathbf{A}_k \in \mathbb{R}^{|\mathbf{w}| \times |\mathbf{w}|}$ one time, and use the update rule (3.3) to adjust the linear weights. We implement the convolutional operator framework in PyTorch [39]. The code is available on `github` at https://github.com/elizabethnewman/slimTrain.

4.2. Peaks. To explore the hyperparameters in `slimTrain`, we examine a scalar function approximation task. We train a DNN to fit the `peaks` function in MATLAB, which is a mixture of two-dimensional Gaussians. We use a small residual neural network (ResNet) [27] with a width of $w = 8$ and a depth of $d = 8$ corresponding to a final time of $T = 5$. Further details about the ResNet architecture can be found in Appendix B. The nonlinear feature extractor maps $F : \mathbb{R}^2 \times \mathbb{R}^{528} \to \mathbb{R}^8$, where

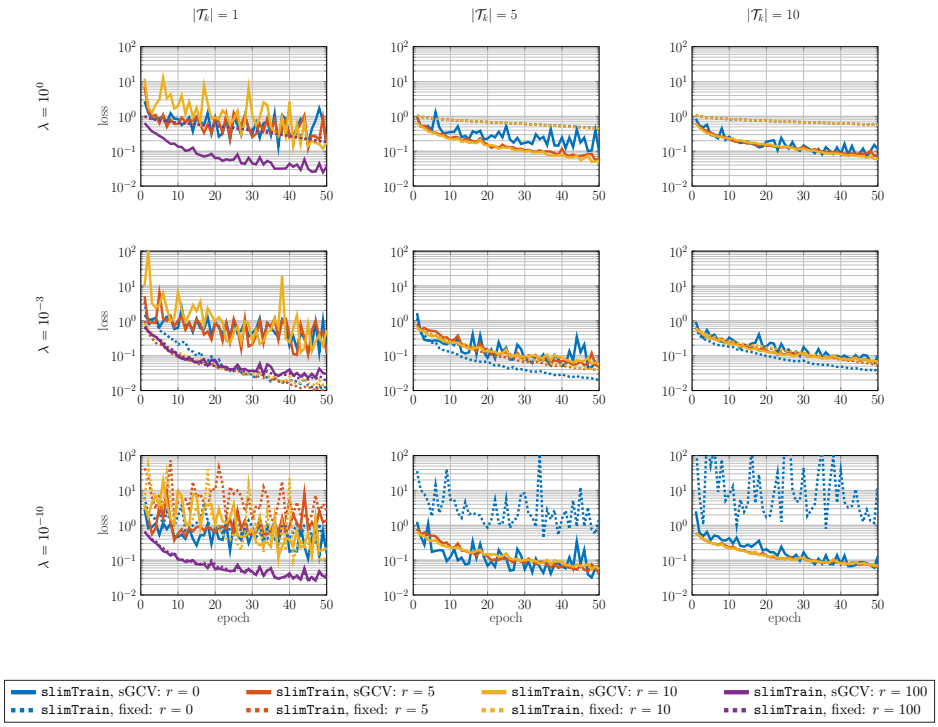


FIG. 2. *Convergence of training loss for the* `peaks` *experiment when training with* `slimTrain` *with a learning rate of* $\gamma = 10^{-3}$. *Each row corresponds to a different choice of fixed regularization parameter for* $\mathbf{W}$, $\lambda = 10^{0}, 10^{-3}, 10^{-10}$. *When training with adaptive regularization parameter selection, the initial regularization parameter* Λ_0 *is set to be the same as the fixed regularization parameter. Each column corresponds to a different batch size,* $|\mathcal{T}_k| = 1, 5, 10$. *Each convergence plot consists of dashed and solid lines corresponding to using a fixed regularization parameter and adaptively choosing the regularization parameter using sGCV, respectively. The color of each line corresponds to memory depth* $r = 0, 5, 10$ *and, additionally,* $r = 100$ *for* $|\mathcal{T}_k| = 1$.

528 is the number of weights in $\boldsymbol{\theta}$. The final linear layer introduces the weights $\mathbf{W} \in \mathbb{R}^{1\times 9}$, where the number of columns equals the width of the ResNet plus an additive bias. Our training data consists of 2,000 points sampled uniformly on the domain $[-3, 3] \times [-3, 3]$. We display the convergence of `slimTrain` for various combinations of hyperparameters in Figure 2.

The interplay between number of output features, the batch size, and the memory depth is apparent in Figure 2. In this scalar-function example, we seek nine weights (i.e., $\mathbf{W} \in \mathbb{R}^{1\times 9}$) to fit $(r+1)|\mathcal{T}_k|$ samples. With small memory depth and batch size, the problem is underdetermined (or not sufficiently overdetermined) and solving for $\mathbf{W}$ significantly overfits the given batch at each iteration. This results in the slow, oscillatory convergence behavior, particularly with a batch size of $|\mathcal{T}_k| = 1$ (Figure 2, first column). When the memory depth and batch size are large enough (e.g., $r = 100$ in the $|\mathcal{T}_k| = 1$), the linear least-squares problem is sufficiently overdetermined and the training loss converges faster and to a lower value (Figure 2, purple line in first column).

Solving the optimization problem and decreasing the loss of the training data is

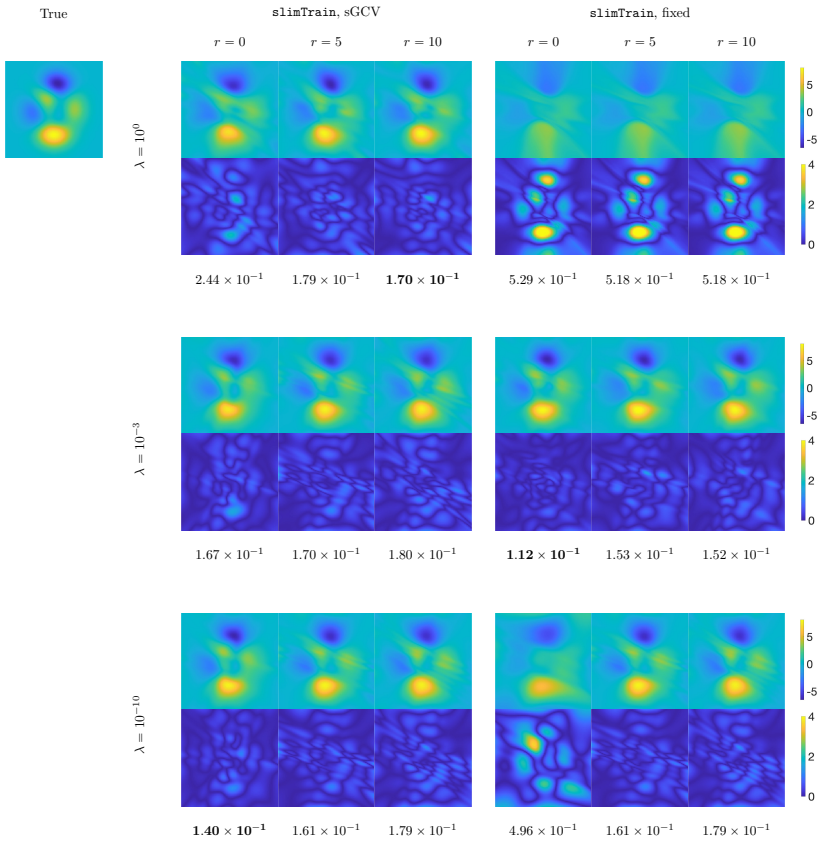


FIG. 3. *DNN approximations for the* `peaks` *experiment with batch size of* $|\mathcal{T}_k| = 5$ *and a learning rate of* $\gamma = 10^{-3}$. *The results use the network weights corresponding to the lowest validation loss for each training method. Each block row corresponds to a different choice of fixed regularization parameter for* $\mathbf{W}$, $\lambda = 10^0, 10^{-3}, 10^{-10}$. *The top rows of images in each block depict the DNN approximations of the* `peaks` *function. The bottom rows of images in each block depict the absolute difference of the DNN approximations and the true* `peaks` *function. The DNN weights used provided the smallest validation loss during training. The relative error of the DNN approximation versus the true function is displayed below the corresponding absolute difference image.*

a proxy to the goal of DNN training: to generalize to unseen data. To illustrate the generalizability of DNNs trained with `slimTrain`, we display the DNN approximations in Figure 3 corresponding to a batch size of $|\mathcal{T}_k| = 5$ (second column of Figure 2) of the convergence plots.

