JFB: Jacobian-Free Backpropagation for Implicit Networks

Samy Wu Fung,[*1] **Howard Heaton,**[*2] **Qiuwei Li,**[2] **Daniel McKenzie,**[2] **Stanley Osher,**[2] **Wotao Yin**[3]

[1] Department of Applied Mathematics and Statistics, Colorado School of Mines
[2] Department of Mathematics, University of California, Los Angeles
[3] Alibaba Group (US), Damo Academy

Abstract

A promising trend in deep learning replaces traditional feedforward networks with implicit networks. Unlike traditional networks, implicit networks solve a fixed point equation to compute inferences. Solving for the fixed point varies in complexity, depending on provided data and an error tolerance. Importantly, implicit networks may be trained with fixed memory costs in stark contrast to feedforward networks, whose memory requirements scale linearly with depth. However, there is no free lunch — backpropagation through implicit networks often requires solving a costly Jacobian-based equation arising from the implicit function theorem. We propose Jacobian-Free Backpropagation (JFB), a fixed-memory approach that circumvents the need to solve Jacobian-based equations. JFB makes implicit networks faster to train and significantly easier to implement, without sacrificing test accuracy. Our experiments show implicit networks trained with JFB are competitive with feedforward networks and prior implicit networks given the same number of parameters.[1]

A new direction has emerged from explicit to implicit neural networks (Winston and Kolter 2020; Bai, Kolter, and Koltun 2019; Bai, Koltun, and Kolter 2020; Chen et al. 2018; Ghaoui et al. 2019; Dupont, Doucet, and Teh 2019; Jeon, Lee, and Choi 2021; Zhang et al. 2020; Lawrence et al. 2020; Revay and Manchester 2020; Look et al. 2020; Gould, Hartley, and Campbell 2019). In the standard feedforward setting, a network prescribes a series of computations that map input data d to an inference y. Networks can also explicitly leverage the assumption that high dimensional signals typically admit low dimensional representations in some latent space (Van der Maaten and Hinton 2008; Osher, Shi, and Zhu 2017; Peyré 2009; Elad, Figueiredo, and Ma 2010; Udell and Townsend 2019). This may be done by designing the network to first map data to a latent space via a mapping Q_Θ and then apply a second mapping S_Θ to map the latent variable to the inference. Thus, a traditional feedforward $\mathcal{E}_\Theta$ may take the compositional form

$$\mathcal{E}_\Theta(d) = S_\Theta(Q_\Theta(d)), \tag{1}$$

[*]These authors contributed equally.

[1]All codes for experiments in this work are available on Github: https://github.com/howardheaton/jacobian_free_backprop

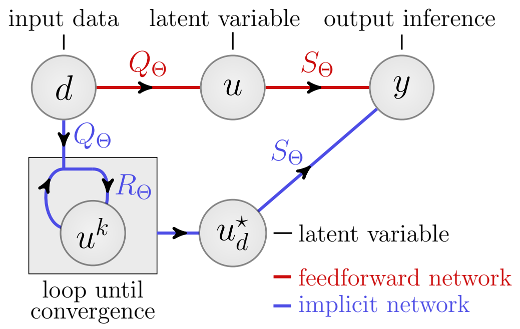


Figure 1: Feedforward networks act by computing $S_\Theta \circ Q_\Theta$. Implicit networks add a fixed point condition using R_Θ. When R_Θ is contractive (more generally: averaged) repeatedly applying R_Θ to update a latent variable u^k converges to a fixed point $u^\star = R_\Theta(u^\star; Q_\Theta(d))$.

which is illustrated by the red arrows in Figure 1. One can allow for computation in the latent space $\mathcal{U}$ by introducing a self-map $R_\Theta(\cdot; Q_\Theta(d))$ and the iteration

$$u^{k+1} = R_\Theta(u^k; Q_\Theta(d)). \tag{2}$$

Iterating k times may be viewed as a weight-tied, input-injected network, where each feedforward step applies R_Θ (Bai, Kolter, and Koltun 2019). As $k \to \infty$, *i.e.* the latent space portion becomes deeper, the limit of (2) yields a *fixed point equation*. Implicit networks capture this "infinite depth" behaviour by using $R_\Theta(\cdot\,; Q_\Theta(d))$ to define a fixed point condition rather than an explicit computation:

$$\mathcal{N}_\Theta(d) \triangleq S_\Theta(u_d^\star) \quad \text{where} \quad u_d^\star = R_\Theta(u_d^\star, Q_\Theta(d)), \tag{3}$$

as shown by blue in Figure 1. Special cases of the network in (3) recover architectures introduced in prior works:

- ▷ Taking S_Θ to be the identity recovers the well-known Deep Equilibrium Model (DEQ) (Bai, Kolter, and Koltun 2019; Bai, Koltun, and Kolter 2020).
- ▷ Choosing S_Θ as the identity, Q_Θ to be an affine map and $R_\Theta(u, Q_\Theta(d)) = \sigma(Wu + Q_\Theta(d))$ yields Monotone Operator Networks (Winston and Kolter 2020) as long as W and σ satisfy additional conditions. Allowing S_Θ to be linear yields the model proposed in (Ghaoui et al. 2019).

Three immediate questions arise from (3):

- ► Is the definition in (3) well-posed?
- ► How is $\mathcal{N}_\Theta(d)$ evaluated?
- ► How are the weights Θ of $\mathcal{N}_\Theta$ updated during training?

Since the first two points are well-established (Winston and Kolter 2020; Bai, Kolter, and Koltun 2019), we briefly review these in Section 2 and focus on the third point. Using gradient-based methods for training requires computing $d\mathcal{N}_\Theta \big/ d\Theta$, and in particular, $du_d^\star \big/ d\Theta$. Hitherto, previous works computed $du_d^\star \big/ d\Theta$ by solving a Jacobian-based equation (see Section 3). Solving this linear system is computationally expensive and prone to instability, particularly when the dimension of the latent space is large and/or includes certain structures (*e.g.* batch normalization and/or dropout) (Bai, Kolter, and Koltun 2019; Bai, Koltun, and Kolter 2020).

