Neural network optimization under PDE constraints

G. Eli Jergensen [1]

[1]**Lawrence Berkeley National Labs,** GEJergensen@lbl.gov

ABSTRACT

We examine two methods of applying multiple equality constraints to neural networks influenced by dynamical systems and differential geometry. The first method can be applied directly to constrain neural network training and is shown to be equivalent to a particular choice of Lagrange multipliers, enabling use with standard backpropagation techniques. We evaluate the speed of this method in light of the known theoretical guarantees and propose a second method which trades guarantees for speed. The second method for constraining neural networks can be applied to a model post-training and is therefore also completely independent of the model design and architecture. Experimentally, we evaluate the performance and computational efficiency of these methods against both unconstrained and soft-constrained baselines on a simple toy problem which allows for detailed investigation. We primarily investigate the Helmholtz equation as a linear partial differential equation (PDE) constraint, as many constraints for scientific domains can be framed as PDEs. We show that while the outputs of the constrained models do sometimes seem qualitatively better and are less prone than soft-constraints to over-constraining the problem, all methods seem to be unpromising in practice, despite theoretical guarantees. Finally, we discuss difficulties of implementing these methods for practical problems and offer suggestions for future improvements.

Keywords: neural networks, constrained optimization, nonlinear optimization, PDE constrained optimization, PDEs

INTRODUCTION

Neural networks have shown considerable promise in recent years as a data-driven sample generation process. For example, Karras et al. (2018) were able to produce high-quality images of human faces and showed that the generation process even possessed a latent space which agrees at least partially with human intuition. In light of this success, many groups have turned to applying neural networks as a generative process for scientific domains, such as modelling fluid dynamics (Kim et al., 2018; Wiewel et al., 2018), turbulence (King et al., 2018; Mohan et al., 2019), and airflow (Thuerey et al., 2018). In contrast to naturalistic domains, such as images of human faces or animals, for scientific domains experts are often aware of many rules underlying the generative processes. For example, in modelling incompressible fluid flow, we know that conservation of momentum holds. Scientific laws and conservation and governing equations are vital to our understanding of such phenomena, but neural networks are presently incapable of taking advantage of this domain knowledge. This is problematic in primarily two regards. Firstly, because data-driven neural networks do not know the underlying equations, they require more data, as they must learn the underlying equations before they can be a truly representative data generator. Incorporating these equations into the model will hopefully allow for more detailed and skillful sample generation with smaller datasets. Secondly, models which do not explicitly respect the known equations are more difficult to trust. For data-driven generative processes which explicitly conform to given equations, we have greater trust that their predictions conform to the equations even in regions with few training examples. Thus, building governing equations into neural networks will hopefully provide us with a powerful data-driven generative process which is more skillful and trustworthy.

Many scientific laws, conservation equations, and governing equations can be written in the form of a partial differential equation (PDE). For this reason, in this work we focus on general methods for constructing neural networks which satisfy to an arbitrary PDE constraint (or set of

PDE constraints). Of course, in the case of multiple PDE constraints, we can only apply fewer constraints than the number of degrees of freedom of the neural network. However, since the network is typically over-parameterized by many thousands or millions of trainable weights, we are unlikely to reach that limit for a typical neural network. Neural networks are surprisingly well designed for use with PDEs. Because a PDE can be written in a form which takes a function and outputs a number for how well that function satisfies the PDE and because neural networks are themselves functions in the form of a computational graph, we can apply any desired PDE to a neural network as long as we know how to take derivatives. Fortunately, popular neural network frameworks such as Tensorflow (Abadi et al., 2015) and PyTorch (Paszke et al., 2017) use automatic differentiation to perform backpropagation on the neural network, so we already have the tools necessary to compute the derivatives of the neural network's computational graph (Baydin et al., 2018). As such, computing the value of a PDE constraint in a modern neural network framework is as simple as writing the PDE in the framework itself. The wide-ranging applicability of PDEs combined with the simplicity of calculating PDEs of neural networks motivates the investigation of PDE constrained neural networks.

There are several methods for constraining neural networks. Two which have previously found utility are task-redefining and "soft-constraints." Task-redefining is best described by example. If you wish to produce samples of fluid flow, then it is typical to have a neural network output the velocity of the fluid on a discrete grid. In order to ensure that the flow is incompressible (*i.e.* divergence free), rather than outputting the velocity of the fluid directly, the curl of the network outputs can first be computed and this interpreted as the fluid velocity. Since the divergence of the curl of a vector field is identically zero, this ensures the constraint is satisfied exactly. This process of rephrasing the problem into a form which exactly satisfies the desired constraints by design is very powerful, but very difficult to employ. Rephrasing the problem in such a manner requires extensive domain knowledge and becomes increasingly difficult when multiple constraints should be satisfied exactly. For these reasons, we do not investigate this method further in this work.

"Soft-constraints" are the most commonly used method for constraining neural networks. If a neural network is trying to minimize the loss function f and has constraint functions g_i such that $g_i = 0$ indicates the constraint is satisfied, then the soft-constraint method typically defines an augmented loss function

$$\mathcal{L} = f + c\sum_{i=1}^{M} g_i^2$$

and then uses this augmented loss function for training. Here, c is some penalty coefficient and the sum of the squares of the constraints could be replaced by any positive semi-definite regularization term. The primary problem with this method is that it only ensures that the minimization of the residual of the constraint function, but never that the constraint exactly reaches zero. There is no guarantee that the fully trained neural network exactly satisfies any of the constraints and further there is no guarantee that any constraint residual is smaller than a previously specified value. Despite this, this method is commonly used in practice because it does provide derivative information for the constraints and is very computationally cheap. For this reason, we use this as a comparison baseline for evaluating our two proposed methods of constraining neural networks.

Recent work has also considered the relationships between neural networks and constraints and also neural networks and PDEs. Márquez-Neila et al. (2017) recently examined hard constraints through the process of constrained neural network optimization and showed that under their formulation it was too computationally expensive to be of practical use. Similarly, Zhang and Constantinides (1992) and Platt and Barr (1988) tackled the idea of having a neural network predict its own Lagrange multipliers as a method of performing constrained optimization. Long et al. (2017) and Long et al. (2018) investigated the reverse problem of learning PDE governing equations from data. Raissi (2018) expanded on the task of learning PDE equations by having a neural network itself model an arbitrary PDE. Lastly, Chen et al. (2018) investigated building neural networks which implicitly respect certain PDEs. They note that residual neural network architecture have remarkable similarities to numerical evaluation and approximation of ODEs.

The contributions of this work are summarized as follows:

- Motivated by existing dynamical systems designed for constrained optimization, we develop a method of performing constrained optimization of neural networks under arbitrary PDE constraints, provide some theoretical guarantees, and perform an experimental analysis of its complexity.
- Further, in observation of the high computational cost of the constrained optimization method, we present the more computationally efficient method of nonlinear projection of the neural network weights, which trades theoretical guarantees for a large speed increase.
- Lastly, we experimentally evaluate these methods on a toy problem and show that, while both methods seem promising in theory, they do not seem practically feasible or useful.

CONSTRAINED OPTIMIZATION

Neural network training can be viewed as the optimization problem of finding

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \underset{x_B}{\mathbb{E}} \left[f(x_B; \theta) \right]$$

where f is the minimization function, which is usually a loss function applied to the outputs of the neural network, x_B is a minibatch of inputs to the network, Θ is the space of possible parameters and θ and θ^* are the current and optimal parameters of the network, respectively. Given this interpretation of neural network training, we consider the following constrained optimization problem:

$$\theta^* = \underset{\substack{\theta \in \Theta \\ \boldsymbol{g}(\theta) = \mathbf{0}}}{\operatorname{argmin}} \underset{x_B}{\mathbb{E}} \left[f(x_B; \theta) \right] \tag{1}$$

where $\boldsymbol{g}$ is a (possibly vector-valued) constraint function which is similarly applied to the parameters of the network. In practice, the function $\boldsymbol{g}$ is often estimated by using the same minibatch of inputs to the network as that which is used for the minimization function. In other words, we often use $\boldsymbol{g}(\theta) \approx \mathbb{E}_{x_B} \left[\boldsymbol{g}(x_B; \theta) \right]$. Estimating the constraints in this manner is typically very efficient, because the constraints can therefore be computed at the same time as the value of the minimization function. Especially in the case where the constraint function and the minimization function can be viewed as functions of the outputs of the neural network, most of the computational effort can be shared. In this case, the optimization problem becomes:

$$\begin{aligned} \theta^* = & \underset{\theta \in \Theta}{\operatorname{argmin}} \underset{x_B}{\mathbb{E}} \left[f(\hat{y}(x_B; \theta)) \right] \\ & \text{s.t. } \boldsymbol{g}(\hat{y}(x_B; \theta)) = \mathbf{0} \end{aligned}$$

where $\hat{y}(x_B; \theta)$ is the output of the neural network for the minibatch of inputs x_B and the parameters θ. A depiction of the general optimization process is shown in Figures 1.

The constraint function $\boldsymbol{g}$ itself implicitly defines a series of "level surfaces", which are the preimages of the values that the constraint function takes. One of these preimages, $\boldsymbol{g}^{-1}(\mathbf{0})$ defines the constraint manifold for our optimization problem. We observe that, assuming the constraint function is twice differentiable and the first partial derivatives of the constraint functions are linearly independent (*i.e.* the Jacobian of the constraint function is full-rank), then the set $\boldsymbol{g}^{-1}(\mathbf{0})$ does indeed form a manifold. Figure 2 offers a second depiction of the optimization problem, this time viewed from the perspective of the constraints instead of the loss function. For simplicity of illustration, we assume in Figure 2 that the constraint function is a single real-valued function instead of a set of real-valued functions or, equivalently, a vector-valued function.

As neural networks are trained iteratively, we can further view the training process itself as a dynamical system. The topic of constrained optimization as a dynamical system has previously been discussed in the literature and, in fact, a dynamical system for constrained optimization was proposed by Tanabe (1980) using the original work of Branin (1972). For the sake of completeness, we quickly summarize this method and comment on its guarantees. Let $f \colon \mathbb{R}^N \to \mathbb{R} \colon \theta \mapsto f(\theta)$ be

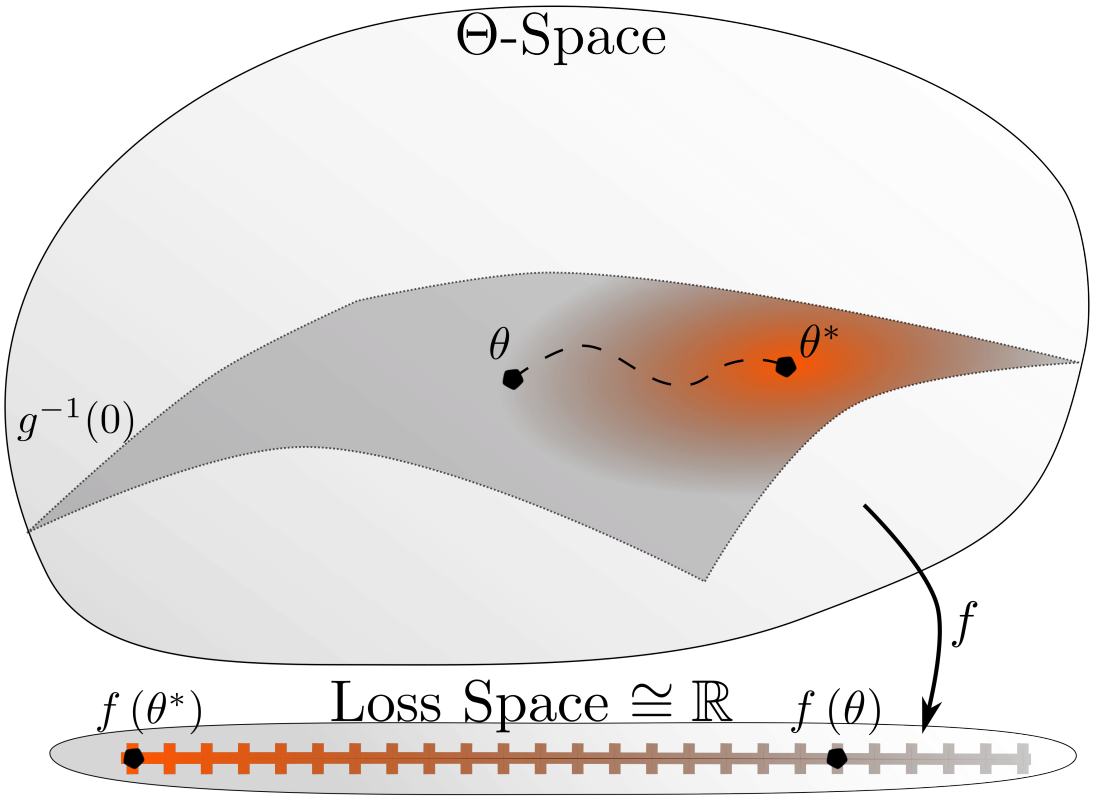

Figure 1. An abstract view of how training of a neural network to minimize a loss function f under the constraint $\boldsymbol{g}(\theta) = \mathbf{0}$ proceeds. During training, the parameters of the network move along the dashed path, thereby minimizing the minimization function. The orange color fill indicates how negative the value of the loss function is, as neural network training attempts to minimize the loss function.