Exemplified in Figure 3, the choice of regularization parameter for $\mathbf{W}$ significantly impacts the approximation quality of the network when training with a fixed regularization parameter (Figure 3, second column set of figures). If the optimization problem over-regularizes the linear weights ($\lambda = 10^0$), the DNN approximation is smoother than the true `peaks` function and does not fit the extremes tightly (Figure 3, first row). In the under-regularized case ($\lambda = 10^{-10}$) with a small memory depth ($r = 0$), $\mathbf{W}$ overfits the batches and the DNN approximation does not gener-

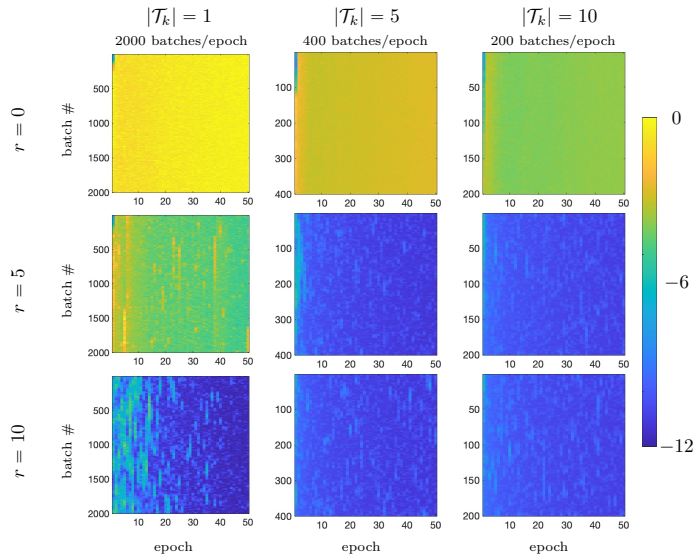


FIG. 4. *Regularization parameters selected by approximately minimizing the sGCV function in the* `peaks` *example for a learning rate of* $\gamma = 10^{-3}$ *and an initial regularization parameter of* $\Lambda_0 = 10^{-10}$. *Each column corresponds to a different batch size,* $|\mathcal{T}_k| = 1, 5, 10$, *respectively. Each row corresponds to a different memory depth,* $r = 0, 5, 10$, *respectively. In each image, the horizontal axis is the number of epochs, in this case* 50, *and the vertical axis is the number of iterations per epoch. For example, when the batch size is* $|\mathcal{T}_k| = 5$, *the vertical axis has* 400 *iterations (the number of training samples divided by the batch size). Each pixel corresponds to the regularization parameter used for a particular batch and the batches change because we shuffle the training data at the start of each epoch. The images are displayed in log scale. The first few regularization parameters in each case are small (top left corner of each image) because we start with a small initial regularization parameter.*

alize well (e.g., we miss the small peaks) (Figure 3, third row). With a well-chosen regularization parameter (here, $\lambda = 10^{-3}$), the DNN approximation is close to the true `peaks` function, but tuning this regularization parameter can be costly (Figure 3, second row). In comparison, the DNN approximations when automatically choosing a regularization parameter using the sGCV method are good approximations and look similar, no matter the initial regularization parameter or memory depth (Figure 3, first column set of figures).

The selected regularization parameters are related to the ill-posedness of the problem, as illustrated for the $\lambda = 10^{-10}$ case in Figure 4. When the batch size is $|\mathcal{T}_k| = 1$ (Figure 4, first column), the linear least-squares problem is underdetermined for memory depths $r = 0$ and $r = 5$ and is overdetermined when $r = 10$. To avoid overfitting in the underdetermined cases, larger regularization parameters are selected. In the overdetermined case, overfitting is less likely and thus less regularization is needed.

With an adequate choice of memory depth and batch size, training a DNN with `slimTrain` decreases the training loss and generalizes well to unseen data. The choice of regularization parameter significantly impacts the resulting network: too much regularization and the training stagnates; too little regularization and the training oscillates. Employing adaptive regularization parameter selection mitigates these extremes and simplifies the costly a priori step of tuning the parameter.

TABLE 1

Training and validation loss in the CDR experiment for batch size $|\mathcal{T}_k| = 5, 10$ and learning rates $\gamma = 10^{-3}, 10^{-2}, 10^{-1}$. We display the loss after the first 20 epochs to compare early performance. Because the memory depth does not significantly impact convergence, we display the loss for `slimTrain` *with a memory depth of $r = 0$. Closeness between the training and validation losses indicates good generalization. The best overall performance (lowest loss) is achieved by* `slimTrain` *with a batch size of $|\mathcal{T}_k| = 10$, denoted in bold.*

		$\gamma = 10^{-3}$		$\gamma = 10^{-2}$		$\gamma = 10^{-1}$	
		Train	Valid	Train	Valid	Train	Valid
$\lvert\mathcal{T}_k\rvert = 5$	`slimTrain`, $r = 0$	42.98	41.17	22.06	22.25	18.74	23.25
	ADAM	1453.00	1338.00	45.24	42.73	8.07	8.70
$\lvert\mathcal{T}_k\rvert = 10$	`slimTrain`, $r = 0$	47.65	52.95	**4.28**	**5.30**	15.61	16.60
	ADAM	4405.00	4143	49.92	41.23	10.67	10.71

4.3. PDE surrogate modeling. Due to their approximation properties, there has been increasing interest in using DNNs as efficient surrogate models for computationally expensive tasks arising in scientific applications. One common task is PDE surrogate modeling in which a DNN replaces expensive linear system solves [37, 4, 56, 49]. Here, we consider a parameterized PDE

$$\mathbf{c} = \mathcal{P}u, \quad \text{where} \quad \mathcal{A}(u, \mathbf{y}) = 0, \tag{4.1}$$

where u is the solution to a PDE defined by $\mathcal{A}$ and parameterized by $\mathbf{y}$ (which could be discrete or continuous). In our case, the solution is measured at discrete points given by the linear operator $\mathcal{P}$ and the observations are contained in $\mathbf{c}$. The goal is to train a DNN as a surrogate mapping from parameters $\mathbf{y}$ to observables $\mathbf{c}$ and avoid costly PDE solves.