Our primary contribution is a new and simple **Jacobian-Free Backpropagation** (JFB) technique for training implicit networks that avoids *any* linear system solves. Instead, our scheme backpropagates by omitting the Jacobian term, resulting in a form of preconditioned gradient descent. JFB yields much faster training of implicit networks and allows for a wider array of architectures.

1 Why Implicit Networks?

Below, we discuss several advantages of implicit networks over explicit, feedforward networks.

Implicit networks for implicitly defined outputs In some applications, the desired network output is most aptly described implicitly as a fixed point, not via an explicit function. As a toy example, consider predicting the variable $y \in \mathbb{R}$ given $d \in [-1/2, 1/2]$ when (d, y) is known to satisfy

$$y = d + y^5. \tag{4}$$

Using $y_1 = 0$ and the iteration

$$y_{k+1} = T(y_k; d) \triangleq d + y_k^5, \quad \text{for all } k \in \mathbb{N}, \tag{5}$$

one obtains $y_k \to y$. In this setting, y is exactly (and implicitly) characterized by $y = T(y, d)$. On the other hand, an explicit solution to (4) requires an infinite series representation, unlike the simple formula $T(y, d) = d + y^5$. See Appendix F for further details. Thus, it can be simpler and more appropriate to model a relationship implicitly. For example, in areas as diverse as game theory and inverse problems, the output of interest may naturally be characterized as the fixed point to an operator parameterized by the input data d. Since implicit networks find fixed points by design, they are well-suited to such problems as shown by recent works (Heaton et al. 2021a,b; Gilton, Ongie, and Willett 2021).

"Infinite depth" with constant memory training As mentioned, solving for the fixed point of $R_\Theta(\cdot\,; Q_\Theta(d))$ is analogous to a forward pass through an "infinite depth" (in practice, very deep) weight-tied, input injected feedforward network. However, implicit networks do not need to store intermediate quantities of the forward pass for backpropagation. Consequently, implicit networks are trained using *constant memory costs* with respect to depth – relieving a major bottleneck of training deep networks.

No loss of expressiveness Implicit networks as defined in (3) are at least as expressive as feedforward networks. This can easily be observed by setting R_Θ to simply return Q_Θ; in this case, the implicit $\mathcal{N}_\Theta$ reduces to the feedforward $\mathcal{E}_\Theta$ in (1). More interestingly, the class of implicit networks in which S_Θ and Q_Θ are constrained to be affine maps contains all feedforward networks, and is thus at least as expressive (Ghaoui et al. 2019), (Bai, Kolter, and Koltun 2019, Theorem 3). Universal approximation properties of implicit networks then follow immediately from such properties of conventional deep neural models (*e.g.* see (Csáji et al. 2001; Lu et al. 2017; Kidger and Lyons 2020)).

We also mention a couple limitations of implicit networks.

Architectural limitations As discussed above, in theory given any feedforward network one may write down an implicit network yielding the same output (for all inputs). In practice, evaluating the implicit network requires finding a fixed point of R_Θ. The fixed point finding algorithm then places constraints on R_Θ (*e.g.* Assumption 2.1). Guaranteeing the existence and computability of $d\mathcal{N}_\Theta \big/ d\Theta$ places further constraints on R_Θ. For example, if Jacobian-based backpropagation is used, R_Θ cannot contain batch normalization (Bai, Kolter, and Koltun 2019).

Slower inference Once trained, inference with an implicit network requires solving for a fixed point of R_Θ. Finding this fixed point using an iterative algorithm requires evaluating R_Θ repeatedly and, thus, is often slower than inference with a feedforward network.

2 Implicit Network Formulation

All terms presented in this section are provided in a general context, which is later made concrete for each application. We include a subscript Θ on various terms to emphasize the indicated mapping will ultimately be parameterized in terms of tunable weights[2] Θ. At the highest level, we are interested in constructing a neural network $\mathcal{N}_\Theta : \mathcal{D} \to \mathcal{Y}$ that maps from a data space[3] $\mathcal{D}$ to an inference space $\mathcal{Y}$. The implicit portion of the network uses a latent space $\mathcal{U}$, and data is mapped to this latent space by $Q_\Theta : \mathcal{D} \to \mathcal{U}$. We define the *network operator* $T_\Theta : \mathcal{U} \times \mathcal{D} \to \mathcal{U}$ by

$$T_\Theta(u; d) \triangleq R_\Theta(u, Q_\Theta(d)). \tag{6}$$

Provided input data d, our aim is to find the unique fixed point $u_d^\star$ of $T_\Theta(\cdot\,; d)$ and then map $u_d^\star$ to the inference space $\mathcal{Y}$ via a final mapping $S_\Theta : \mathcal{U} \to \mathcal{Y}$. This enables us to define an implicit network $\mathcal{N}_\Theta$ by

$$\mathcal{N}_\Theta(d) \triangleq S_\Theta(u_d^\star) \quad \text{where} \quad u_d^\star = T_\Theta(u_d^\star; d). \tag{7}$$

[2]We use the same subscript for all terms, noting each operator typically depends on a portion of the weights.

[3]Each space is assumed to be a real-valued finite dimensional Hilbert space (*e.g.* $\mathbb{R}^n$) endowed with a product $\langle \cdot, \cdot \rangle$ and norm $\|\cdot\|$. It will be clear from context which space is being used.

Algorithm 1: Implicit Network with Fixed Point Iteration

```
1: $\mathcal{N}_\Theta(d)$:                                ◁ Input data is $d$
2:     $u^1 \leftarrow \hat{u}$                               ◁ Assign latent term
3:     while $\|u^k - T_\Theta(u^k; d)\| > \varepsilon$    ◁ Loop til converge
4:         $u^{k+1} \leftarrow T_\Theta(u^k; d)$                 ◁ Refine latent term
5:         $k \leftarrow k+1$                             ◁ Increment counter
6: return $S_\Theta(u^k)$                              ◁ Output estimate
```

Implementation considerations for T_Θ are discussed below. We also introduce assumptions on T_Θ that yield sufficient conditions to use the simple procedure in Algorithm 1 to approximate $\mathcal{N}_\Theta(d)$. In this algorithm, the latent variable initialization $\hat{u}$ can be any fixed quantity (*e.g.* the zero vector). The inequality in Step 3 gives a fixed point residual condition that measures convergence. Step 4 implements a fixed point update. The estimate of the inference $\mathcal{N}_\Theta(d)$ is computed by applying S_Θ to the latent variable u^k in Step 6. The blue path in Figure 1 visually summarizes Algorithm 1.