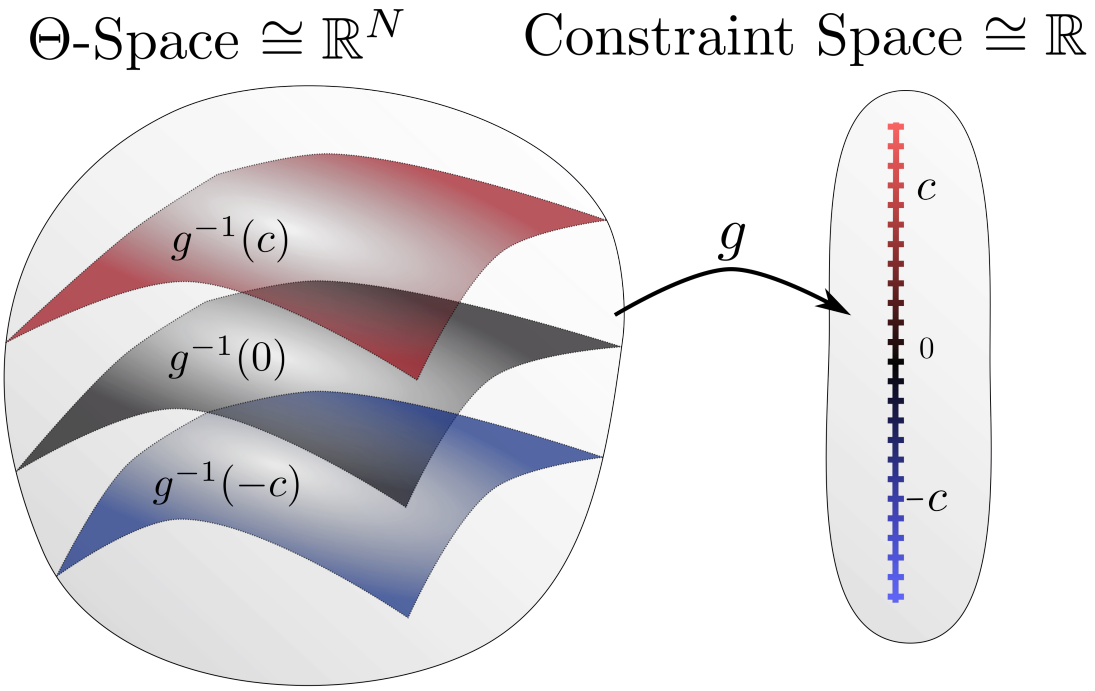

Figure 2. An abstract view of the constraint manifold and other preimages of the constraint function for an optimization problem. For simplicity of illustration, the constraint function g is assumed to be real-valued. In general, it may be vector-valued, in which case the constraint space will be isomorphic to $\mathbb{R}^M$ for some M.

a function to be minimized and similarly let $\boldsymbol{g}\colon \mathbb{R}^N \to \mathbb{R}^M \colon \theta \mapsto \boldsymbol{g}(\theta)$ be a vector of M constraint functions ($M < N$) such that $\boldsymbol{g}(\theta) = \mathbf{0}$ indicates all constraints are exactly satisfied. Further, assume that f and $\boldsymbol{g}$ are twice differentiable and that the Jacobian of the constraint function g is full-rank on a sufficiently large domain (along the trajectory of the dynamical system). Then for the constrained optimization problem,

$$\begin{aligned} \theta^* = & \underset{\theta \in \Theta \cong \mathbb{R}^N}{\operatorname{argmin}} f(\theta) \\ & \text{s.t. } \boldsymbol{g}(\theta) = \mathbf{0} \end{aligned}$$

is in the limit set of the dynamical system

$$\dot{\theta} = \frac{\mathrm{d}\theta}{\mathrm{d}t} = \Psi(\theta) = -\left(\mathrm{I} - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right) J(f(\theta))^T - J(\boldsymbol{g}(\theta))\boldsymbol{g}(\theta) \tag{2}$$

where $J(f(\theta))$ is the Jacobian matrix of the function f at the point θ and A^+ is the Moore-Penrose Pseudoinverse of the matrix A (Penrose, 1955). Here, we view the vectors θ and $\boldsymbol{g}(\theta)$ as column vectors. The requirement that the Jacobian of the constraint function g be full-rank is equivalent to requiring that linearizations of all constraints be linearly independent at a given point and implies that $J(\boldsymbol{g}(\theta))^+$ can be computed as $J(\boldsymbol{g}(\theta))^+ = J(\boldsymbol{g}(\theta))^T \left(J(\boldsymbol{g}(\theta))J(\boldsymbol{g}(\theta))^T\right)^{-1}$. With this in mind, we can rewrite the dynamical system (Eq. 2) equivalently as

$$\begin{aligned} \dot{\theta}(t) = \frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = \Psi(\theta(t)) = -J(f(\theta(t)))^T - J(\boldsymbol{g}(\theta(t))))^T \Lambda(\theta(t)), \\ \text{where} \quad \Lambda(\theta(t)) = \left(J(\boldsymbol{g}(\theta(t)))J(\boldsymbol{g}(\theta(t)))^T\right)^{-1} \left(-J(\boldsymbol{g}(\theta(t)))J(f(\theta(t)))^T + \boldsymbol{g}(\theta(t))\right) \end{aligned} \tag{3}$$

Tanabe (1980) showed that, under the assumptions above, and assuming the dynamical system does not diverge, the dynamical system would ensures $\boldsymbol{g}(t) = \boldsymbol{g}(\theta(t)) = \boldsymbol{g}(\theta(t_0)\mathrm{e}^{-t}$, where $t \in [t_0, \infty)$ (see the appendix for a proof of this). In other words, starting from the original value of $\|\boldsymbol{g}(\boldsymbol{\theta}(\boldsymbol{t_0}))\|$, the magnitude of the residual of the vector-valued constraint function would decay exponentially quickly to 0. Tanabe (1980) did note, however, that the convergence of the minimization function f to its minimum was generally much slower and is not guaranteed to be superlinear. Figure 3 describes this process geometrically. The dynamical system is equivalent to projecting the standard backpropagation vector to the tangent space of the constraint surface and adding a corrective term which ensures the exponential decay of the constraint residual. For further reference, Figures 8 and 4 of Tanabe (1980) provide similar descriptions of the dynamical system and the exponential decay of the constraint function residual, respectively.

While (Eq. 3) describes a valid dynamical system for constrained optimization, it requires some modifications before it can be used for neural networks. In particular, we need to discretize this continuous system and modify the minimization and constraint functions f and $\boldsymbol{g}$ for use with an arbitrary minibatched backpropagation scheme, such as stochastic gradient descent (SGD; LeCun et al., 2012) or ADAM (Kingma and Ba, 2014). We begin by noticing that (Eq. 3) is better suited to backpropagation, as the definition of $\Psi(\theta(t))$ can be written equivalently as

$$\Psi(\theta(t)) = -\nabla f(\theta(t)) - \sum_{i=1}^{M} \lambda_i(\theta(t)) * \nabla g_i(\theta(t))$$

where $\lambda_i(\theta(t))$ is the i^{th} element of the vector $\Lambda(\theta(t))$. Written this way, it becomes clearer that the λ_i's are acting as Lagrange multipliers. To discretize this, we apply a simple forward Euler scheme and recover that, given the learning rate η, the k^{th} iteration of the parameters of the network θ_k can be given as

$$\theta_k = \theta_{k-1} + \eta * \Psi(\theta_{k-1}) = \theta_{k-1} - \eta * \left(\nabla f(\theta_{k-1}) + \sum_{i=1}^{M} \lambda_i(\theta_{k-1}) \nabla g_i(\theta_{k-1})\right)$$

Similarly, the standard backpropagation algorithm applied to the function $\mathcal{L}$ computes

$$\theta_k = \theta_{k-1} - \eta * \nabla \mathcal{L}(\theta_{k-1})$$

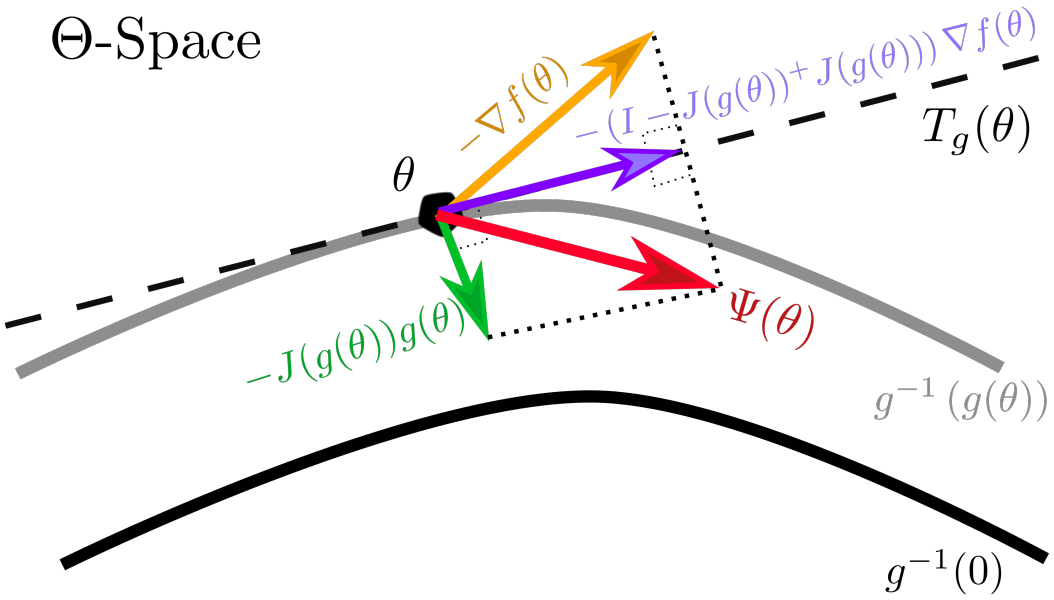

Figure 3. A geometric depiction of the dynamical system of Tanabe (1980). The gradient of the minimization function (yellow) is projected (purple) to the tangent space of the constraint function at the current parameter values θ. Additionally, a corrective term (green) ensures that any deviation from the constraint manifold is dampened exponentially quickly. This results in the total update vector $\Psi(\theta)$ (red).