In our experiment, we consider the convection diffusion reaction (CDR) equation which models physical phenomena in many fields including climate modeling [48] and mathematical biology [17, 8]. As its name suggests, the CDR equation is composed of three terms: a diffusion term that encourages an even distribution of the solution u (e.g., chemical concentration), a convection (or advection) term that describes how the flow (e.g., of the fluid containing the chemical) moves the concentration, and a reaction term that captures external factors that affect the concentration levels. In our example, the reaction term is a linear combination of 55 different reaction functions and the parameters $\mathbf{y} \in \mathbb{R}^{55}$ are the coefficients. The observables $\mathbf{c} \in \mathbb{R}^{72}$ are measured at the same six spatial coordinates and 12 different time points; for details, see [34]. We train a ResNet with a width of $w = 16$ and a depth of $d = 8$ corresponding to a final time of $T = 4$; see Appendix B for further details. The linear weights in the final, separable layer are stored as a matrix $\mathbf{W} \in \mathbb{R}^{72\times 17}$, where the number of columns is the width of the ResNet plus an additive bias. The results of training the ResNet with `slimTrain` are displayed in Figure 5. The major takeaway is that `slimTrain` exploits the separable structure of the ResNet and, as a result, trains the network faster and fits the observed data better (lower loss) than ADAM with the recommended learning rate ($\gamma = 10^{-3}$).

In Table 1, we examine whether the performance of `slimTrain` and ADAM generalizes to unseen after 20 epochs; we choose 20 epochs to analyze early performance and because the training loss decreases more slowly after 20 epochs in Figure 5. The training and validation losses are close for both `slimTrain` and ADAM, indicating

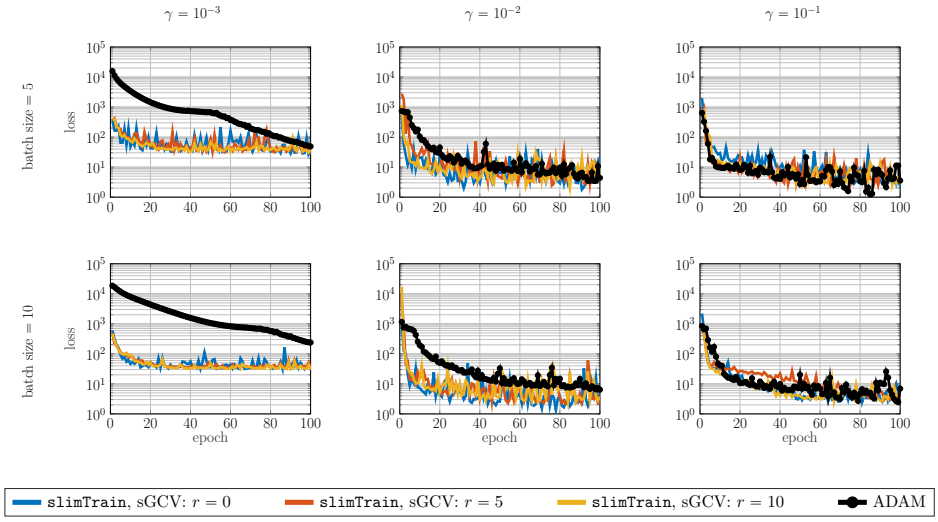


FIG. 5. *Convergence results for the training loss (mean squared error without regularization) for the CDR experiment. The rows correspond to different batch sizes, $|\mathcal{T}_k| = 5, 10$, and the columns correspond to different learning rates, $\gamma = 10^{-3}, 10^{-2}, 10^{-1}$. The colorful, solid lines depict the convergence of the training loss using* `slimTrain` *with sGCV regularization parameter selection. Each color corresponds to a different memory depth, $r = 0, 5, 10$. The black line with markers depicts the convergence of the training loss using ADAM. (Figure in color online.)*

that both training algorithms produce networks that generalize well. For ADAM's suggested learning rate, $\gamma = 10^{-3}$, `slimTrain` achieves a validation loss that is two orders of magnitude less than that of ADAM. When the learning rate is tuned to $\gamma = 10^{-1}$, the performance of ADAM improves, but the overall best performance is achieved by `slimTrain`. Most significantly, the performance of `slimTrain` is less sensitive to the choice of learning rate.

As with the numerical experiment in subsection 4.2, there is a relationship between batch size, memory depth, and the number of output features. In this experiment, because $\mathbf{W} \in \mathbb{R}^{72\times 17}$, we solve 72 independent least-squares problems with 17 unknowns in each problem. Illustrated in Figure 6, when the memory depth is small ($r = 0, 5$), each least-squares problem is underdetermined or not sufficiently overdetermined, and hence more regularization on $\mathbf{W}$ is needed to avoid overfitting. Because we use sGCV to automatically select the regularization parameter, the training with `slimTrain` achieves a comparable loss for all memory depths. In addition, the learning rate to update $\boldsymbol{\theta}$ plays a role in the regularization parameters chosen. When the learning rate is large ($\gamma = 10^{-1}$), the output features of the network can change rapidly. As a result, larger regularization parameters are selected, even in the sufficiently overdetermined case ($r = 10$), to avoid fitting features that will change significantly at the next iteration.

In this surrogate modeling example, `slimTrain` converges faster to the same or a better accuracy than ADAM using the recommended learning rate ($\gamma = 10^{-3}$) by exploiting the separability of the DNN architecture. Tuning the learning rate can improve the results for ADAM, but training with `slimTrain` produces comparable results and reaches a desirable loss in the same or fewer epochs. Using sGCV to select the regularization parameter on the weights $\mathbf{W}$ provides more robust training,

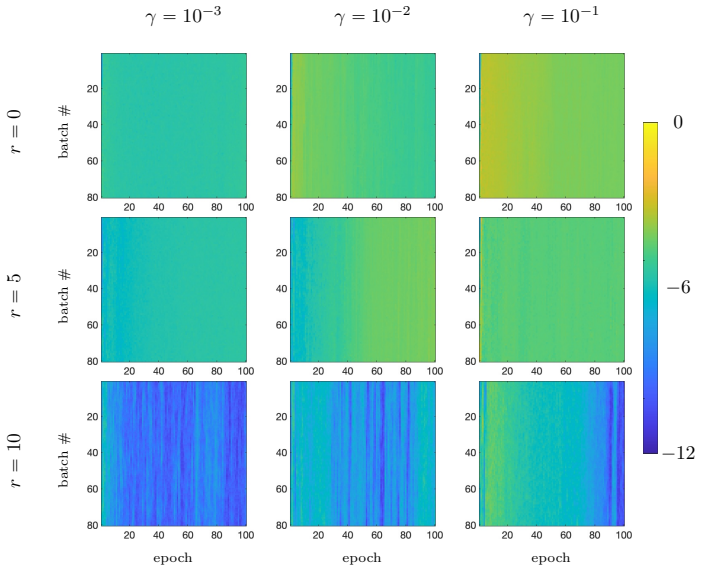


FIG. 6. *Effect of learning rate and memory depth on the choice of regularization parameters in the CDR experiment. The presented plots are from training using* `slimTrain` *for a batch size of* $|\mathcal{T}_k| = 5$. *We show the regularization parameters (in log scale) obtained for various learning rates (columns) and memory depths (rows).*

adjusting automatically to the various hyperparameters (memory depth, learning rate) to produce consistent convergence.