Convergence Finitely many loops in Steps 3 and 4 of Algorithm 1 is guaranteed by a classic functional analysis result (Banach 1922). This approach is used by several implicit networks (Ghaoui et al. 2019; Winston and Kolter 2020; Jeon, Lee, and Choi 2021). Below we present a variation of Banach's result for our setting.

Assumption 2.1. *The mapping T_Θ is L-Lipschitz with respect to its inputs (u, d), i.e. ,*

$$\|T_\Theta(u;\ d) - T_\Theta(v;\ w)\| \leq L\|(u,d) - (v,w)\|, \tag{8}$$

for all $(u,d),(v,w) \in \mathcal{U} \times \mathcal{D}$. Holding d fixed, the operator $T_\Theta(\cdot; d)$ is a contraction, i.e. there exists $\gamma \in [0,1)$ such that

$$\|T_\Theta(u;d) - T_\Theta(v;d)\| \leq \gamma\|u - v\|, \quad \textit{for all } u, v \in \mathcal{U}. \tag{9}$$

Remark 2.1. *The L-Lipschitz condition on T_Θ is used since recent works show Lipschitz continuity with respect to inputs improves generalization (Sokolić et al. 2017; Gouk et al. 2021; Finlay et al. 2018) and adversarial robustness (Cisse et al. 2017; Anil, Lucas, and Grosse 2019).*

Theorem 2.1. (BANACH) *For any $u^1 \in \mathcal{U}$, if the sequence $\{u^k\}$ is generated via the update relation*

$$u^{k+1} = T_\Theta(u^k;\ d), \quad \textit{for all } k \in \mathbb{N}, \tag{10}$$

and if Assumption 2.1 holds, then $\{u^k\}$ converges linearly to the unique fixed point $u_d^\star$ of $T_\Theta(\cdot; d)$.

Alternative Approaches In (Bai, Kolter, and Koltun 2019; Bai, Koltun, and Kolter 2020) Broyden's method is used for finding $u_d^\star$. Broyden's method is a quasi-Newton scheme and so at each iteration it updates a stored approximation to the Jacobian J_k and then solves a linear system in J_k. Since in this work our goal is to explore truly *Jacobian-free* approaches, we stick to the simpler fixed point iteration scheme when computing $\tilde{u}$ (*i.e.* Algorithm 1). In the contemporaneous (Gilton, Ongie, and Willett 2021), it is reported that using fixed point iteration in conjunction with Anderson acceleration finds $\tilde{u}$ faster than both vanilla fixed point iteration and Broyden's method. Combining JFB with Anderson accelerated fixed point iteration is a promising research direction we leave for future work.

Other Implicit Formulations A related implicit learning formulation is the well-known neural ODE model (Chen et al. 2018; Dupont, Doucet, and Teh 2019; Ruthotto and Haber 2021). Neural ODEs leverage known connections between deep residual models and discretizations of differential equations (Haber and Ruthotto 2017; Weinan 2017; Ruthotto and Haber 2019; Chang et al. 2018; Finlay et al. 2020; Lu et al. 2018), and replace these discretizations by black-box ODE solvers in forward and backward passes. The implicit property of these models arise from their method for computing gradients. Rather than backpropagate through each layer, backpropagation is instead done by solving the adjoint equation (Jameson 1988) using a black-box ODE solver as well. This is analogous to solving the Jacobian-based equation when performing backpropagation for implicit networks (see (13)) and allows the user to alleviate the memory costs of backpropagation through deep neural models by solving the adjoint equation at additional computational costs. A drawback is that the adjoint equation must be solved to high-accuracy; otherwise, a descent direction is not necessarily guaranteed (Gholami, Keutzer, and Biros 2019; Onken and Ruthotto 2020; Onken et al. 2021).

3 Backpropagation

We present a simple way to backpropagate with implicit networks, called Jacobian-free backprop (JFB). Traditional backpropagation will *not* work effectively for implicit networks since forward propagation during training could entail hundreds or thousands of iterations, requiring ever growing memory to store computational graphs. On the other hand, implicit models maintain fixed memory costs by backpropagating "through the fixed point" and solving a Jacobian-based equation (at potentially substantial added computational costs). The key step to circumvent this Jacobian-based equation with JFB is to tune weights by using a preconditioned gradient. Let $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ be a smooth loss function, denoted by $\ell(x,y)$, and consider the training problem

$$\min_\Theta \mathbb{E}_{d\sim\mathcal{D}}\big[\ell\left(y_d, \mathcal{N}_\Theta(d)\right)\big], \tag{11}$$

where we abusively write $\mathcal{D}$ to also mean a distribution. For clarity of presentation, in the remainder of this section we notationally suppress the dependencies on weights Θ by letting $u_d^\star$ denote the fixed point in (7). Unless noted otherwise, mapping arguments are implicit in this section; in each implicit case, this will correspond to entries in (7). We begin with standard assumptions enabling us to differentiate $\mathcal{N}_\Theta$.

Assumption 3.1. *The mappings S_Θ and T_Θ are continuously differentiable with respect to u and Θ.*

Assumption 3.2. *The weights Θ may be written as a tuple $\Theta = (\theta_S, \theta_T)$ such that weight paramaterization of S_Θ and T_Θ depend only on θ_S and θ_T, respectively.*[4]

[4]This assumption is easy to ensure in practice. For notational brevity, we use the subscript Θ throughout.