Comparing these two equations shows that we can use the Lagrange multipliers λ_i together with the minimization and loss functions to obtain an equivalent $\mathcal{L}$ which satisfies the desired forward Euler scheme when the backpropagation algorithm is applied:

$$\begin{aligned} \mathcal{L}(\theta_k) &= f(\theta_k) + \sum_{i=1}^{M} \texttt{nograd}(\lambda_i(\theta_k)) * g_i(\theta_k) \\ \text{where} \quad \lambda_i(\theta_k) &= \Lambda(\theta_k)_i \\ \Lambda(\theta_k) &= \left(J(\boldsymbol{g}(\theta_k)) \cdot J(\boldsymbol{g}(\theta_k))^T\right)^{-1} \left(-J(\boldsymbol{g}(\theta_k)) \cdot J(f(\theta_k))^T + \boldsymbol{g}(\theta_k)\right) \end{aligned} \tag{4}$$

Here, the `nograd` operation ensures that the Lagrange multipliers are treated as a constant in terms of θ_k when backpropagation is performed, as otherwise we would wind up with additional terms. Equation 4 thus enables a form a constrained backpropagation for the minimization of the function f parameterized by θ such that the magnitude of the residual of the vector-valued equality constraint function $\boldsymbol{g}$, also parameterized by θ, exponentially decays to 0 (under the assumptions described above).

Finally, to ensure that this method works with neural network training with minibatches, we need to modify the functions f and $\boldsymbol{g}$ to be minibatched functions as is typical of neural network training. Ensuring that f is a minibatched function is simple, as the minibatched backpropagation algorithm already handles the minimization of an expectation of a minibatched function, $\mathbb{E}_{x_B}\left[f(x_B;\theta_k)\right]$ by using

$$\mathcal{L}(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k)\right]$$

directly. However, handling the products $\lambda_i(\theta_k) * g_i(\theta_k)$ is trickier because it is not immediately clear how one should go about taking the expectation of this. Here, we propose two slightly different versions which have slightly different interpretations. For the more direct version, which we will refer to as the fully constrained version, we compute $\mathbb{E}_{x_B}\left[\lambda_i(x_B;\theta_k) * g_i(x_B;\theta_k)\right]$. In other words, we are computing the expected value of the reweighted sum of the individual constraint functions. Geometrically, this is equivalent to computing the average tangent plane to the function $\boldsymbol{g}(\theta_k)$ in Θ-space and using that as the effective tangent plane for constraining the minimization function. Using this version yields a minibatch backpropagation scheme of

$$\begin{aligned} \mathcal{L}(x_B;\theta_k) &= \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k) + \sum_{i=1}^{M} \texttt{nograd}(\lambda_i(x_B;\theta_k)) * g_i(x_B;\theta_k)\right] \\ &= \frac{1}{B}\sum_{i=1}^{B} f(x_i;\theta_k) + \sum_{j=1}^{M} \texttt{nograd}(\lambda_j(x_i;\theta_k)) * g_j(x_i;\theta_k) \\ \text{where} \quad \lambda_j(x_i;\theta_k) &= \Lambda(x_i;\theta_k)_j \\ \Lambda(x_i;\theta_k) &= \left(J(\boldsymbol{g}(x_i;\theta_k)) \cdot J(\boldsymbol{g}(x_i;\theta_k))^T\right)^{-1} \left(-J(\boldsymbol{g}(x_i;\theta_k)) \cdot J(f(x_i;\theta_k))^T + \boldsymbol{g}(x_i;\theta_k)\right) \end{aligned} \tag{5}$$

where we are interpreting the minibatch x_B as $\{x_i : 1 \leq i \leq B\}$ and considering the functions f and $\boldsymbol{g}$ as unbatched. In practice, this can be done by performing a batched forward pass on both f and $\boldsymbol{g}$, computing the Lagrange multipliers $\Lambda(x_i;\theta_k)$ similarly using a batched function, computing $\mathcal{L}$ batched, and then taking the mean along the batch direction. We will describe how to compute the Lagrange multipliers below.

Alternatively, rather than taking the expected value of the reweighted sum of the constraint functions, we could first compute the expectation of the function $\boldsymbol{g}$ and then compute the (single set of) Lagrange multipliers for the resulting expectation. In other words, if for shorthand we define $\tilde{\boldsymbol{g}}(x_B;\theta_k) := \texttt{Reduce}_{x_i \in x_B}(\boldsymbol{g}(x_i;\theta_k))$ to be some batch-wise reduction of the function $\boldsymbol{g}$, such as the mean squared error, we could alternatively compute the products $\tilde{\lambda}_i(x_B;\theta_k) * \tilde{g}_i(x_B;\theta_k)$, where the multipliers $\tilde{\Lambda}(x_B;\theta_k)$ are computed using $\tilde{\boldsymbol{g}}$ instead of $\boldsymbol{g}$. Choosing the batch-wise reduction must be done with care, as we need to ensure that the zero values of the functions $\boldsymbol{g}$ and $\tilde{\boldsymbol{g}}$. In particular, we cannot simply take the mean of the function $\boldsymbol{g}$, as that could result in

spurious zero values. For instance, if the constraint function takes value $+1$ for half of the batch and -1 for half of the batch, then the mean value is 0, even though none of the constraints are actually satisfied. For this reason, we recommend using a standard error function, such as mean squared error, as the mean squared error is only zero if every single term is the sum is also zero. As this version hinges on the use of some batch-wise reduction function (labeled $\texttt{Reduce}_{x_i \in x_B}$ below), we refer to this version as the reduction version. The resulting minibatch backpropagation scheme is

$$\begin{aligned}
\mathcal{L}(x_B;\theta_k) &= \bar{f}(x_B;\theta_k) + \sum_{j=1}^{M} \texttt{nograd}\left(\tilde{\lambda}_j(x_B;\theta_k)\right) * \tilde{g}_j(x_B;\theta_k) \\
&= \frac{1}{B}\sum_{i=1}^{B} f(x_i;\theta_k) + \sum_{j=1}^{M} \texttt{nograd}\left(\tilde{\lambda}_j(x_B;\theta_k)\right) * \tilde{g}_j(x_B;\theta_k) \\
\text{where} \quad \tilde{\lambda}_j(x_B;\theta_k) &= \tilde{\Lambda}(x_B;\theta_k)_j \\
\tilde{\Lambda}(x_B;\theta_k) &= \left(J(\tilde{\boldsymbol{g}}(x_B;\theta_k)) \cdot J(\tilde{\boldsymbol{g}}(x_B;\theta_k))^T\right)^{-1} \left(-J(\tilde{\boldsymbol{g}}(x_B;\theta_k)) \cdot J(\bar{f}(x_B;\theta_k))^T + \tilde{\boldsymbol{g}}(x_B;\theta_k)\right) \\
\bar{f}(x_B;\theta_k) &= \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k)\right] = \frac{1}{B}\sum_{i=1}^{B} f(x_i;\theta_k) \\
\tilde{\boldsymbol{g}}(x_B;\theta_k) &= \texttt{Reduce}_{x_i \in x_B}(\boldsymbol{g}(x_B;\theta_k)) = \quad e.g. \quad \sum_{i=1}^{B}\sum_{j=1}^{M} \left(g_j(x_i;\theta_k)\right)^2
\end{aligned} \tag{6}$$

Notice that this second version is completely identical to the unbatched version by using the functions $\bar{f}(x_B;\theta_k)$ and $\tilde{\boldsymbol{g}}(x_B;\theta_k)$ as approximations of the minimization and constraint functions.

Before we provide a quick analysis of the theoretical complexity of the two versions of this method, we first wish to comment on the method for computing $\Lambda(x_i;\theta_k)$. Rather than forming the inverse matrix of the product of the Jacobian $J(\boldsymbol{g}(x_i;\theta_k))$ (or $J(\tilde{\boldsymbol{g}}(x_i;\theta_k))$) and its transpose, we recommend solving the equation directly:

$$\left(J(\boldsymbol{g}(x_i;\theta_k)) \cdot J(\boldsymbol{g}(x_i;\theta_k))^T\right) \cdot \Lambda(x_i;\theta_k) = -J(\boldsymbol{g}(x_i;\theta_k)) \cdot J(f(x_i;\theta_k))^T + \boldsymbol{g}(x_i;\theta_k)$$

In particular, we recommend using an iterative method such as Conjugate Gradient (Straeter, 1971), since $\Lambda(x_i;\theta_{k-1})$ is likely a good estimate for $\Lambda(x_i;\theta_k)$. However, observing that the size of the matrix in this matrix-vector equation is $M \times M$ and that $M \ll B$ and $M \ll N$ (N the number of trainable parameters of the network), this final step in computing $\Lambda(x_i;\theta_k)$ does not contribute strongly to the overall complexity and therefore the precise method of computing the multipliers is not very significant.

In our analysis of the asymptotic complexity of this method, we will use N to denote the number of trainable parameters in the network, B for the batch size, and M for the number of (scalar-valued) constraint functions. Similarly, we let $BP(N)$ denote the complexity of backpropagation, $G(N,M,B)$ denote the complexity of computing the constraint function values, and $F(N,B)$ denote the complexity of a normal forward pass on the network with N trainable parameters and batchsize B. In general, these may also depend on the architecture of the neural network. For the fully constrained version, $FC(N,M,B)$, we have

$$\begin{aligned}
FC(N,M,B) &\in \mathcal{O}(\underbrace{BP(N)}_{\substack{\text{backpropagation of} \\ \text{single valued } \mathcal{L}(x_B;\theta_k)}} + F(N,B) + G(N,M,B) + B * (\underbrace{M^3}_{\substack{\text{upper bound for solving} \\ \text{matrix-vector equation}}} + \overbrace{\overbrace{M^2 * N}^{J(\boldsymbol{g})\cdot J(\boldsymbol{g})^T}}^{\text{batched computation of multipliers}} + \underbrace{M * N}_{J(\boldsymbol{g})\cdot J(f)^T} + \overbrace{M}^{\text{sum on RHS}})) \\
&\in \mathcal{O}\left(BP(N) + F(N,B) + G(N,M,B) + B * M^3 + B * M^2 * N\right) \\
&\in \mathcal{O}\left(BP(N) + F(N,B) + G(N,M,B) + B * M^2 * N\right)
\end{aligned} \tag{7}$$

where in the last line we used the assumption that $M \ll N$, which holds for any practical application. Similarly, if we define $\bar{F}(N, B)$ as the complexity of computing $\bar{f}$ and $\tilde{G}(N, M, B)$ as the complexity of computing $\tilde{\boldsymbol{g}}$, for the reduction version $Red(N, M, B)$ we have

$$\begin{aligned} Red(N, M, B) &\in \mathcal{O}(\underbrace{BP(N)}_{\substack{\text{backpropagation of} \\ \text{single valued } \mathcal{L}(x_B;\theta_k)}} + \bar{F}(N, B) + \tilde{G}(N, M, B) + \underbrace{M^3}_{\substack{\text{upper bound for solving} \\ \text{matrix-vector equation}}} + \overbrace{\overbrace{M^2 * N}^{J(\tilde{\boldsymbol{g}})\cdot J(\tilde{\boldsymbol{g}})^T}}^{\text{computation of multipliers}} + \underbrace{M * N}_{J(\tilde{\boldsymbol{g}})\cdot J(\bar{f})^T} + \overbrace{M}^{\text{sum on RHS}}) \\ &\in \mathcal{O}\left(BP(N) + \bar{F}(N, B) + \tilde{G}(N, M, B) + M^3 + M^2 * N\right) \\ &\in \mathcal{O}\left(BP(N) + \bar{F}(N, B) + \tilde{G}(N, M, B) + M^2 * N\right) \end{aligned} \tag{8}$$

From this, we can see that, as the complexity of using soft-constraints is approximately $\mathcal{O}\left(BP(N) + F(N, B) + G(N, M, B)\right) \approx \mathcal{O}\left(BP(N) + \bar{F}(N, B) + \tilde{G}(N, M, B)\right)$, the cost of this method adds only the final term in both (Eq. 7) and (Eq. 8). Further, we notice that the reduction version is more computationally efficient by a factor of the batch size, which was exactly the motivation of defining and using the batch-wise reduction. However, as both versions scale with the number of trainable parameters of the network, even from a theoretical perspective we have reason to worry that the complexity of this method is prohibitive for all but the smallest of networks. Despite this, since this method come with the guarantee that the magnitude of the residual of the constraints will decay exponentially quickly, assuming that the training process does not diverge, it may be worth the computational cost for certain problems.