4.4. Autoencoders. Autoencoders are a dimensionality-reduction technique using two neural networks: an encoder that represents high-dimensional data in a low-dimensional space and a decoder that reconstructs the high-dimensional data from this encoding, illustrated in Figure 7. Training an autoencoder is an unsupervised learning problem that can be phrased as an optimization problem,

$$\min_{\mathbf{w},\boldsymbol{\theta}_{\text{dec}},\boldsymbol{\theta}_{\text{enc}}} \Phi_{\text{auto}}(\mathbf{w},\boldsymbol{\theta}_{\text{dec}},\boldsymbol{\theta}_{\text{enc}}) \equiv \mathbb{E}\ \tfrac{1}{2}\|\mathbf{K}(\mathbf{w})F_{\text{dec}}(F_{\text{enc}}(\mathbf{y},\boldsymbol{\theta}_{\text{enc}}),\boldsymbol{\theta}_{\text{dec}}) - \mathbf{y}\|_2^2 + \tfrac{\alpha_{\text{enc}}}{2}\|\boldsymbol{\theta}_{\text{enc}}\|_2^2 + \tfrac{\alpha_{\text{dec}}}{2}\|\boldsymbol{\theta}_{\text{dec}}\|_2^2 + \tfrac{\lambda}{2}\|\mathbf{w}\|_2^2, \tag{4.2}$$

where the components of the objective function are the following:

- **Encoder:** $F_{\text{enc}} : \mathcal{Y} \times \mathbb{R}^{|\boldsymbol{\theta}_{\text{enc}}|} \to \mathbb{R}^{n_{\text{lat}}}$ is the encoding neural network that reduces the dimensionality of the input features n_{in} to an *intrinsic dimension* n_{lat} with $n_{\text{lat}} \ll n_{\text{in}}$. Typically, the true intrinsic dimension is not known and must be chosen manually. The weights are $\boldsymbol{\theta}_{\text{enc}} \in \mathbb{R}^{|\boldsymbol{\theta}_{\text{enc}}|}$, the number of encoder weights is $|\boldsymbol{\theta}_{\text{enc}}|$, and the regularization parameter is $\alpha_{\text{enc}} \geq 0$.
- **Decoder feature extractor:** $F_{\text{dec}} : \mathbb{R}^{n_{\text{lat}}} \times \mathbb{R}^{|\boldsymbol{\theta}_{\text{dec}}|} \to \mathbb{R}^{n_{\text{out}}}$ is the decoder feature extractor. The weights are $\boldsymbol{\theta}_{\text{dec}} \in \mathbb{R}^{|\boldsymbol{\theta}_{\text{dec}}|}$, the number of weights is $|\boldsymbol{\theta}_{\text{dec}}|$, and the regularization parameter is $\alpha_{\text{dec}} \geq 0$.
- **Decoder final layer:** $\mathbf{K}(\cdot) : \mathbb{R}^{|\mathbf{w}|} \to \mathbb{R}^{n_{\text{in}} \times n_{\text{out}}}$ is a linear operator, mapping $\mathbf{w}$ to a matrix $\mathbf{K}(\mathbf{w})$. For instance, $\mathbf{K}(\mathbf{w})$ could be a sparse convolution matrix which can be accessed via function calls. The learnable weights $\mathbf{w}$ have a regularization parameter $\lambda \geq 0$.

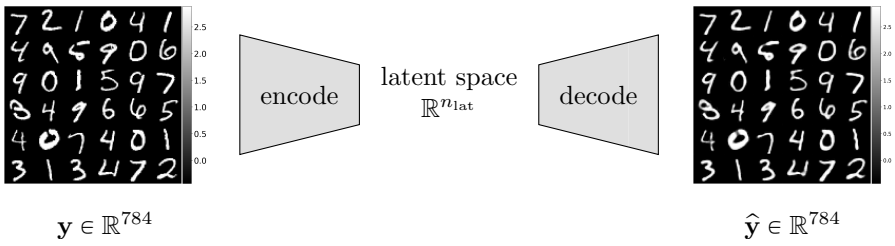


FIG. 7. *Illustration of autoencoder for the MNIST data. The goal is to represent high-dimensional data in a low-dimensional, latent space for dimensionality reduction and feature extraction* [23]. *The encoder is a neural network that maps input data* $\mathbf{y}$ *to the latent space with intrinsic dimension* n_{lat} *(typically user-defined). The decoder is a neural network that maps from the latent space to obtain an approximation of the original input,* $\widehat{\mathbf{y}}$.

For notational simplicity, we let $\boldsymbol{\theta} = (\boldsymbol{\theta}_{\text{enc}}, \boldsymbol{\theta}_{\text{dec}})$ and $\alpha = \alpha_{\text{enc}} = \alpha_{\text{dec}}$ for the remainder of this section.

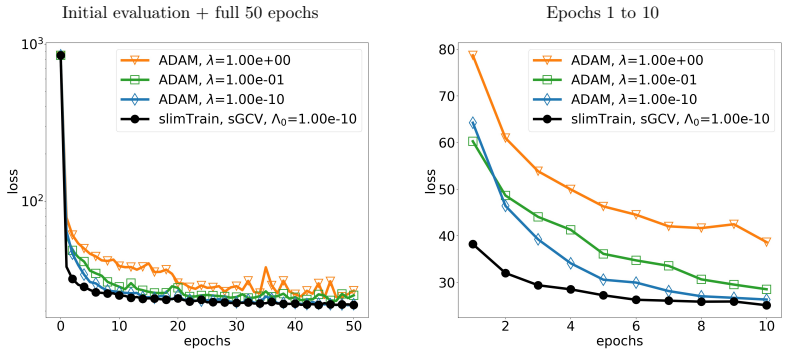
In this experiment, we train a small autoencoder on the MNIST dataset [31]. The data consists of 60,000 training and 10,000 test gray-scale images of size 28×28 (i.e., 784 input features). We implement convolutional neural networks for both the encoder and decoder with intrinsic dimension $n_{\text{lat}} = 50$; see details in Appendix C. Unlike the dense matrices in the previous experiments, the final, separable layer is a (transposed) convolution. Because convolutions use few weights and the prediction is high-dimensional, the least-squares problem is always overdetermined for this application. Hence, we require only a moderate memory depth in our experiments and, motivated by our results in subsections 4.2 and 4.3, we use a memory depth of $r = 5$ when training with `slimTrain`.

The convergence results comparing `slimTrain` and ADAM are presented in Figure 8. Here, we see that training with `slimTrain` converges faster than ADAM in the first 10 epochs and to a comparable lowest loss after 50 epochs. Each training scheme forms an autoencoder that approximates the MNIST data accurately and generalizes well, even after the first epoch. However, the absolute difference between the `slimTrain` approximation and the true test images after the first epoch is noticeably less noisy than the ADAM-trained approximations after the first epoch, particularly for a poor choice of regularization parameter on $\mathbf{w}$ (e.g., $\lambda = 10^0$). We note that because we employ automatic regularization parameter selection, the performance of `slimTrain` was nearly identical with different initial regularization parameters, Λ_0. We display the case that produced slightly less oscillatory convergence.

Using a good choice of the regularization parameter on the nonlinear weights ($\alpha = 10^{-10}$) is partially responsible for the quality approximations obtained for each training method. The results in Figure 9 support our choice of a small regularization parameter on $\boldsymbol{\theta}$. It can be seen that smaller regularization parameters on $\boldsymbol{\theta}$ produce better DNN approximations. When α is poorly chosen (in this case, when α is large), `slimTrain` produces a considerably smaller loss than training with ADAM. Hence, training with `slimTrain` and sGCV can adjust to poor hyperparameter selection, even when those hyperparameters are not directly related to the regularization on $\mathbf{w}$.