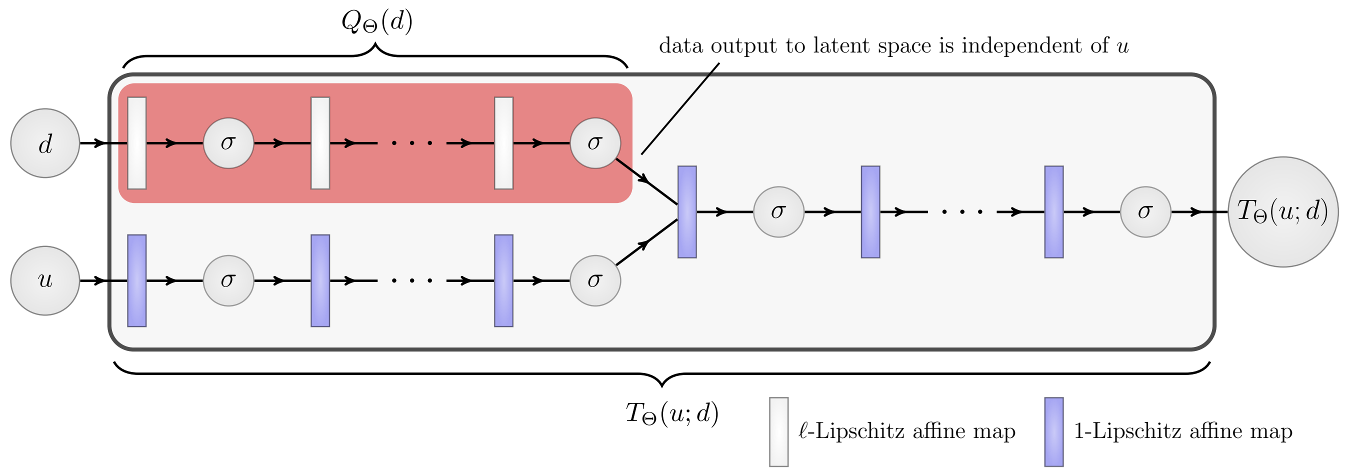


Figure 2: Diagram of a possible architecture for network operator T_Θ (in large rectangle). Data d and latent u variables are processed in two streams by nonlinearities (denoted by σ) and affine mappings (denoted by rectangles). These streams merge into a final stream that may also contain transformations. Light gray and blue affine maps are ℓ-Lipschitz and 1-Lipschitz, respectively. The mapping Q_Θ from data space to latent space is enclosed by the red rectangle.

Let $\mathcal{J}_\Theta$ be defined as the identity operator, denoted by I, minus the Jacobian[5] of T_Θ at (u, d), *i.e.*

$$\mathcal{J}_\Theta(u;d) \triangleq \mathrm{I} - \frac{\mathrm{d}T_\Theta}{\mathrm{d}u}(u;d). \tag{12}$$

Following (Winston and Kolter 2020; Bai, Kolter, and Koltun 2019), we differentiate both sides of the fixed point relation in (7) to obtain, by the implicit function theorem,

$$\frac{\mathrm{d}u_d^\star}{\mathrm{d}\Theta} = \frac{\partial T_\Theta}{\partial u}\frac{\mathrm{d}u_d^\star}{\mathrm{d}\Theta} + \frac{\partial T_\Theta}{\partial \Theta} \implies \frac{\mathrm{d}u_d^\star}{\mathrm{d}\Theta} = \mathcal{J}_\Theta^{-1}\cdot\frac{\partial T_\Theta}{\partial \Theta}, \tag{13}$$

where $\mathcal{J}_\Theta^{-1}$ exists whenever $\mathcal{J}_\Theta$ exists (see Lemma A.1). Using the chain rule gives the loss gradient

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\Theta}\left[\ell(y_d, \mathcal{N}_\Theta(d))\right] &= \frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u_d^\star, d))\Big] \\ &= \frac{\partial \ell}{\partial y}\left[\frac{\mathrm{d}S_\Theta}{\mathrm{d}u}\mathcal{J}_\Theta^{-1}\frac{\partial T_\Theta}{\partial\Theta} + \frac{\partial S_\Theta}{\partial \Theta}\right].\end{aligned} \tag{14}$$

The matrix $\mathcal{J}_\Theta$ satisfies the inequality (see Lemma A.1)

$$\left\langle u, \mathcal{J}_\Theta^{-1} u\right\rangle \geq \frac{1-\gamma}{(1+\gamma)^2}\|u\|^2, \quad \text{for all } u \in \mathcal{U}. \tag{15}$$

Intuitively, this coercivity property makes it seem possible to remove $\mathcal{J}_\Theta^{-1}$ from (14) and backpropagate using

$$\begin{aligned}p_\Theta &\triangleq -\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=u_d^\star} \\ &= -\frac{\partial \ell}{\partial y}\left[\frac{\mathrm{d}S_\Theta}{\mathrm{d}u}\frac{\partial T_\Theta}{\partial\Theta} + \frac{\partial S_\Theta}{\partial\Theta}\right].\end{aligned} \tag{16}$$

The omission of $\mathcal{J}_\Theta^{-1}$ admits two straightforward interpretations. Note $\mathcal{N}_\Theta(d) = S_\Theta(T_\Theta(u_d^\star; d))$, and so p_Θ is precisely the gradient of the expression $\ell(y_d, S_\Theta(T_\Theta(u_d^\star; d)))$, treating $u_d^\star$ as a constant *independent* of Θ. The distinction is that using $S_\Theta(T_\Theta(u_d^\star; d))$ assumes, perhaps by chance, the user chose the first iterate u^1 in their fixed point iteration (see Algorithm 1) to be precisely the fixed point $u_d^\star$. This makes the iteration trivial, "converging" in one iteration. We can simulate this behavior by using the fixed point iteration to find $u_d^\star$ and only backpropagating through the final step of the fixed point iteration, as shown in Figure 4.

Since the weights Θ typically lie in a space of much higher dimension than the latent space $\mathcal{U}$, the Jacobians $\partial S_\Theta/\partial\Theta$ and $\partial T_\Theta/\partial\Theta$ effectively always have full column rank. We leverage this fact via the following assumption.