NONLINEAR PROJECTION

While both versions of the above method operate within the framework of training the neural network as a constrained optimization task, there is another option available which may not suffer from the high computational complexity of constrained neural network optimization. Namely, we can train a neural network in the normal, unconstrained fashion using a standard minimization function and then, once we have finished training the network completely, we can clone the model and the train the clone using just the constraint function and a set of data specifically for constraining (similar to validation data). The idea here relies on the following observation: for any vector of neural network parameters θ_k at timestep k, there exists a nearest vector θ^+ in parameter space (Θ) which satisfies the constraint function. In other words, we define

$$\theta^+ = \underset{\substack{\theta \in \Theta \\ \boldsymbol{g}(\theta) = \mathbf{0}}}{\operatorname{argmin}} \|\theta_k - \theta\|$$

Geometrically, this projection of the trainable parameters of the network to the implicit constraint surface is related to the constrained optimization methods as shown in Figure 4. While the constrained optimization technique takes the "direct" path through parameter space to the optimal parameters which satisfy the constraints, the nonlinear projection method breaks the problem into two steps which together yield a similar result.

There are some distinct advantages to this method over the constrained optimization. First, since for the first stage we are only dealing with unconstrained problem, we can simply train the model in any way we like before applying this method. In other words, we can apply this method to a fully trained model. Further, since during the second stage we are only worrying about the constraint function, if we define a batch-wise reduction of $\boldsymbol{g}$ in exactly the same way as the reduction version of constrained optimization, for the second stage we also have a scalar-valued (positive semi-definite) minimization function, so we can "project" the model exactly like during training by simply swapping out the minimization function. Additionally, we note that the constraint manifold as embedded in $\mathbb{R}^N \cong \Theta$ may be very contorted. If this is the case, then the Euclidean distance between θ and θ^*, which is used when training a neural network normally, may be much less than the distance along the constraint surface, as illustrated in Figure 5. As a

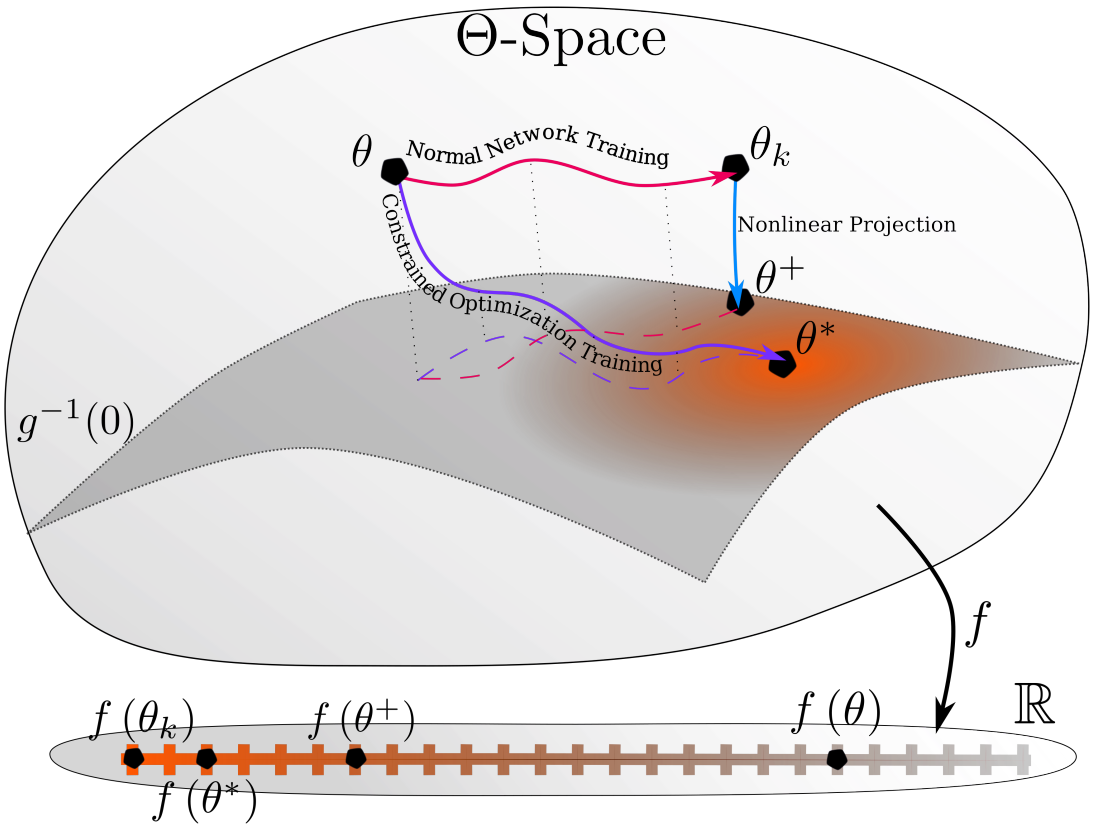

Figure 4. Comparison of the constrained optimization training method and the nonlinear projection method. While the constrained optimization method (purple) takes a more direct path to the optimal parameters θ^*, nonlinear projection (blue) can be applied to an already trained model at iteration k with parameters θ_k. This allows for any desired method for the initial training of an unconstrained model (red). The final parameters θ^+ resulting from the nonlinear projection may not be the most optimal parameters. The orange color fill indicates how negative the value of the loss function is, as neural network training attempts to minimize the loss function.

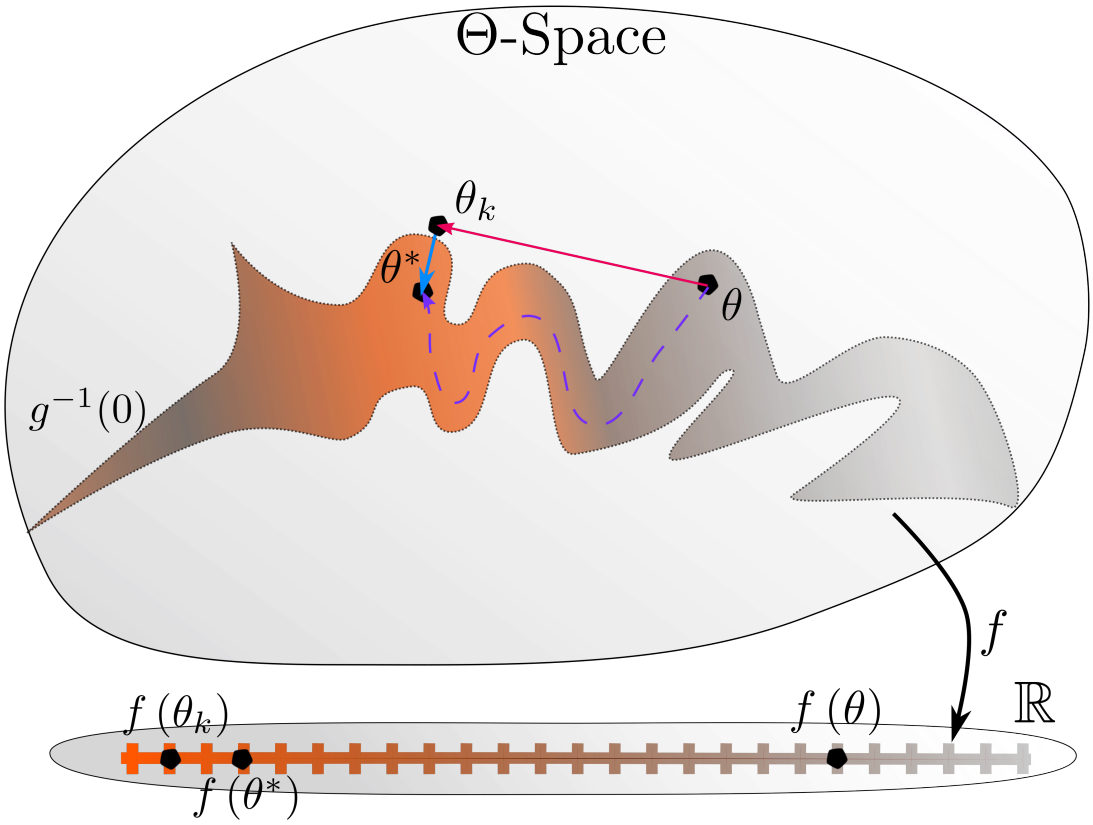

Figure 5. Comparison of the Euclidean distance and distance along the constraint surface of the current parameters of the network θ and the optimal parameters θ^* in the case where the constraint manifold is highly contorted. Colors as in Figure 4.

consequence, training normally could require fewer iterations than the constrained optimization method, even though the constrained optimization method is more effective with the steps it takes. This observation, in combination with the observation that the overall complexity of normal training followed by nonlinear projection does not have the extra $M^2 * N$ term, suggests this will likely scale better to larger networks.

The main disadvantage to this method compared to the previous method is that we have no convergence guarantees beyond that of normal neural network backpropagation. In particular, when we are projecting the model during the second stage of training, we may move far away from the minimum of the original minimization function (θ^*). In other words, while we will find the closest point in parameter space to the minimum of the original minimization function which satisfies the constraint function (θ^+), that does not mean that our final point is a minimum of the constrained optimization problem ($\theta^+ \neq \theta^*$). Another disadvantage of this technique is that the learning rate for the first stage of training is not necessarily the same as the learning rate during the second stage of projection. This possibly requires an additional hyperparameter search for the model, which could slow the overall training process. Despite this, since each stage of the training process is only as expensive as the normal method for training a neural network, this is a much more computationally feasible method.

Interestingly, this technique of nonlinearly projecting the weights of the network through gradient descent has a dual interpretation. Since we are only modifying a clone of the model and not the original trained model, this process can be repeated independently for different constraint functions and different data for constraining. As such, this method can be viewed as asking hypothetical questions of a trained model, where the question itself is phrased in terms of a constraint function with accompanying data. This leads to an interesting distinction between this method of producing a constrained model and the method of constrained optimization.

For a given model architecture and initial parameter values, constrained optimization produces the nearest parameter values which minimize the minimization function and the error of the constraint function at the observed training data. On the other hand, for a given model architecture and final parameter values, nonlinear projection produces the nearest parameter values which minimize the error of the constraint function on the provided data for constraining. While the first answers the question: "What is the solution given these constraints?", the second answers the question: "Given the previous learned domain knowledge from the training data and this constraint function and data as counterfactual evidence, what is the best solution?" As such, we suggest that the method of nonlinearly projecting the network parameters post-training offers an interesting interpretability technique, but we do not consider this idea further here, leaving that to possible future work.