In addition to adjusting regularization parameters for the linear weights, we found

(a) Convergence of ADAM and `slimTrain`



(b) DNN approximations after one epoch

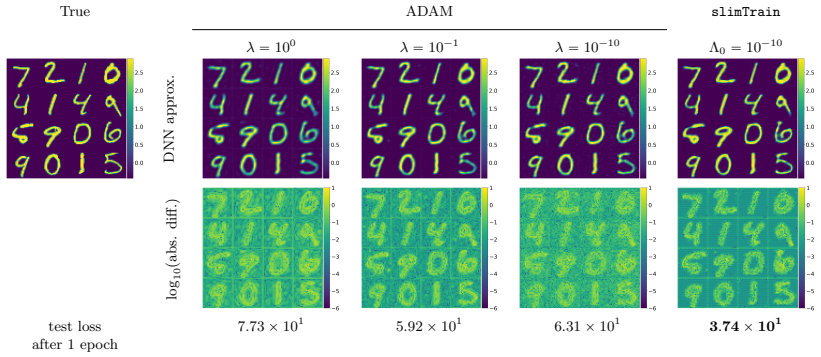


FIG. 8. *Training loss convergence and visualizations of MNIST autoencoder approximations. For the convergence plots, the networks are trained for* 50 *epochs and with recommended learning rate of* $\gamma = 10^{-3}$*, batch size of* $|\mathcal{T}_k| = 32$*, regularization parameter* $\alpha = 10^{-10}$ *for* $\boldsymbol{\theta}$*, and* $50,000$ *training images plus* $10,000$ *for validation. For ADAM, we train with three different regularization parameters for* $\mathbf{w}$*,* $\lambda = 10^0, 10^{-1}, 10^{-10}$*. When using* `slimTrain`*, we automatically select the regularization parameters using sGCV with initial parameter* $\Lambda_0 = 10^{-10}$ *and choose a modest memory depth of* $r = 5$*. We display the DNN approximations after the first epoch below the convergence plots. The top row of MNIST images are, from left to right,* 16 *test images, the approximation from the ADAM-trained networks with various regularization parameters on* $\mathbf{w}$*, and the approximation obtained from* `slimTrain`*. The bottom row oiages are the absolute differences (in log scale) between the network approximations and the true test images. The value below the absolute difference images is the test loss over all* $10,000$ *test images after the first epoch.*

that training with `slimTrain` offers significant performance benefits in the limited-data setting; see Figure 10. When only a few training samples were used, training with `slimTrain` produces a lower training and validation loss. In the small training data regime, the optimization problem is more ill-posed and there are fewer network weight updates per epoch. Hence, the automatic regularization selection and fast initial convergence of `slimTrain` produce a more effective autoencoder.

Consistent with the results in our previous experiments, in the autoencoder example with a final convolutional layer, `slimTrain` converges faster initially than ADAM to a good approximation and is less sensitive to the choice regularization on the nonlinear weights, $\boldsymbol{\theta}$. In the case of limited data, a common occurrence for scientific

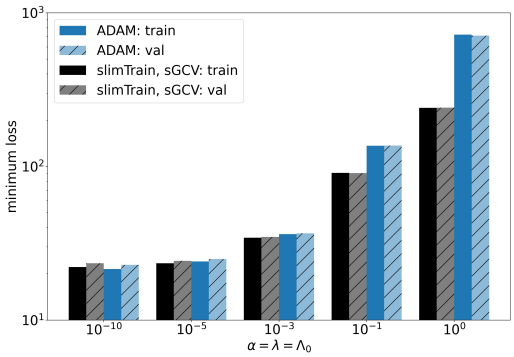


FIG. 9. *Effect of regularization parameters on the minimum loss. We vary the regularization parameter α of $\boldsymbol{\theta}$ for both ADAM (blue) and* `slimTrain` *(black), the regularization parameter λ on $\mathbf{w}$ for ADAM, and the initial regularization parameter Λ_0 on $\mathbf{w}$ for* `slimTrain`*. For simplicity, we set the (initial) regularization parameters equal, $\alpha = \lambda = \Lambda_0$. The height of each bar is the training (solid) and validation (striped) loss for the network that obtained the lowest validation loss in* 50 *epochs for the given hyperparameters. (Figure in color online.)*

applications, the training problem becomes more ill-posed. Here, `slimTrain` produces networks that fit and generalize better than ADAM. By solving for good weights $\mathbf{w}$ and automatically choosing an appropriate regularization parameter at each iteration, `slimTrain` achieves more consistent training performance for many different choices of hyperparameters.

5. Conclusions. We addressed the challenges of training DNNs by exploiting the separability inherent in most commonly used architectures whose output depends linearly on the weights of the final layer. Our proposed algorithm, `slimTrain`, leverages this separable structure for function approximation tasks where the optimal weights of the final layer can be obtained by solving a stochastic regularized linear least-squares problem. The main idea of `slimTrain` is to iteratively estimate the weights of the final layer using the sampled limited-memory Tikhonov scheme `slimTik` [13], which is a state-of-the-art method to solve stochastic linear least-squares problems. By using `slimTik` to update the linear weights, `slimTrain` provides a reasonable approximation for the optimal linear weights and simultaneously estimates an effective regularization parameter for the linear weights. The latter point is crucial—`slimTrain` does not require a difficult-to-tune learning rate and automatically adapts the regularization parameter for the linear weights, which can simplify the training process. In our numerical experiments, `slimTrain` is less sensitive to the choice of hyperparameters, which can make it a good candidate to train DNNs for new datasets with limited experience and no clear hyperparameter selection guidelines.

From a theoretical perspective, `slimTrain` can be seen as an inexact version of the variable projection [20, 35] (VarPro) scheme extended to the stochastic approximation (SA) setting. Using this viewpoint, we show in subsection 3.2 that we obtain unbiased gradient estimates for the nonlinear weights when the linear weights are estimated accurately. This motivates the design of `slimTrain` as a tractable alternative to VarPro SA, which is infeasible as it requires re-evaluation of the nonlinear feature

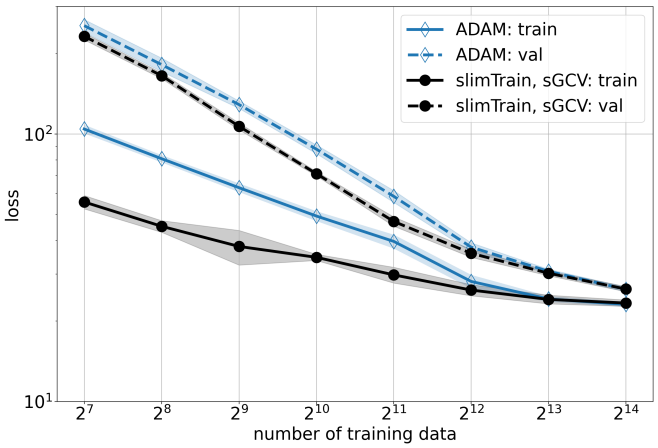


FIG. 10. *Mean loss of MNIST autoencoder for small number of training data with a batch size of* 32. *All networks were trained for* 50 *epochs for* 10 *different weight initializations using the same hyperparameters as with the full training data in Figure* 8. *For each initialization, we choose the network that produced the minimal validation loss over* 50 *epochs. In the plot, each point denotes the mean loss over the* 10 *runs and the bands depict one standard deviation from the mean.*

extractor over many samples after every training step. The computational costs of `slimTrain` are limited as it reuses features from the most recent batches and therefore adds little computational overhead; see subsection 4.1. In addition, `slimTrain` approximates the optimal linear weights obtained from VarPro, thereby reducing the bias introduced by the approximation when updating the nonlinear weights.