Assumption 3.3. *Under Assumption 3.2, given any weights $\Theta = (\theta_S, \theta_T)$ and data d, the matrix*

$$M \triangleq \begin{bmatrix} \frac{\partial S_\Theta}{\partial \theta_S} & 0 \\ 0 & \frac{\partial T_\Theta}{\partial \theta_T}\end{bmatrix} \tag{17}$$

has full column rank and is sufficiently well conditioned to satisfy the inequality[6]

$$\kappa(M^\top M) = \frac{\lambda_{\max}(M^\top M)}{\lambda_{\min}(M^\top M)} \leq \frac{1}{\gamma}. \tag{18}$$

Remark 3.1. *The conditioning portion of the above assumption is useful for bounding the worst-case behavior in our analysis. However, we found it unnecessary to enforce this in our experiments for effective training (e.g. see Figure 5), which we hypothesize is justified because worst case behavior rarely occurs in practice and we train using averages of p_Θ for samples drawn from large data sets.*

Assumption 3.3 gives rise to a second interpretation of JFB. Namely, the full column rank of M enables us to

[5]Under Assumption 2.1, the Jacobian $\mathcal{J}_\Theta$ exists almost everywhere. However, presentation is cleaner by assuming smoothness.

[6]The term γ here refers to the contraction factor in (9).

rewrite p_Θ as a preconditioned gradient, *i.e.*

$$p_\Theta = \underbrace{\left(M \begin{bmatrix} \mathrm{I} & 0 \\ 0 & \mathcal{J}_\Theta \end{bmatrix} M^+ \right)}_{\text{preconditioning term}} \frac{\mathrm{d}\ell}{\mathrm{d}\Theta}, \tag{19}$$

where M^+ is the Moore-Penrose pseudo inverse (Moore 1920; Penrose 1955). These insights lead to our main result.

Theorem 3.1. *If Assumptions 2.1, 3.1, 3.2, and 3.3 hold for given weights Θ and data d, then*

$$p_\Theta \triangleq -\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=u_d^\star} \tag{20}$$

is a descent direction for $\ell(y_d, \mathcal{N}_\Theta(d))$ with respect to Θ.

Theorem 3.1 shows we can avoid difficult computations associated with $\mathcal{J}_\Theta^{-1}$ in (14) (*i.e.* solving an associated linear system/adjoint equation) in implicit network literature (Chen et al. 2018; Dupont, Doucet, and Teh 2019; Bai, Kolter, and Koltun 2019; Winston and Kolter 2020). Thus, our scheme more naturally applies to general multilayered T_Θ and is substantially simpler to code. Our scheme is juxtaposed in Figure 4 with classic and Jacobian-based schemes.

Two additional considerations must be made when determining the efficacy of training a model using (20) rather than Jacobian-based gradients (14).

- ► Does use of p_Θ in (20) degrade training/testing performance relative to (14)?
- ► Is the term p_Θ in (20) resilient to errors in estimates of the fixed point $u_d^\star$?

The first answer is our training scheme takes a different path to minimizers than using gradients with the implicit model. Thus, for nonconvex problems, one should not expect the results to be the same. In our experiments in Section 4, using (20) is competitive (14) for all tests (when applied to nearly identical models). The second inquiry is partly answered by the corollary below, which states JFB yields descent even for approximate fixed points.

Corollary 3.1. *Given weights Θ and data d, there exists $\varepsilon > 0$ such that if $u_d^\varepsilon \in \mathcal{U}$ satisfies $\|u_d^\varepsilon - u_d^\star\| \leq \varepsilon$ and the assumptions of Theorem 3.1 hold, then*

$$p_\Theta^\varepsilon \triangleq -\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=u_d^\varepsilon} \tag{21}$$

is a descent direction of $\ell(y_d, \mathcal{N}_\Theta(u_d^\star, d))$ with respect to Θ.

We are not aware of any analogous results for error tolerances in the implicit depth literature.

Coding Backpropagation A key feature of JFB is its simplicity of implementation. In particular, the backpropagation of our scheme is similar to that of a standard backpropagation. We illustrate this in the sample of PyTorch (Paszke et al. 2017) code in Figure 3. Here `explicit_model` represents $S_\Theta(T_\Theta(u;d))$. The fixed point $u_d^\star =$ `u_fxd_pt` is computed by successively applying T_Θ (see Algorithm 1) within a `torch.no_grad()` block. With this fixed point, `explicit_model` evaluates and returns $S_\Theta(T_\Theta(u_d^\star, d))$ to `y` in `train` mode (to create the computational graph). Thus, our scheme coincides with standard backpropagation through an explicit model with *one* latent space layer. On the other hand, standard implicit models backpropagate by solving a linear system to apply $\mathcal{J}_\Theta^{-1}$ as in (14). That approach requires users to manually update the parameters, use more computational resources, and make considerations (*e.g.* conditioning of $\mathcal{J}_\Theta^{-1}$) for each architecture used.

Implicit Forward + Proposed Backprop

```
u_fxd_pt = find_fixed_point(d)
y = explicit_model(u_fxd_pt, d)
loss = criterion(y, labels)
loss.backward()
optimizer.step()
```

Figure 3: Sample PyTorch code for backpropagation

Neumann Backpropagation The inverse Jacobian in (12) can be expanded using a Neumann series, *i.e.*

$$\mathcal{J}_\Theta^{-1} = \left(\mathrm{I} - \frac{\mathrm{d}T_\Theta}{\mathrm{d}u}\right)^{-1} = \sum_{k=0}^{\infty} \left(\frac{\mathrm{d}T_\Theta}{\mathrm{d}u}\right)^k. \tag{22}$$

Thus, JFB is a zeroth order approximation to the Neumann series. In particular, JFB resembles the Neumann-RBP approach for recurrent networks (Liao et al. 2018). However, Neumann-RBP does not guarantee a descent direction or guidelines on how to truncate the Neumann series. This is generally difficult to achieve in theory and practice (Aicher, Foti, and Fox 2020). Our work differs from (Liao et al. 2018) in that we focus purely on implicit networks, prove descent guarantees for JFB, and provide simple PyTorch implementations. Similar approaches exist in hyperparameter optimization, where truncated Neumann series are is used to approximate second order updates during training (Luketina et al. 2016; Lorraine, Vicol, and Duvenaud 2020).

4 Experiments

This section shows the effectiveness of JFB using PyTorch (Paszke et al. 2017). All networks are ResNet-based such that Assumption 3.2 holds.[7] One can ensure Assumption 2.1 holds (*e.g.* via spectral normalization). Yet, in our experiments we found this unnecessary since tuning the weights automatically encouraged contractive behavior.[8] All experiments are run on a single NVIDIA TITAN X GPU with 12GB RAM. Further details are in Appendix E.