To summarize the constrained optimization training method and the nonlinear projection method in terms of the functions which they minimize during backpropagation (their effective loss functions), we see that the fully constrained version of constrained optimization minimizes

$$\mathcal{L}(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k) + \sum_{i=1}^{M} \texttt{nograd}(\lambda_i(\theta_k)) * g_i(\theta_k)\right] \tag{9}$$

the reduction version minimizes

$$\mathcal{L}(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k)\right] + \sum_{j=1}^{M} \texttt{nograd}\left(\tilde{\lambda}_j(x_B;\theta_k)\right) * \tilde{g}_j(x_B;\theta_k) \tag{10}$$

and the nonlinear projection method minimizes

$$\mathcal{L}_1(x_B;\theta_k) = \mathop{\mathbb{E}}_{x_B}\left[f(x_B;\theta_k)\right] \tag{11}$$

and

$$\mathcal{L}_2(x_B;\theta_k) = \sum_{j=1}^{M} \mu_j(x_B;\theta_k) * \tilde{g}_j(x_B;\theta_k) \tag{12}$$

during the training ($\mathcal{L}_1$) and projecting ($\mathcal{L}_2$) stages, with μ_j weights for each of the component constraint functions which are selected or defined beforehand and all other variables as defined in the above section. In particular, we note that, if we select $\mu_j = \tilde{\lambda}_j$ and use the same data for the projection stage as the training stage, then the reduction version of constrained optimization is very similar to the nonlinear projection, except that the training and projection stages are interwoven within each batched parameter update.

METHODOLOGY

Constrained Optimization

To test and compare the different constraining methods, we investigated a simple toy problem: whether a relatively small neural network could learn to represent an arbitrary 1-D sine wave under the loss of mean squared error with the Helmholtz equation as a PDE constraint. The Helmholtz equation written in standard form is

$$\nabla_x^2 u(x) + k^2 u(x) = 0$$

where u is a 1-D function being constrained, k is the frequency of the ground truth sine wave, and $\nabla_x^2 u(x)$ is the second derivative of $u(x)$. Here, we compare the two different constrained optimization methods against two baselines: an unconstrained model and a soft-constrained model. For the reduction version, we take the mean of the Huber error of the constraint along the batch (Huber, 1992). For the soft-constrained model, we use a weighting of $\lambda = 1/B$, where B is the batch size of 100. The learning rate was held constant at 0.001 for all methods. The architecture of the network was also constant and is shown in Figure 6. This architecture was

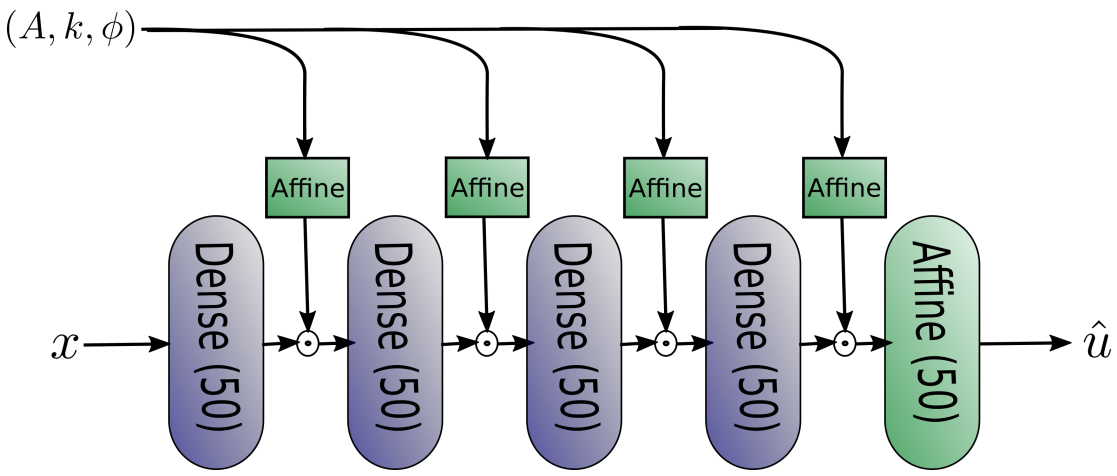

Figure 6. Architecture for the neural network used in all main experiments. Each blue box corresponds to a dense layer followed by only a Swish activation. Each green box corresponds to a dense layer with linear activation. The $\odot$ operations indicate element-wise multiplication (scaling).

inspired by both Karras et al. (2018) and Ha et al. (2016), as it functions both as a "style-transfer" network which transfers the style of each individual sine wave to the network outputs and also as a network parameterized by another network through the upper row of affine transformations. Importantly, rather than use ReLU as the activation function, which has a second derivative of 0 everywhere, we use the Swish function proposed by Ramachandran et al. (2017), which has a symmetric, non-zero second derivative.

The training data consists of 1000 different sine waves with 50 x-values sampled uniformly in $[-1, 1]$ for each wave. The sine waves are formed with

$$f(x; A, k, \phi) = A * \sin(k * x + \phi)$$

where A, k, ϕ can each take 10 possible values equally spaced across the intervals $[0.2, 5.0]$, $[0.4 * \pi, 10 * \pi]$, and $[-0.5, 0.5]$ (including the endpoints). The testing data is a single sine wave with $A = 1.0$, $k = 1.0$, and $\phi = 0.0$ and 500 x-values equally spaced across the interval $[-1, 1]$ (including the endpoints). The model is provided with the tuple (x, A, k, ϕ) as input and should output $f(x; A, k, \phi)$. All methods other than the fully constrained version of constrained optimization were trained 40 epochs. The fully constrained version was only trained for long enough to get statistics on the amount of time required for a single iteration, as training the model under this method was nearly computationally infeasible.

Nonlinear Projection

We tested the post-training projection method on the same toy problem as the constrained optimization experiment. Here, we trained the same architecture (Figure 6) with the same learning rate of 0.001 and the same loss of mean squared error. The training data also consisted of the same 1000 sine waves, but for this experiment we sampled 500 x-values uniformly in $[-1, 1]$ for each wave. The data used for projecting the model after training was the same as the testing data from above, except we only drew 100 x-values equally spaced across the interval $[-1, 1]$ (including the endpoints). For training, since we increased the amount of data by a factor of 10, we also increase the batch size to 1000. For projection, we used a different batch size and learning rate than during training. We found a batch size of 50 and a learning rate of 10^{-4} to work well.

The training and projection process was as follows: during each training epoch, the model was trained as normal. After each epoch, a clone was made and the clone was projected until the mean squared error of the Helmholtz constraint was less than 10^{-3}. On average, this was around 6000 projection iterations. Training on the original model was then resumed. In this way, the

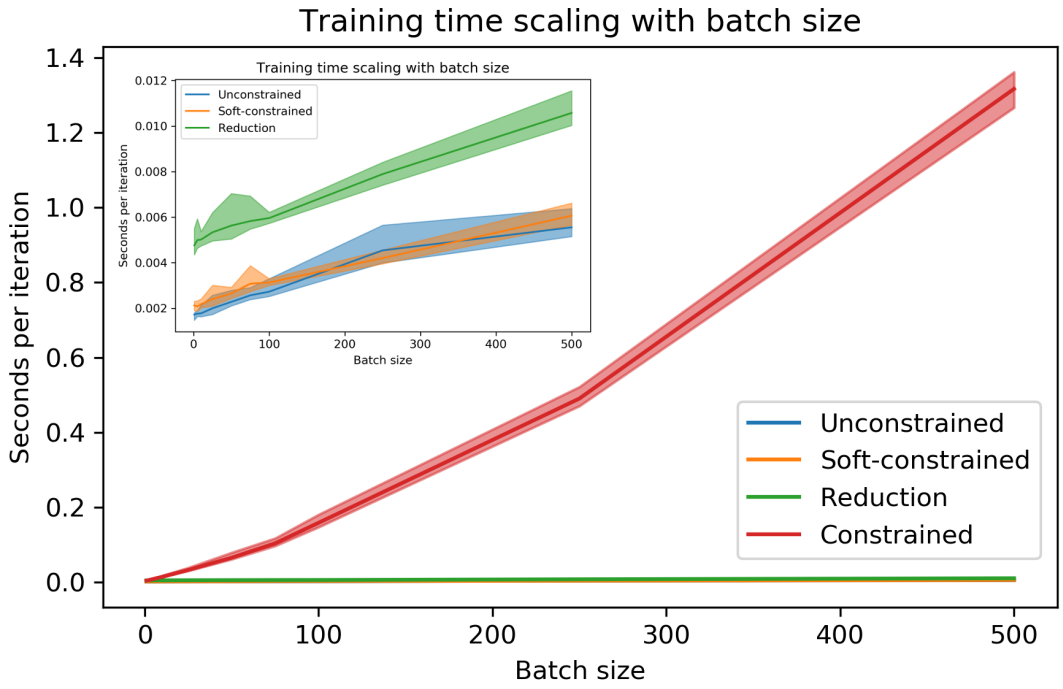

Figure 7. Dependence of training time on batch size for different constrained optimization techniques. Solid lines show the mean time required for the method and the fill shows the 95% confidence interval over 100 runs. Inset shows all methods except the most expensive to better enable comparison. Here, the number of trainable parameters is approximately 600 and the number of constraint functions is 1.

original model was never influenced by information about the constraint. After the final epoch, we similarly clone the model and project the clone so that we have an unprojected and projected version of the final model. For this experiment, the model was allowed to train for 20 epochs.

RESULTS AND DISCUSSION

Constrained Optimization

Figures 7-9 show the impact of different training methods on the time required to perform a single update iteration. Figure 7 depicts the dependence on the batch size, Figure 8 depicts the dependence on the size of the model, and Figure 9 depicts the dependence on the number of constraint functions. All plots show that there is no statistical difference between the training time for the soft-constrained and unconstrained baselines (see inset of all three plots). This is to be expected, as the only additional cost that the soft-constrained baseline requires over the unconstrained baseline is the computation of the residual of the constraints. For most problems and neural network architectures, the cost of backpropagation already greatly exceeds that of a forward pass, so the cost of computing the constraint residual alone (which is typically not more expensive than a second forward pass) is minimal. By comparison, we can see that the cost of the fully constrained method for a single constraint is orders of magnitude larger than the baselines and the cost of the reduction version for a single constraint is approximately twice that of the baselines.