From a numerical perspective, the benefits of `slimTrain`, and specifically automated hyperparameter selection, are demonstrated by the numerical experiments for both fully connected and convolutional final layers. In subsection 4.2, we explore the relationship of the `slimTrain` parameters, observing that memory depth and batch size play a crucial role in determining the ill-posedness of the least-squares problem to solve for the linear weights. The regularization parameter adapts to the least-squares problem accordingly—larger regularization parameters are selected when the problem is underdetermined. In subsection 4.3, we observe that `slimTrain` is less sensitive to the choice of learning rate, outperforming the recommended settings for ADAM. Again, the regularization parameters adapt to the learning rate—larger parameters are chosen when the nonlinear weights change more rapidly. In subsection 4.4, we show that `slimTrain` can be applied to a final convolutional layer and outperforms ADAM in the limited-data regime, which is typical in scientific applications.

Appendix A. Stochastic linear Tikhonov problem. In this section, we show that, under certain assumptions, the stochastic Tikhonov-regularized least-squares problem (2.6) has a closed form solution (2.7). Let us begin by defining $\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta}) = \mathbb{E}F(\mathbf{y}, \boldsymbol{\theta})$, $\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta}) = \mathbb{E}(F(\mathbf{y}, \boldsymbol{\theta}) - \boldsymbol{\mu}_{\mathbf{y}})(F(\mathbf{y}, \boldsymbol{\theta}) - \boldsymbol{\mu}_{\mathbf{y}})^\top$ and $\boldsymbol{\mu}_{\mathbf{c}}(\boldsymbol{\theta}) = \mathbb{E}\mathbf{c}$, $\boldsymbol{\Sigma}_{\mathbf{c}} = \mathbb{E}(\mathbf{c} -$

$\boldsymbol{\mu}_{\mathbf{c}})(\mathbf{c}-\boldsymbol{\mu}_{\mathbf{c}})^\top$. Then using the identity

$$\mathbb{E}(\boldsymbol{\delta}^\top \boldsymbol{\Lambda}\boldsymbol{\delta}) = \mathrm{tr}(\boldsymbol{\Lambda}\boldsymbol{\Sigma}_{\boldsymbol{\delta}}) + \boldsymbol{\mu}_{\boldsymbol{\delta}}^\top \boldsymbol{\Lambda}\boldsymbol{\mu}_{\boldsymbol{\delta}},$$

where $\mathrm{tr}(\,\cdot\,)$ denotes the trace of a matrix, we have (sans constant from the regularization term for $\boldsymbol{\theta}$)

$$\Phi(\mathbf{W},\boldsymbol{\theta}) = \mathbb{E}\ \tfrac{1}{2}\left\|\mathbf{W}F(\mathbf{y},\boldsymbol{\theta})-\mathbf{c}\right\|_2^2 + \tfrac{\lambda}{2}\left\|\mathbf{W}\right\|_{\mathrm{F}}^2 \tag{A.1}$$

$$= \mathbb{E}\ \tfrac{1}{2}\left(\mathbf{W}F(\mathbf{y},\boldsymbol{\theta})-\mathbf{c}\right)^\top\left(\mathbf{W}F(\mathbf{y},\boldsymbol{\theta})-\mathbf{c}\right) + \tfrac{\lambda}{2}\left\|\mathbf{W}\right\|_{\mathrm{F}}^2 \tag{A.2}$$

$$= \mathbb{E}\ \tfrac{1}{2}F(\mathbf{y},\boldsymbol{\theta})^\top\mathbf{W}^\top\mathbf{W}F(\mathbf{y},\boldsymbol{\theta}) - \mathbb{E}\mathbf{c}^\top\mathbf{W}F(\mathbf{y},\boldsymbol{\theta}) + \tfrac{1}{2}\mathbb{E}\mathbf{c}^\top\mathbf{c} + \tfrac{\lambda}{2}\left\|\mathbf{W}\right\|_{\mathrm{F}}^2 \tag{A.3}$$

$$= \tfrac{1}{2}\mathrm{tr}\big(\mathbf{W}^\top\mathbf{W}\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta})\big) + \tfrac{1}{2}\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})^\top\mathbf{W}^\top\mathbf{W}\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta}) - \mathbb{E}\mathbf{c}^\top\mathbf{W}F(\mathbf{y},\boldsymbol{\theta}) \tag{A.4}$$

$$+ \tfrac{1}{2}\mathrm{tr}(\boldsymbol{\Sigma}_{\mathbf{c}}) + \tfrac{1}{2}\boldsymbol{\mu}_{\mathbf{c}}^\top\boldsymbol{\mu}_{\mathbf{c}} + \tfrac{\lambda}{2}\left\|\mathbf{W}\right\|_{\mathrm{F}}^2. \tag{A.5}$$

Notice that this function is quadratic in $\mathbf{W}$, and so for a given $\boldsymbol{\theta}$ a minimizer (2.6) can be found by differentiation. That is,

$$\mathrm{D}_{\mathbf{W}}\Phi(\mathbf{W},\boldsymbol{\theta}) = \mathbf{W}\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta}) + \mathbf{W}\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})^\top + \lambda\mathbf{W} - \mathbb{E}\mathbf{c}\mathrm{F}(\mathbf{y},\boldsymbol{\theta})^\top \tag{A.6}$$

assuming we can switch order $\mathrm{D}\mathbb{E} = \mathbb{E}\mathrm{D}$. Now setting $\mathrm{D}_{\mathbf{W}}\Phi = \mathbf{0}$, we get

$$\widehat{\mathbf{W}}(\boldsymbol{\theta})\left(\boldsymbol{\Sigma}_{\mathbf{y}}(\boldsymbol{\theta}) + \boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})\boldsymbol{\mu}_{\mathbf{y}}(\boldsymbol{\theta})^\top + \lambda\mathbf{I}\right) = \mathbb{E}\mathbf{c}F(\mathbf{y},\boldsymbol{\theta})^\top \tag{A.7}$$

and hence (2.7).

Appendix B. Residual neural networks (ResNets). Residual neural networks (ResNets), among the most popular DNN architectures, are composed of layers of the form

$$\mathbf{u}_0 = \sigma(\mathbf{K}_{\mathrm{in}}\mathbf{y} + \mathbf{b}_{\mathrm{in}}), \tag{B.1}$$

$$\mathbf{u}_{j+1} = \mathbf{u}_j + h\sigma(\mathbf{K}_j\mathbf{u}_j + \mathbf{b}_j) \text{ for } j = 0,\dots,d-1. \tag{B.2}$$

The architecture is defined by the width (the number of entries in the feature vectors $\mathbf{u}_j$), the depth (the number of layers d), and the step size $h > 0$. The key property of ResNets is the identity mapping or skip connection which enables deeper, more expressive networks to be trained [27]. Recent work has interpreted ResNets as discretizations of continuous differential equations or dynamical systems [19] which have led to notions of stability [25], PDE-inspired architectures [45], and neural ODEs [10].

In subsection 4.2, we train a DNN to map $\mathbf{y} \in \mathbb{R}^2$ to a scalar $c \in \mathbb{R}$. The feature extractor is a ResNet with a width of $w = 8$ and a depth of $d = 8$ corresponding to a final time of $T = 5$ or equivalently with a step size of $h = 5/8$. In subsection 4.3, we train a DNN to map $\mathbf{y} \in \mathbb{R}^{55}$ to a scalar $c \in \mathbb{R}^{72}$. The feature extractor is a ResNet with a width of $w = 8$ and a depth of $d = 8$ corresponding to a final time of $T = 5$ or equivalently with a step size of $h = 5/8$. In both experiments, we use the smooth hyperbolic tangent activation function, $\sigma(x) = \tanh(x)$.