[7] A weaker version of Assumption 3.1 also holds in practice, *i.e.* differentiability almost everywhere.

[8] We found (9) held for batches of data during training, even when using batch normalization. See Appendix E for more details.

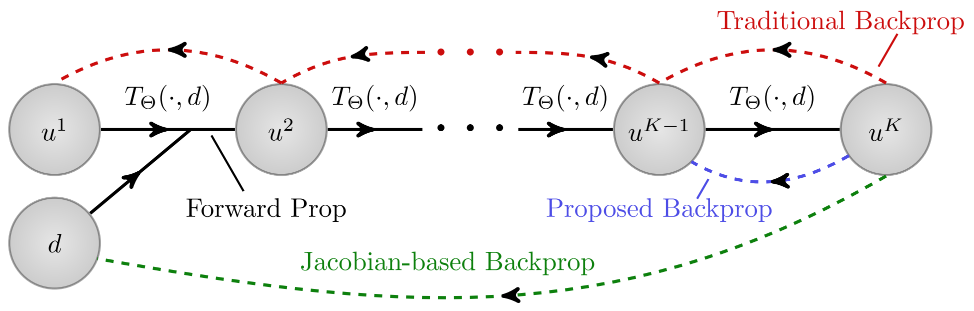


Figure 4: Diagram of backpropagation schemes for recurrent implicit depth models. Forward propagation is tracked via solid arrows point to the right (*n.b.* each forward step uses d). Backpropagation is shown via dashed arrows pointing to the left. Traditional backpropagation requires memory capacity proportional to depth (which is implausible for large K). Jacobian-based backpropagation solves an associated equation dependent upon the data d and operator T_Θ. JFB uses a single backward step, which avoids both large memory capacity requirements and solving a Jacobian-type equation.

MNIST

Method	Network size	Acc.
Explicit	54K	99.4%
Neural ODE[†]	84K	96.4%
Aug. Neural ODE[†]	84K	98.2%
MON [‡]	84K	99.2%
JFB-trained Implicit ResNet (ours)	54K	**99.4%**

SVHN

Method	Network size	Acc.
Explicit	164K	93.7%
Neural ODE[†]	172K	81.0%
Aug. Neural ODE[†]	172K	83.5%
MON (Multi-tier lg)[‡]	170K	92.3%
JFB-trained Implicit ResNet (ours)	164K	**94.1%**

CIFAR-10

Method	Network size	Acc.
Explicit (ResNet-56)*	0.85M	93.0%
MON (Multi-tier lg)[‡]*	1.01M	89.7%
JFB-trained Implicit ResNet (ours)*	0.84M	**93.7%**
Multiscale DEQ*	10M	93.8%

Table 1: Test accuracy of JFB-trained Implicit ResNet compared to Neural ODEs, Augmented NODEs, and MONs; [†]as reported in (Dupont, Doucet, and Teh 2019); [‡]as reported in (Winston and Kolter 2020); *with data augmentation

Classification

We train implicit networks on three benchmark image classification datasets licensed under CC-BY-SA: SVHN (Netzer et al. 2011), MNIST (LeCun, Cortes, and Burges 2010), and CIFAR-10 (Krizhevsky and Hinton 2009). Table 1 compares our results with state-of-the-art results for implicit networks, including Neural ODEs (Chen et al. 2018), Augmented Neural ODEs (Dupont, Doucet, and Teh 2019), Multiscale DEQs (Bai, Koltun, and Kolter 2020), and MONs (Winston and Kolter 2020). We also compare with corresponding explicit versions of our ResNet-based networks given in (1) as well as with state-of-the-art ResNet results (He et al. 2016) on the augmented CIFAR10 dataset. The explicit networks are trained with the same setup as their implicit counterparts. Table 1 shows JFBs are an effective way to train implicit networks, substantially outperform all the ODE-based networks as well as MONs using similar or fewer parameters. Moreover, JFB is competitive with Multiscale DEQs (Bai, Koltun, and Kolter 2020) despite having less than a tenth as many parameters. Appendix B contains additional results.

Comparison to Jacobian-based Backpropagation

Table 2 compares performance between using the standard Jacobian-based backpropagation and JFB. The experiments are performed on all the datasets described in Section 4. To apply the Jacobian-based backpropagation in (13), we use the conjugate gradient (CG) method on an associated set of normal equations similarly to (Liao et al. 2018). To maintain similar costs, we set the maximum number of CG iterations to be the same as the maximum depth of the forward propagation. The remaining experimental settings are kept the same as those from our proposed approach (and are therefore not tuned to the best of our ability). Note the network architectures trained with JFB contain batch normalization in the latent space whereas those trained with Jacobian-based backpropagation do not. Removal of batch normalization for the Jacobian-based method was necessary due to a lack of convergence when solving (13), thereby increasing training loss (see Appendix E for further details). This phenomena is also observed in previous works (Bai, Koltun, and Kolter 2020; Bai, Kolter, and Koltun 2019). Thus, we find JFB to be (empirically) effective on a wider class of network architectures (*e.g.* including batch normalization). The main purpose of the Jacobian-based results in Figure 5 and Table 2 is to show speedups in training time while maintaining a competitive accuracy with previous state-of-the-art implicit networks. More plots are given in Appendix B.

	Dataset	Avg time per epoch (s)	# of $\mathcal{J}$ mat-vec products	Accuracy %
Jacobian based	MNIST	28.4	6.0×10^6	99.2
	SVHN	92.8	1.4×10^7	90.1
	CIFAR10	530.9	9.7×10^8	87.9
JFB	MNIST	17.6	0	99.4
	SVHN	36.9	0	94.1
	CIFAR10	146.6	0	93.67

Table 2: Comparison of Jacobian-based backpropagation (first three rows) and our proposed JFB approach. "Mat-vecs" denotes matrix-vector products.