Training time for both proposed constraining methods scales linearly with both the model size and the number of constraints. Given the asymptotic analysis above, this is expected for model size, but not for the number of constraints. However, we suspect that the complexity of the forward pass for computing the constraints has a much larger coefficient than 100^2, and so even for 100 constraints, it is still the dominant term. Training time for the reduction version also

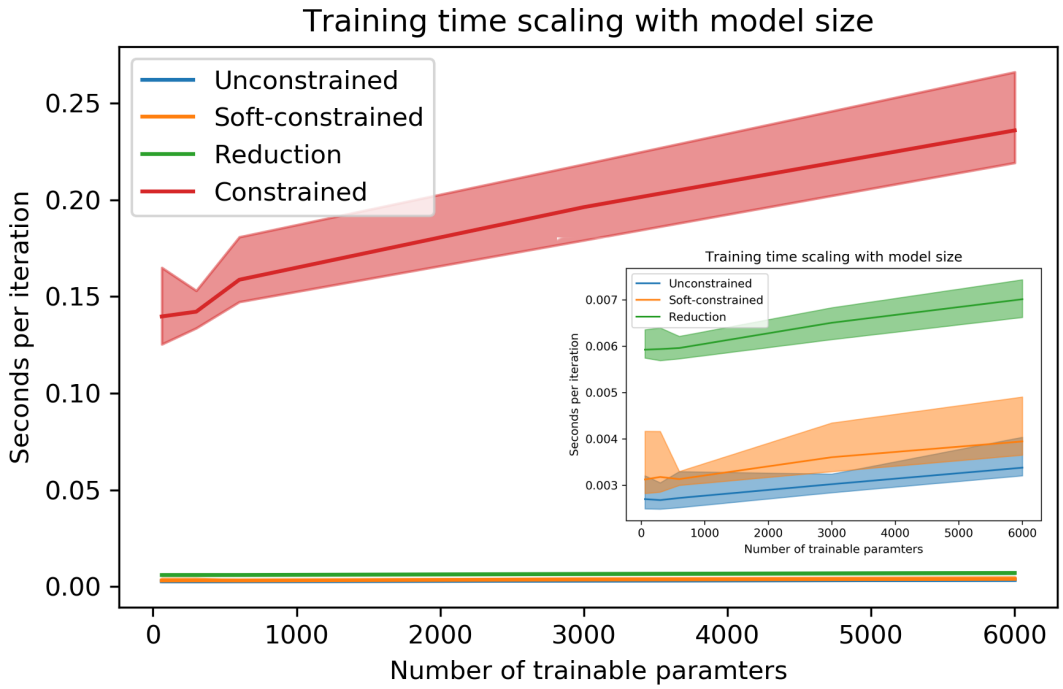

Figure 8. As in Figure 7, but for the dependence of training time on model size. Here, the batch size is 100 and the number of constraints functions is 1.

scales linearly with batch size, whereas training time for the fully constrained version appears to scale quadratically. This shows the key advantage of the reduction version over the fully constrained version: since the reduction is performed along the batch axis, the complexity of the reduction version with regards to the batch size should scale similarly to that of the two baselines. This is confirmed by the inset of Figure 7. As indicated by all plots, the complexity of the fully constrained version of constrained optimization is simply far too high to be useful for any practical problem, so for the remaining experiment on constrained optimization, we have excluded this version. In fact, this complexity is so high that we conclude here that the fully constrained version is not a promising tool for constraining neural networks.

Figure 10 shows the data loss and magnitude of the constraint residual for the reduction version of constrained optimization and both baselines on the toy sine wave with Helmholtz constraint problem. Figure 11 similarly shows the predictions of these methods on a held-out test case. Examining the reduction method and the unconstrained baseline, we see that both methods have surprisingly large constraint residual magnitudes, especially considering that the predictions of both methods appear to be fairly close to the ground truth. It seems unlikely that that the predictions could be as close as they are to the desired curves and as smooth as they are while the second derivative of the model is possibly as high as 10^3. This apparent discrepancy we explain by considering what the architecture of neural networks implies about their derivatives. Most neural network architectures (including those tested here) consist of a large number of neurons whose output y in terms of the inputs $\boldsymbol{x}$ is given by

$$y = h(\sum_{i=1}^{N} w_i * x_i)$$

where $\boldsymbol{w}$ is the learned weights of the neuron and h is a nonlinear function. When a large number of these are applied in sequence (*i.e.* when the network is several layers deep), then it is very likely that the later neurons in the network produce a weighted sum of their inputs which largely cancels out. As such, the derivative across the neuron for each of its inputs can be very large,

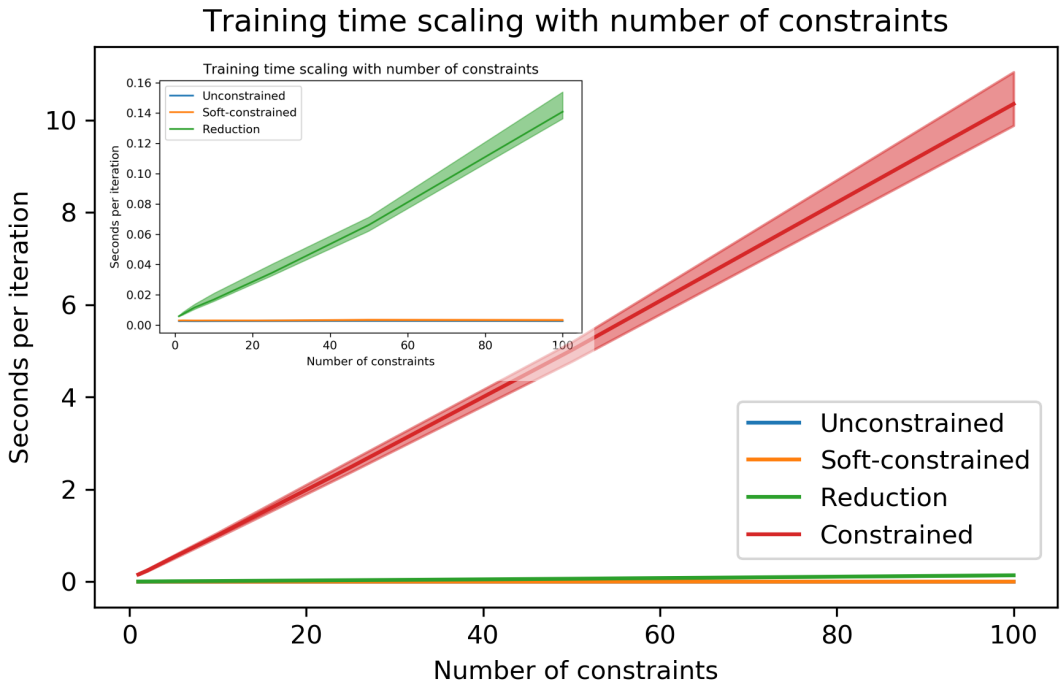

Figure 9. As in Figure 7, but for the dependence of training time on number of constraint functions. Here, the batch size is 100 and the number of trainable parameters is approximately 600.

even though the output of the neuron itself remains fairly small across a wide range of different input values. We believe that this is the source of the discrepancy: the predictions of the models are deceptively smooth on the scales required for plotting, even though in reality the derivatives at each individual point are quite large.

Given the discrepancy noted above, one might question whether it even makes sense to try and constrain the derivatives of the network directly. In fact, we suggest that it does not. Little is known about the properties of the neural networks as representations of function and whether they even form a complete basis for all L_2 functions. In any case, the very large second derivatives for both the reduction method and the unconstrained baseline suggest that it is very difficult to produce a neural network which produces the correct output and also produces the correct derivatives.

Actually comparing the reduction version of constrained optimization with the unconstrained baseline, we see that the constrained optimization does not seem to be helping with regards to either the data loss or the constraint magnitude. In fact, the median value of the magnitude of the constraint residual for the reduction method is nearly 5 times that of the unconstrained case. However, we caution that these results are for a single run and therefore it is possible that the difference is not actually significant. In any case, these results are not promising for the reduction version of constrained optimization, as while it is possible that it did not hurt the test performance it is unlikely that it helped. Considering the additional computational cost of the reduction method over the baselines, we conclude that, similar to the fully constrained version, the reduction version of constrained optimization is not a promising technique for constraining neural networks.

As a side note, from Figure 11 it appears that the soft-constrained method has failed to train. In actuality, the soft-constrained method was simply too strongly penalized and has found a solution which does minimize the constraints fairly well, namely always outputting 0. For a complete analysis, this would be an unfair comparison. However, since we have already determined that the constrained optimization method is not of practical utility, we have not

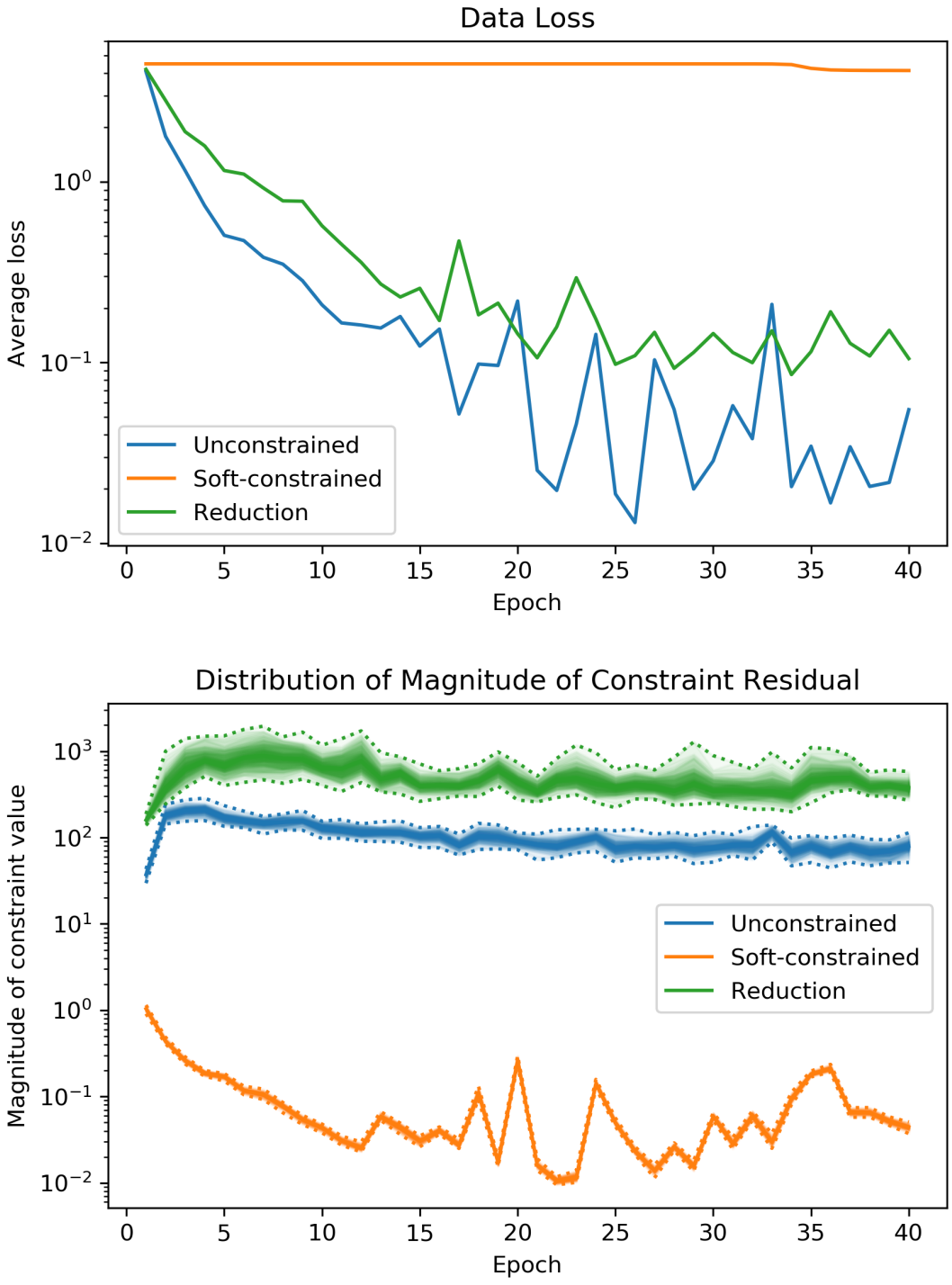

Figure 10. Data loss and magnitude of the constraint residual for three models. For the constraint residual, the distribution across all individual data points across the epoch is plotted. The dashed lines indicates the minimum and maximum values and the shading indicates how close the percentiles across the distribution are to the median.