Appendix C. Autoencoder architecture. We adapt the MNIST autoencoder from [32]. The autoencoder consists of two convolutional neural networks with a user-defined width w and intrinsic dimension d. The width controls the number of convolutional filters used and the intrinsic dimension is the size of the low-dimensional embedding. The architecture is described in Table 2.

The final layer is a (transposed) convolution, denoted in subsection 4.1 as $\mathbf{K}(\cdot) : \mathbb{R}^{|\mathbf{w}|} \to \mathbb{R}^{n_{\mathrm{in}} \times n_{\mathrm{out}}}$. Note that $n_{\mathrm{in}} > n_{\mathrm{out}}$ in our case. As we did in subsection 3.1,

TABLE 2

Autoencoder architecture with a width $w = 16$ and intrinsic dimension d. For the convolutional layers, s is the stride and p is the padding. The layer ConvT indicates a transpose convolution. The dashed line indicates the separable final layer.

	Layer type	Description	# Feat. out	# Weights	$w = 16$, $d = 50$
Enc	Conv. + ReLU	w, $4 \times 4 \times 1$ filters, $s = 2$, $p = 1$	$14 \times 14 \times w$	$16w + w$	272
	Conv. + ReLU	$2w$, $4 \times 4 \times w$ filters, $s = 2$, $p = 1$	$7 \times 7 \times 2w$	$32w^2 + 2w$	8,224
	Affine	$d \times (49 \cdot 2w)$ matrix + $d \times 1$ bias	$d \times 1$	$98wd + d$	78,450
Dec	Affine	$(49 \cdot 2w) \times d$ matrix + $(49 \cdot 2w) \times 1$ bias	$98w \times 1$	$98wd + 98w$	79,968
	Batch norm	—	—	—	—
	ConvT. + ReLU	w, $4 \times 4 \times 2w$ filters, $s = 2$, $p = 1$	$14 \times 14 \times w$	$32w^2 + w$	8,208
	ConvT. + ReLU	1, $4 \times 4 \times w$ filter, $s = 2$, $p = 1$	$28 \times 28 \times 1$	$16w + 1$	257
			Total	—	$175,122 + 257$

we can express the operation of $\mathbf{K}(\mathbf{w}) \in \mathbb{R}^{n_{\text{in}} \times n_{\text{out}}}$ on the output features $\mathbf{Z}_k(\boldsymbol{\theta}) \in \mathbb{R}^{n_{\text{out}} \times |\mathcal{T}_k|}$ as a linear operator applied to the weights $\mathbf{w}$; that is,

$$\mathbf{K}(\mathbf{w})\mathbf{Z}_k(\boldsymbol{\theta}) \qquad \underset{\text{conv}}{\overset{\text{de-conv}}{\rightleftarrows}} \qquad \mathbf{A}_k(\boldsymbol{\theta})\mathbf{w}.$$

The matrix $\mathbf{A}_k(\boldsymbol{\theta}) \in \mathbb{R}^{|\mathcal{T}_k| n_{\text{in}} \times |\mathbf{w}|}$ has known structure. In particular, each column of $\mathbf{A}(\boldsymbol{\theta})$ contains a shifted copy of $\text{vec}(\mathbf{Z}_k(\boldsymbol{\theta}))$. Naïvely, we can form each column of $\mathbf{A}_k(\boldsymbol{\theta})$ explicitly by applying the (transposed) convolution operator to "standard basis" filters. Specifically, the jth column of $\mathbf{A}_k(\boldsymbol{\theta})$ is $\text{vec}(\mathbf{K}(\mathbf{e}_j)\mathbf{Z}_k(\boldsymbol{\theta}))$, where $\mathbf{e}_j \in \mathbb{R}^{|\mathbf{w}|}$ is the jth unit vector. In our implementation, we construct $\mathbf{A}_k(\boldsymbol{\theta})$ by recognizing that the samples and the channels of $\mathbf{Z}_k(\boldsymbol{\theta})$ are independent, requiring fewer evaluations of the (transposed) convolution operator.

Acknowledgments. This work was initiated as a part of the SAMSI Program on Numerical Analysis in Data Science in 2020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] A. AGARWAL, P. L. BARTLETT, P. RAVIKUMAR, AND M. J. WAINWRIGHT, *Information-theoretic lower bounds on the oracle complexity of stochastic convex optimization*, IEEE Trans. Inform. Theory, 58 (2012), pp. 3235–3249.

[2] A. BECK AND L. TETRUASHVILI, *On the convergence of block coordinate descent type methods*, SIAM J. Optim., 23 (2013), pp. 2037–2060, https://doi.org/10.1137/120887679.

[3] J. C. BEZDEK AND R. J. HATHAWAY, *Some notes on alternating optimization*, in AFSS International Conference on Fuzzy Systems, Springer, Berlin, Heidelberg, 2002, pp. 288–300.

[4] K. BHATTACHARYA, B. HOSSEINI, N. B. KOVACHKI, AND A. M. STUART, *Model reduction and neural networks for parametric PDEs*, SMAI J. Comput. Math., 7 (2021), pp. 121–157.

[5] R. BOLLAPRAGADA, R. H. BYRD, AND J. NOCEDAL, *Exact and inexact subsampled Newton methods for optimization*, IMA J. Numer. Anal., 39 (2018), pp. 545–578, https://doi.org/10.1093/imanum/dry009.

[6] L. BOTTOU AND Y. CUN, *Large scale online learning*, in Advances in Neural Information Processing Systems, 2004, pp. 217–224.

[7] L. BOTTOU, F. E. CURTIS, AND J. NOCEDAL, *Optimization methods for large-scale machine learning*, SIAM Rev., 60 (2018), pp. 223–311, https://doi.org/10.1137/16M1080173.

[8] N. BRITTON, *Reaction-diffusion Equations and Their Applications to Biology*, Academic Press, London, 1986.

[9] R. BYRD, S. HANSEN, J. NOCEDAL, AND Y. SINGER, *A stochastic quasi-Newton method for large-scale optimization*, SIAM J. Optim., 26 (2016), pp. 1008–1031, https://doi.org/10.1137/140954362.