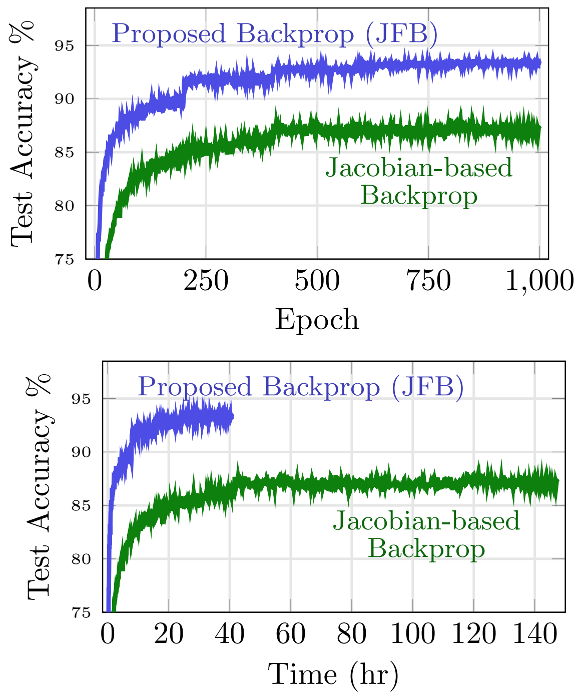


Figure 5: CIFAR10 results using comparable networks/-configurations, but with two backpropagation schemes: our proposed JFB method (blue) and standard Jacobian-based backpropagation in (14) (green), with fixed point tolerance $\epsilon = 10^{-4}$. JFB is faster and gives better test accuracy.

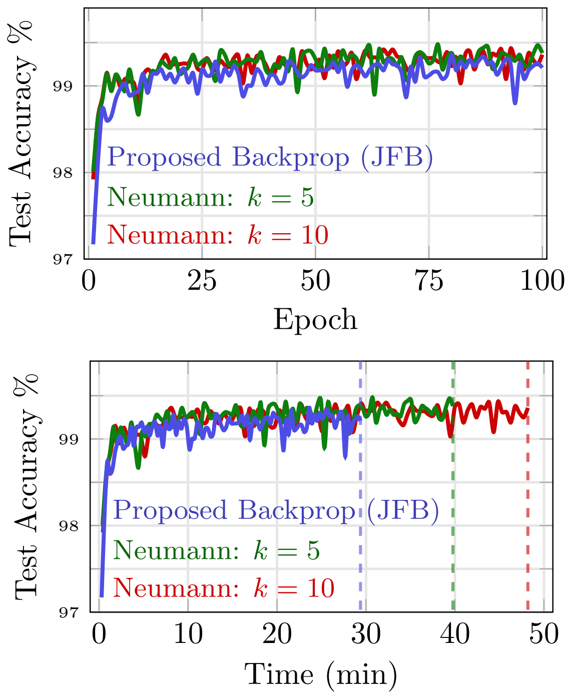


Figure 6: MNIST training using different truncations k of the Neumann series (22) to approximate the inverse Jacobian $\mathcal{J}_\Theta^{-1}$. Plots show faster training with fewer terms (fastest with JFB, *i.e.* $k = 0$) and competitive test accuracy.

Higher Order Neumann Approximation

As explained in Section 3, JFB can be interpreted as an approximation to the Jacobian-based approach by using a truncated series expansion. In particular, JFB is the zeroth order (*i.e.* $k = 0$) truncation to the Neumann series expansion (22) of the Jacobian inverse $\mathcal{J}_\Theta^{-1}$. In Figure 6, we compare JFB with training that uses more Neumann series terms in the approximation of the the Jacobian inverse $\mathcal{J}_\Theta^{-1}$. Figure 6 shows JFB is competitive at reduced time cost. More significantly, JFB is also much easier to implement as shown in Figure 3. An additional experiment with SVHN data and discussion about code are provided in Appendix D.

5 Conclusion

This work presents a new and simple Jacobian-free backpropagation (JFB) scheme. JFB enables training of implicit networks with fixed memory costs (regardless of depth), is easy to code (see Figure 3), and yields efficient backpropagation (by removing computations to do linear solves at each step). Use of JFB is theoretically justified (even when fixed points are approximately computed). Our experiments show JFB yields competitive results for implicit networks. Extensions will enable satisfaction of additional constraints for imaging and phase retrieval (Klibanov 1986; Fienup 1982; Heaton et al. 2020; Fung and Wendy 2020; Kan, Fung, and Ruthotto 2020), geophysics (Haber 2014; Fung and Ruthotto 2019a,b), and games (Von Neumann 1959; Lin et al. 2020; Li et al.; Ruthotto et al. 2020). Future work will analyze our proposed JFB in stochastic settings.

References

Abel, N. H. 1826. Démonstration de l'impossibilité de la résolution algébrique des équations générales qui passent le quatrieme degré. *Journal für die reine und angewandte Mathematik*, 1: 65–96.

Aicher, C.; Foti, N. J.; and Fox, E. B. 2020. Adaptively truncating backpropagation through time to control gradient bias. In *Uncertainty in Artificial Intelligence*, 799–808. PMLR.

Anil, C.; Lucas, J.; and Grosse, R. 2019. Sorting out Lipschitz function approximation. In *International Conference on Machine Learning*, 291–301. PMLR.

Bai, S.; Kolter, J. Z.; and Koltun, V. 2019. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, 690–701.

Bai, S.; Koltun, V.; and Kolter, J. Z. 2020. Multiscale Deep Equilibrium Models. *Advances in Neural Information Processing Systems*, 33.

Banach, S. 1922. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fund. math*, 3(1): 133–181.

Birkeland, R. 1927. Über die Auflösung algebraischer Gleichungen durch hypergeometrische Funktionen. *Mathematische Zeitschrift*, 26(1): 566–578.

Chang, B.; Meng, L.; Haber, E.; Ruthotto, L.; Begert, D.; and Holtham, E. 2018. Reversible architectures for arbitrarily deep residual neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Chen, R. T.; Rubanova, Y.; Bettencourt, J.; and Duvenaud, D. K. 2018. Neural ordinary differential equations. In *Advances in neural information processing systems*, 6571–6583.

Cisse, M.; Bojanowski, P.; Grave, E.; Dauphin, Y.; and Usunier, N. 2017. Parseval networks: Improving robustness to adversarial examples. In *International Conference on Machine Learning*, 854–863. PMLR.