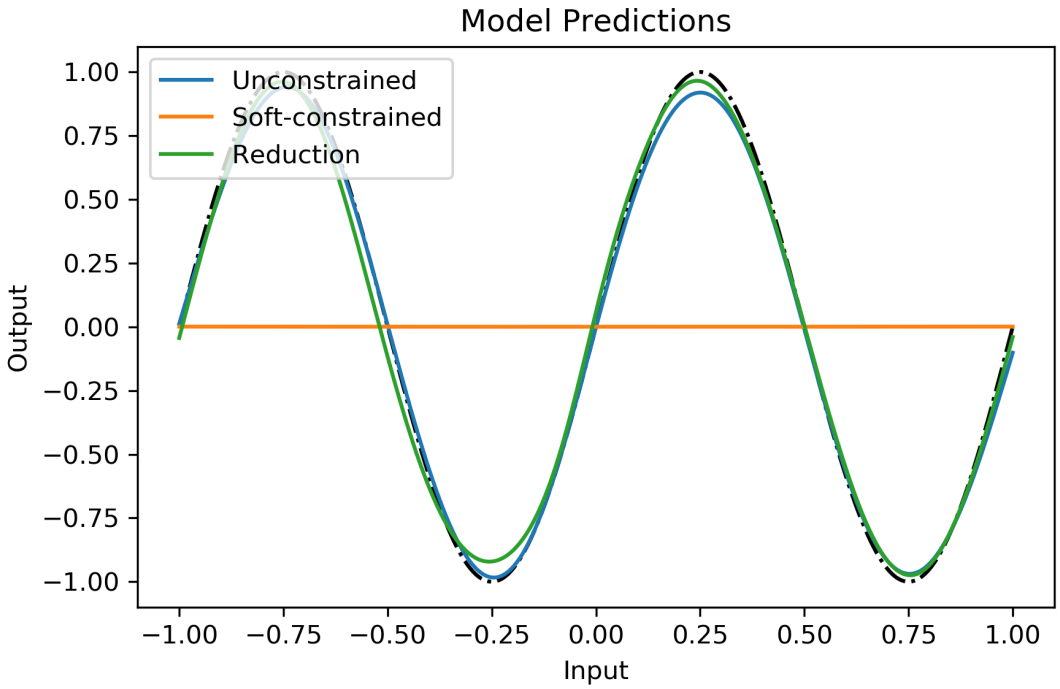

Figure 11. Predictions for the same models as in Figure 10 on testing data. The black dotted and dashed line is the ground truth.

performed a complete hyperparameter search for a better constraint weight. Instead, we offer this model's predictions as further support for our claims above regarding the difficulty of constraining a network's derivatives directly. Even though the model outputs 0 everywhere, the second derivative of the network, according to Figure 10, has magnitude of approximately 0.08 everywhere. This shows that, even for a very simple case, the inner dynamics of the network defy our expectations.

Nonlinear Projection

Figures 12 shows the data loss and magnitude of the residual of the constraint before and after the model parameters are projected to near the constraint surface, as evaluated on the data used for projection. As the projection process was run until the constraint error was less than 10^{-3}, the "Projected" line in the constraint error plot lies almost exactly at the value 10^{-3} across all epochs, regardless of the original constraint error. While the projection process is able to reduce the constraint error by approximately 4 orders of magnitude, it does so at severe cost to the data loss. Although the model has reached a loss of only 10^{-3} by the 20^{th} epoch before being projected, after projection, the model has at best an average loss of 0.18, which is no better than the unprojected model the second epoch of training. Considering that the projection process does not take into account the loss function of the model, this suggests that the projection process is likely undoing most of the effort of training in the first place.

Figures 13 and 14 show the unprojected and projected predictions for the model at each epoch and the predictions of the fully trained model during the projection process, respectively. These provide a much more detailed view into the projection technique and its effects on the model. Examining the unprojected predictions of the model during training, we see that the model quickly learns to predict something which approximates a sine wave, rapidly corrects the phase of the sine wave, and incrementally corrects the amplitude of the sine wave. The dashed purple line in Figure 13 shows that the model has approximated a sine wave even by the end of the first epoch, the dashed indigo line shows that the model has corrected the phase of the sine wave by the end of the third epoch, and the dashed orange and red lines show the model

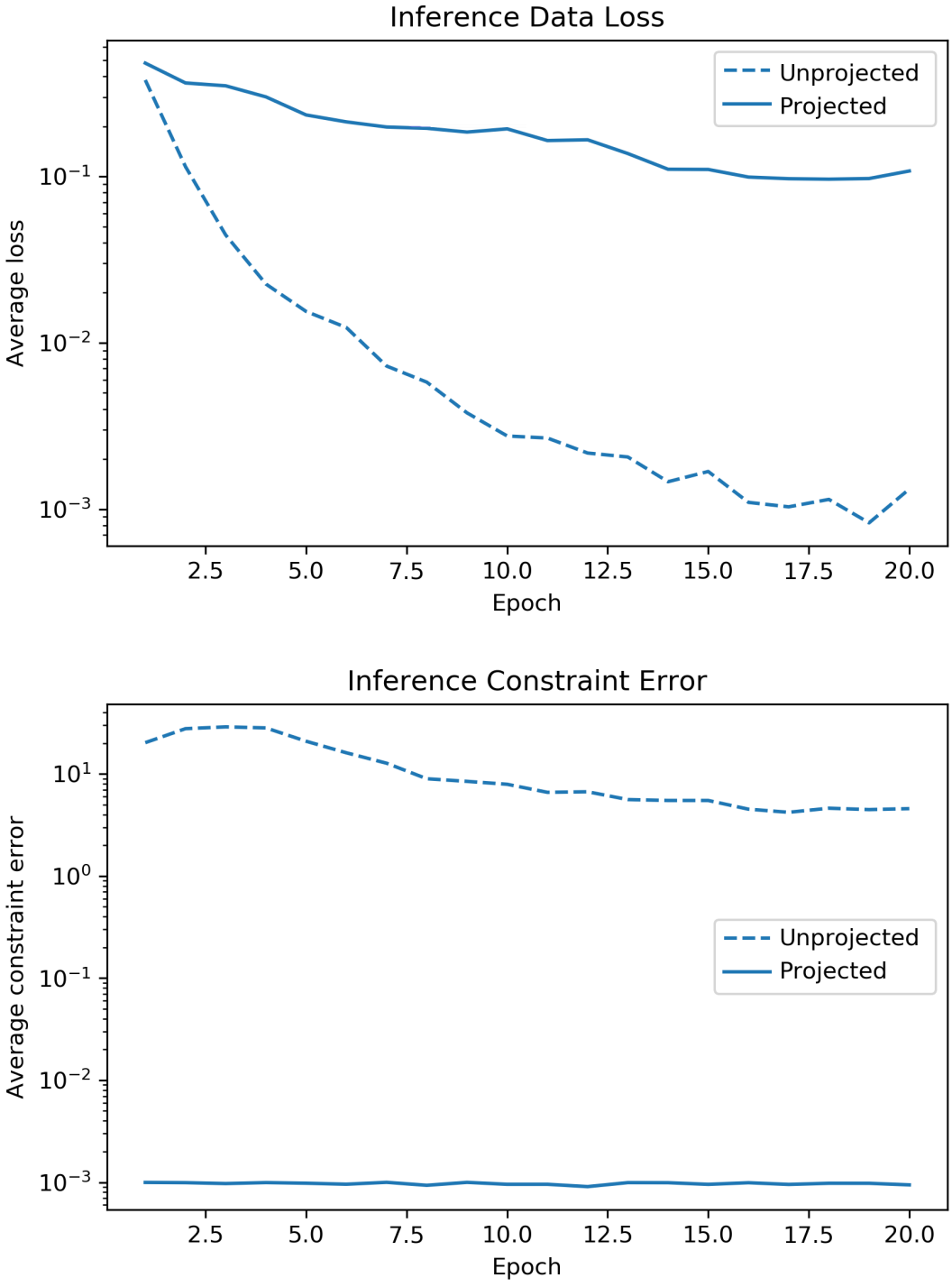

Figure 12. Data loss and magnitude of the constraint residual for the model before and after projection on test data.

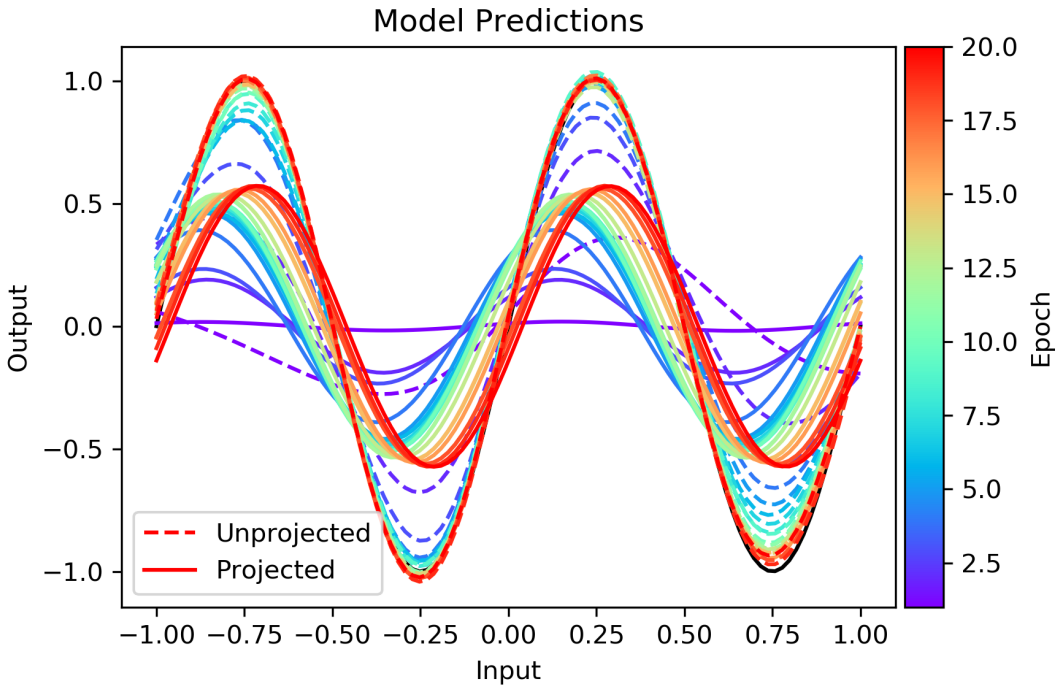

Figure 13. Predictions of the model before and after projection at various epochs. Each color, ranging from purple for epoch 1 to red for epoch 20, shows the predictions of the model before projecting and after projecting at that epoch. The solid black line shows the ground truth.

has quite nearly approximated the entire sine wave correctly by the end of the first 20 epochs of training.

Interestingly, the projected model predictions do not show all of the same trends as the unprojected model predictions. While all projected curves are clearly sine waves, the amplitudes are wildly incorrect. To first order, it appears as though the projection process has managed to correct the tiny imperfections in the model predictions by squishing the sine wave by a factor of 2. At the same time, the phase of the projected sine wave "wanders." To some extend, this is expected, as the constraint only contains information about the frequency of the sine wave, not the amplitude or the phase. Despite this, intuition suggests that the projection process would only need to make small corrections to the model, since the deviation from the nearest valid sine wave in Euclidean space is very small. Looking closely at Figure 14, we can see how the projection process incrementally affects the model predictions. As indicated by the very light grey lines, the model initially is projected by a significant amount away from its previous predictions and the ground truth, but at some point, the projections distance decreases and eventually converges to some sine wave. Exactly why the projection has chosen this sine wave in particular is unclear, but likely is influenced by both the model architecture and the precise values of the model parameters.