[10] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural ordinary differential equations*, in Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 6572–6583.
[11] J. Chung, M. Chung, and J. T. Slagel, *Iterative sampled methods for massive and separable nonlinear inverse problems*, in Proceedings of the International Conference on Scale Space and Variational Methods in Computer Vision, Springer, 2019, pp. 119–130.
[12] J. Chung, M. Chung, J. T. Slagel, and L. Tenorio, *Stochastic Newton and Quasi-Newton Methods for Large Linear Least-squares Problems*, preprint, https://arxiv.org/abs/1702.07367, 2017.
[13] J. Chung, M. Chung, J. T. Slagel, and L. Tenorio, *Sampled limited memory methods for massive linear inverse problems*, Inverse Problems, 36 (2020), 054001.
[14] J. Chung and J. G. Nagy, *An efficient iterative approach for large-scale separable nonlinear inverse problems*, SIAM J. Sci. Comput., 31 (2010), pp. 4654–4674, https://doi.org/10.1137/080732213.
[15] G. Cybenko, *Approximations by superpositions of a sigmoidal function*, Math. Control Signals Systems, 2 (1989), pp. 303–314.
[16] E. C. Cyr, M. A. Gulian, R. G. Patel, M. Perego, and N. A. Trask, *Robust training and initialization of deep neural networks: An adaptive basis viewpoint*, in Mathematical and Scientific Machine Learning, PMLR, 2020, pp. 512–536.
[17] G. de Vries, T. Hillen, M. Lewis, J. Müller, and B. Schönfisch, *A Course in Mathematical Biology*, SIAM, Philadelphia, 2006, https://doi.org/10.1137/1.9780898718256.
[18] J. Duchi, E. Hazan, and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization*, J. Mach. Learn. Res., 12 (2011), pp. 2121–2159.
[19] W. E, *A proposal on machine learning via dynamical systems*, Commun. Math. Stat., 5 (2017), pp. 1–11, https://doi.org/10.1007/s40304-017-0103-z.
[20] G. Golub and V. Pereyra, *The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate*, SIAM J. Numer. Anal., 10 (1973), pp. 413–432, https://doi.org/10.1137/0710036.
[21] G. Golub and V. Pereyra, *Separable nonlinear least squares: The variable projection method and its applications*, Inverse Problems, 19 (2003), pp. R1–R26.
[22] G. H. Golub, M. Heath, and G. Wahba, *Generalized cross-validation as a method for choosing a good ridge parameter*, Technometrics, 21 (1979), pp. 215–223, https://doi.org/10.1080/00401706.1979.10489751.
[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, MA, 2016.
[24] R. Gower and P. Richtárik, *Randomized quasi-Newton updates are linearly convergent matrix inversion algorithms*, SIAM J. Matrix Anal. Appl., 38 (2017), pp. 1380–1409, https://doi.org/10.1137/16M1062053.
[25] E. Haber and L. Ruthotto, *Stable architectures for deep neural networks*, Inverse Problems, 34 (2017), 014004.
[26] P. C. Hansen, *Rank-Deficient and Discrete Ill-Posed Problems*, SIAM, Philadelphia, 1998, https://doi.org/10.1137/1.9780898719697.
[27] 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, 2016, pp. 770–778.
[28] L. Hui and M. Belkin, *Evaluation of neural architectures trained with square loss vs. cross-entropy in classification tasks*, in International Conference on Learning Representations, 2020.
[29] D. P. Kingma and J. Ba, *ADAM: A Method for Stochastic Optimization*, preprint, https://arxiv.org/abs/1412.6980, 2014.
[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in Advances in Neural Information Processing Systems 25, Curran Associates, Red Hook, NY, 2012.
[31] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, *Handwritten digit recognition with a back-propagation network*, in Advances in Neural Information Processing Systems 2, 1990, pp. 396–404.
[32] J. Malmaud and L. White, *Tensorflow.jl: An idiomatic Julia front end for TensorFlow*, Journal of Open Source Software, 3 (2018), 1002, https://doi.org/10.21105/joss.01002.
[33] Y. Nesterov, *Efficiency of coordinate descent methods on huge-scale optimization problems*, SIAM J. Optim., 22 (2012), pp. 341–362, https://doi.org/10.1137/100802001.

[34] E. Newman, L. Ruthotto, J. Hart, and B. van Bloemen Waanders, *Train Like a (Var)Pro: Efficient Training of Neural Networks with Variable Projection*, preprint, https://arxiv.org/abs/2007.13171, 2020.

[35] D. P. O'Leary and B. W. Rust, *Variable projection for nonlinear least squares problems*, Comput. Optim. Appl., 54 (2013), pp. 579–593.

[36] T. O'Leary-Roseberry, N. Alger, and O. Ghattas, *Inexact Newton Methods for Stochastic Nonconvex Optimization with Applications to Neural Network Training*, preprint, https://arxiv.org/abs/1905.06738, 2019.

[37] T. O'Leary-Roseberry, U. Villa, P. Chen, and O. Ghattas, *Derivative-Informed Projected Neural Networks for High-Dimensional Parametric Maps Governed by PDEs*, preprint, https://arxiv.org/abs/2011.15110, 2021.

[38] C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, 8 (1982), pp. 43–71.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An imperative style, high-performance deep learning library*, in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds., Curran Associates, Red Hook, NY, 2019, pp. 8024–8035, http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[40] R. G. Patel, N. A. Trask, M. A. Gulian, and E. C. Cyr, *A Block Coordinate Descent Optimizer for Classification Problems Exploiting Convexity*, preprint, https://arxiv.org/abs/2006.10123, 2020.

[41] M. Raissi, P. Perdikaris, and G. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, J. Comput. Phys., 378 (2019), pp. 686–707.

[42] P. Richtárik and M. Takáč, *Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function*, Math. Program., 144 (2014), pp. 1–38.

[43] H. Robbins and S. Monro, *A stochastic approximation method*, Ann. Math. Statistics, 22 (1951), pp. 400–407.

[44] O. Ronneberger, P. Fischer, and T. Brox, *U-net: Convolutional networks for biomedical image segmentation*, in Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2015, pp. 234–241.

[45] L. Ruthotto and E. Haber, *Deep neural networks motivated by partial differential equations*, J. Math. Imaging Vis., 62 (2019), pp. 352–364, https://doi.org/10.1007/s10851-019-00903-1.

[46] J. Sjöberg and M. Viberg, *Separable non-linear least-squares minimization-possible improvements for neural net fitting*, in Proceedings of the 1997 IEEE Signal Processing Society Workshop on Neural Networks for Signal Processing VII, 1997, pp. 345–354.

[47] J. T. Slagel, J. Chung, M. Chung, D. Kozak, and L. Tenorio, *Sampled Tikhonov regularization for large linear inverse problems*, Inverse Problems, 35 (2019), 114008.

[48] T. Stocker, *Introduction to Climate Modelling*, Advances in Geophysical and Environmental Mechanics and Mathematics, Springer, Berlin, Heidelberg, 2011, https://doi.org/10.1007/978-3-642-00773-6.

[49] R. K. Tripathy and I. Bilionis, *Deep UQ: Learning deep neural network surrogate models for high dimensional uncertainty quantification*, J. Comput. Phys., 375 (2018), pp. 565–588.

[50] C. R. Vogel, *Computational Methods for Inverse Problems*, SIAM, Philadelphia, 2002, https://doi.org/10.1137/1.9780898717570.

[51] G. Wahba, *Practical approximate solutions to linear operator equations when the data are noisy*, SIAM J. Numer. Anal., 14 (1977), pp. 651–667, https://doi.org/10.1137/0714044.

[52] X. Wang, S. Ma, D. Goldfarb, and W. Liu, *Stochastic quasi-Newton methods for nonconvex stochastic optimization*, SIAM J. Optim., 27 (2017), pp. 927–956, https://doi.org/10.1137/15M1053141.

[53] S. J. Wright, *Coordinate descent algorithms*, Math. Program., 151 (2015), pp. 3–34, https://doi.org/10.1007/s10107-015-0892-3.

[54] P. Xu, F. Roosta, and M. W. Mahoney, *Second-order optimization for non-convex machine learning: An empirical study*, in Proceedings of the 2020 SIAM International Conference on Data Mining (SDM), 2020, pp. 199–207, https://doi.org/10.1137/1.9781611976236.23.

[55] Y. Zhu and N. Zabaras, *Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification*, J. Comput. Phys., 366 (2018), pp. 415–447.

[56] Y. Zhu, N. Zabaras, P.-S. Koutsourelakis, and P. Perdikaris, *Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data*, J. Comput. Phys., 394 (2019), pp. 56–81, https://doi.org/10.1016/j.jcp.2019.05.024.