Based on the constrained optimization experiment results, we suggest that possibly the task of learning to exactly match the desired sine wave while satisfying the second derivative constraints is likely surprisingly difficult for the model. Figures 13 and 14 provides some insight into why this might be the case. It seems that the backpropagation update from the constraint function is pointing the model in a completely different direction than the update from the loss function. In other words, the level surfaces of the constraint and loss functions in parameter space are very different and therefore the resulting "landscapes" are also dissimilar. Taking in particular the magnitude of the model outputs, we see that while the loss function is pushing the model to increase the amplitude, the constraint function is encouraging the model to decrease the amplitude, even though the amplitude of a sine wave is completely independent of its frequency.

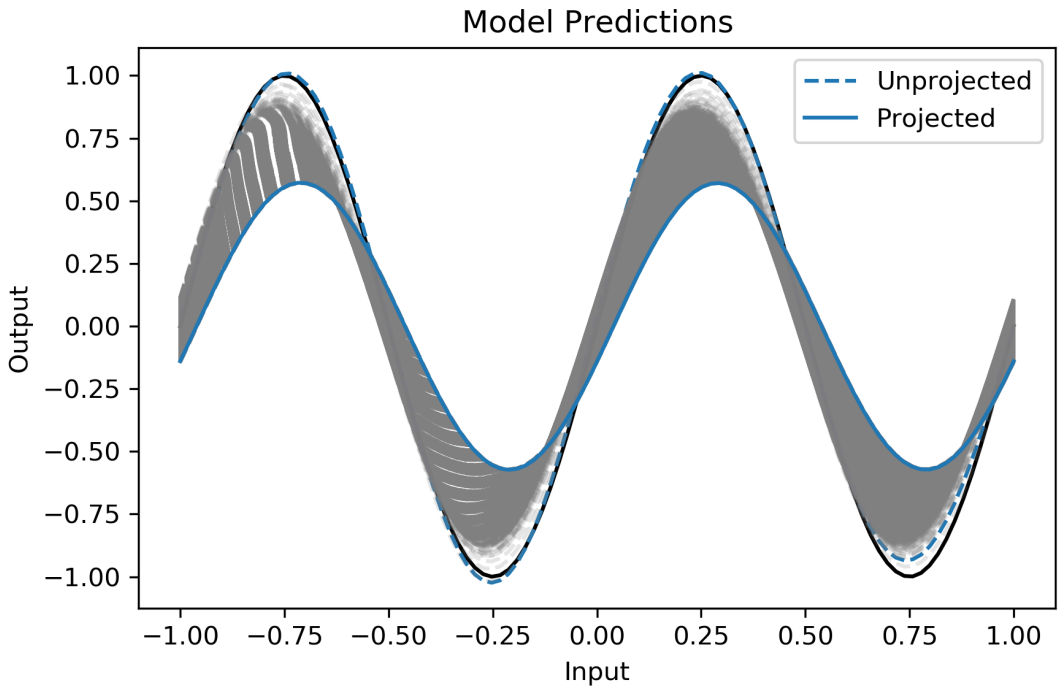

Figure 14. Predictions of the fully trained model during the projection process. The colored dashed line shows predictions before projection and the colored solid line shows predictions after projection is complete. The dashed grey lines show the predictions after each individual epoch of projection. The darker the grey line, the less the model predictions changed during that projection epoch. The solid black line shows the ground truth.

Our interpretation is that the architecture of the model itself is causing entanglement between the magnitude of its output and the magnitude of the second derivative of its output, which makes the task of producing the correct outputs and the correct second derivatives challenging.

CONCLUSION

The results from the constrained optimization toy experiments are particularly disappointing. While theoretically, it can be seen that the both versions of constrained optimization should yield a model whose deviation from the constraint manifold decays exponentially quickly with training time, in practice the complexity of the method makes it prohibitive for even the smallest problems. The reduction version, which trades out the actual desired constraint for the average magnitude of the residual of the constraint, is sufficiently fast enough to be used for small problems, yet still seems to be ineffective in practice.

The nonlinear projection method holds more promise than constrained optimization. It is the authors' belief that there likely exists a way of modifying the projection process to better respect the loss function of the model. The simplest method of doing this would be to perform projection with an augmented constraint function, much in the way that soft-constraining a model augments the standard loss function. We caution that doing this may impact the terminating condition of the projection method. Alternatively, the theory behind the constrained optimization of a neural network suggests that it may be possible to switch the roles of the constraint and loss functions and run the projection method with a sort of "reverse constrained optimization" technique. Of course, since the fully constrained version of constrained optimization is completely computationally infeasible, we recommend using the reduction trick. In the case of the "reverse constrained" projection, since we are simply trying to not increase the value of the loss function, the reduction version should be sufficient.

In light of the observed discrepancy between the smooth predictions of the model (at least, as determined by the human eye from plots) and the rather large magnitude of the constraint residual, we suggest that directly constraining the derivatives of the network, as would naïvely be done for any PDE constraint, is inadvisable. For even a simple toy problem, such as learning single sine functions, it appears to be very difficult to learn network weights which ensure accurate outputs and accurate derivatives, as it seems that the architecture of the model itself entangles the magnitude of the model outputs to its own second derivative. Future work might consider whether modifying the problem could make it more tractable. For instance, if, rather than approximating a desired function directly, the network instead represents a desired function by predicting coefficients for a Fourier basis, then a PDE constraint can be applied directly to those coefficients without restricting the derivatives of the neural network itself. This method has the additional advantages of ensuring completeness, as the Fourier basis is known to be a complete basis for all L_2 functions, and being more computationally efficient, as derivatives in spectral space can be computed very quickly (Canuto et al., 2006).

ACKNOWLEDGMENTS

This research was conducted at Lawrence Berkeley National Labs during the summer of 2019 and used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153).

Branin, F. H. (1972). Widely convergent method for finding multiple solutions of simultaneous nonlinear equations. *IBM Journal of Research and Development*, 16(5):504–522.

Canuto, C., Hussaini, M. Y., Quarteroni, A., and Zang, T. A. (2006). *Spectral methods*. Springer.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 6571–6583. Curran Associates, Inc.

Ha, D., Dai, A., and Le, Q. V. (2016). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Huber, P. J. (1992). Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer.

Karras, T., Laine, S., and Aila, T. (2018). A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*.

Kim, B., Azevedo, V. C., Thuerey, N., Kim, T., Gross, M., and Solenthaler, B. (2018). Deep fluids: A generative network for parameterized fluid simulations. *arXiv preprint arXiv:1806.02071*.

King, R., Hennigh, O., Mohan, A., and Chertkov, M. (2018). From deep to physics-informed learning of turbulence: Diagnostics. *arXiv preprint arXiv:1810.07785*.

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

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer.

Long, Z., Lu, Y., and Dong, B. (2018). Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network. *arXiv preprint arXiv:1812.04426*.

Long, Z., Lu, Y., Ma, X., and Dong, B. (2017). Pde-net: Learning pdes from data. *arXiv preprint arXiv:1710.09668*.

Márquez-Neila, P., Salzmann, M., and Fua, P. (2017). Imposing hard constraints on deep networks: Promises and limitations. *arXiv preprint arXiv:1706.02025*.

Mohan, A., Daniel, D., Chertkov, M., and Livescu, D. (2019). Compressed convolutional lstm: An efficient deep learning framework to model high fidelity 3d turbulence. *arXiv preprint arXiv:1903.00033*.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

Penrose, R. (1955). A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society*, volume 51, pages 406–413. Cambridge University Press.

Platt, J. C. and Barr, A. H. (1988). Constrained differential optimization. In *Neural Information Processing Systems*, pages 612–621.

Raissi, M. (2018). Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research*, 19(1):932–955.

Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 7.

Straeter, T. A. (1971). On the extension of the davidon-broyden class of rank one, quasi-newton minimization methods to an infinite dimensional hilbert space with applications to optimal control problems.

Tanabe, K. (1980). A geometric method in nonlinear programming. *Journal of Optimization Theory and Applications*, 30(2):181–210.

Thuerey, N., Weissenow, K., Mehrotra, H., Mainali, N., Prantl, L., and Hu, X. (2018). Well, how accurate is it? a study of deep learning methods for reynolds-averaged navier-stokes simulations. *arXiv preprint arXiv:1810.08217*.

Wiewel, S., Becher, M., and Thuerey, N. (2018). Latent-space physics: Towards learning the temporal evolution of fluid flow. *arXiv preprint arXiv:1802.10123*.

Zhang, S. and Constantinides, A. (1992). Lagrange programming neural networks. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(7):441–452.

PROOF OF EXPONENTIAL DECAY OF THE CONSTRAINT RESIDUAL

Theorem. *The method presented in Tanabe (1980) ensures exponential convergence of the constraint residual to* $\mathbf{0}$.

Proof. To provide greater enlightenment, we prove this starting from the desired result and work backwards. If the constraint function decays exponentially, then it is of the form

$$\boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t}$$

This corresponds to the differential equation

$$\begin{gathered}\dot{\boldsymbol{g}}(\theta(t)) = -\boldsymbol{g}(\theta(t)) \\ \iff \\ J(\boldsymbol{g}(\theta(t)))\frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = J(\boldsymbol{g}(\theta(t)))\dot{\theta}(t) = -\boldsymbol{g}(\theta(t))\end{gathered}$$

where in the last line, we used the chain rule to break up the left-hand side into terms which include $\dot{\theta}(t)$. Notice that this implicitly provides a condition that $\dot{\theta}(t)$ needs to satisfy. For the remainder of this proof, we will denote $\theta(t)$ by θ and simply keep in mind that θ is a function of time.

Under the assumption that $J(\boldsymbol{g}(\theta))$ is full-rank, then its pseudoinverse is also full-rank.Therefore, multiplying both sides by $J(\boldsymbol{g}(\theta))^+$ yields an equivalent condition:

$$\left(J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)\dot{\theta} = -J(\boldsymbol{g}(\theta))^+\boldsymbol{g}(\theta(t))$$

Since $J(\boldsymbol{g}(\theta))^+J(\boldsymbol{g}(\theta))$ is an orthogonal projection operator, for any vector v we know that

$$v = \left(J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)v + \left(I - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)v$$

In particular, for $\dot{\theta}$ we have

$$\begin{aligned}\dot{\theta} &= \left(J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)\dot{\theta} + \left(I - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)\dot{\theta} \\ &= -J(\boldsymbol{g}(\theta))^+\boldsymbol{g}(\theta(t)) + \left(I - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)\dot{\theta}\end{aligned}$$

Lastly, we note that since $(I - J(\boldsymbol{g}(\theta))^+J(\boldsymbol{g}(\theta)))$ is also an orthogonal projection operator, $\dot{\theta}$ consists of the sum of two perpendicular terms and therefore we can simply choose whatever desired function $\boldsymbol{h}(t)$ for $\dot{\theta}$ in the right summand and it will not affect the exponential decay. In other words, we see that

$$\begin{gathered}\dot{\theta} = -J(\boldsymbol{g}(\theta))^+\boldsymbol{g}(\theta) + \left(I - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)\boldsymbol{h}(t) \\ \implies \\ \boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t}\end{gathered}$$

for any choice of $\boldsymbol{h}(t)$. Since for normal backpropagation we take $\dot{\theta} = \boldsymbol{h}(t) = -\nabla f(\theta) = -J(f(\theta))^T$, we do the same here and obtain

$$\begin{gathered}\dot{\theta} = \Psi(\theta) = -\left(I - J(\boldsymbol{g}(\theta))^+ J(\boldsymbol{g}(\theta))\right)J(f(\theta))^T - J(\boldsymbol{g}(\theta))^+\boldsymbol{g}(\theta) \\ \implies \\ \boldsymbol{g}(t) = \boldsymbol{g}(\theta_0)e^{-t}\end{gathered}$$

□