Csáji, B. C.; et al. 2001. Approximation with artificial neural networks. *Faculty of Sciences, Eötvös Lorànd University, Hungary*, 24(48): 7.

Dupont, E.; Doucet, A.; and Teh, Y. W. 2019. Augmented Neural ODEs. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Elad, M.; Figueiredo, M. A.; and Ma, Y. 2010. On the role of sparse and redundant representations in image processing. *Proceedings of the IEEE*, 98(6): 972–982.

Fienup, J. R. 1982. Phase retrieval algorithms: A comparison. *Applied optics*, 21(15): 2758–2769.

Finlay, C.; Calder, J.; Abbasi, B.; and Oberman, A. 2018. Lipschitz regularized deep neural networks generalize and are adversarially robust. *arXiv preprint arXiv:1808.09540*.

Finlay, C.; Jacobsen, J.-H.; Nurbekyan, L.; and Oberman, A. M. 2020. How to train your neural ODE. *arXiv preprint arXiv:2002.02798*.

Fung, S. W.; and Ruthotto, L. 2019a. A multiscale method for model order reduction in PDE parameter estimation. *Journal of Computational and Applied Mathematics*, 350: 19–34.

Fung, S. W.; and Ruthotto, L. 2019b. An uncertainty-weighted asynchronous ADMM method for parallel PDE parameter estimation. *SIAM Journal on Scientific Computing*, 41(5): S129–S148.

Fung, S. W.; and Wendy, Z. 2020. Multigrid optimization for large-scale ptychographic phase retrieval. *SIAM Journal on Imaging Sciences*, 13(1): 214–233.

Ghaoui, L. E.; Gu, F.; Travacca, B.; Askari, A.; and Tsai, A. Y. 2019. Implicit Deep Learning. *arXiv preprint arXiv:1908.06315*.

Gholami, A.; Keutzer, K.; and Biros, G. 2019. ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs. *arXiv preprint arXiv:1902.10298*.

Gilton, D.; Ongie, G.; and Willett, R. 2021. Deep Equilibrium Architectures for Inverse Problems in Imaging. *arXiv preprint arXiv:2102.07944*.

Golub, G. H.; and Van Loan, C. F. 2013. *Matrix computations*, volume 3. JHU press.

Gouk, H.; Frank, E.; Pfahringer, B.; and Cree, M. J. 2021. Regularisation of neural networks by enforcing Lipschitz continuity. *Machine Learning*, 110(2): 393–416.

Gould, S.; Hartley, R.; and Campbell, D. 2019. Deep declarative networks: A new hope. *arXiv preprint arXiv:1909.04866*.

Haber, E. 2014. *Computational methods in geophysical electromagnetics*. SIAM.

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

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Heaton, H.; Fung, S. W.; Gibali, A.; and Yin, W. 2021a. Feasibility-based Fixed Point Networks. *arXiv preprint arXiv:2104.14090*.

Heaton, H.; Fung, S. W.; Lin, A. T.; Osher, S.; and Yin, W. 2020. Projecting to Manifolds via Unsupervised Learning. *arXiv preprint arXiv:2008.02200*.

Heaton, H.; McKenzie, D.; Li, Q.; Fung, S. W.; Osher, S.; and Yin, W. 2021b. Learn to Predict Equilibria via Fixed Point Networks. *arXiv preprint arXiv:2106.00906*.

Jameson, A. 1988. Aerodynamic design via control theory. *Journal of scientific computing*, 3(3): 233–260.

Jeon, Y.; Lee, M.; and Choi, J. Y. 2021. Differentiable Forward and Backward Fixed-Point Iteration Layers. *IEEE Access*.

Kan, K.; Fung, S. W.; and Ruthotto, L. 2020. PNKH-B: A projected Newton-Krylov method for large-scale bound-constrained optimization. *arXiv preprint arXiv:2005.13639*.

Kidger, P.; and Lyons, T. 2020. Universal approximation with deep narrow networks. In *Conference on Learning Theory*, 2306–2327. PMLR.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*.

Klibanov, M. V. 1986. Determination of a compactly supported function from the argument of its Fourier transform. In *Doklady Akademii Nauk*, volume 289, 539–540. Russian Academy of Sciences.

Kreyszig, E. 1978. *Introductory Functional Analysis with Applications*, volume 1. Wiley New York.

Krizhevsky, A.; and Hinton, G. 2009. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto.

Lawrence, N.; Loewen, P.; Forbes, M.; Backstrom, J.; and Gopaluni, B. 2020. Almost Surely Stable Deep Dynamics. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 18942–18953. Curran Associates, Inc.

LeCun, Y.; Cortes, C.; and Burges, C. 2010. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

Li, S.; Xie, Y.; Li, Q.; and Tang, G. ???? Cubic regularization for differentiable games. In *NeurIPS Workshop 2019*.

Liao, R.; Xiong, Y.; Fetaya, E.; Zhang, L.; Yoon, K.; Pitkow, X.; Urtasun, R.; and Zemel, R. 2018. Reviving and improving recurrent back-propagation. In *International Conference on Machine Learning*, 3082–3091. PMLR.

Lin, A. T.; Fung, S. W.; Li, W.; Nurbekyan, L.; and Osher, S. J. 2020. APAC-Net: Alternating the population and agent control via two neural networks to solve high-dimensional stochastic mean field games. *arXiv preprint arXiv:2002.10113*.

Look, A.; Doneva, S.; Kandemir, M.; Gemulla, R.; and Peters, J. 2020. Differentiable Implicit Layers. *arXiv preprint arXiv:2010.07078*.

Lorraine, J.; Vicol, P.; and Duvenaud, D. 2020. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, 1540–1552. PMLR.

Lu, Y.; Zhong, A.; Li, Q.; and Dong, B. 2018. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In *International Conference on Machine Learning*, 3276–3285. PMLR.

Lu, Z.; Pu, H.; Wang, F.; Hu, Z.; and Wang, L. 2017. The expressive power of neural networks: A view from the width. *arXiv preprint arXiv:1709.02540*.

Luketina, J.; Berglund, M.; Greff, K.; and Raiko, T. 2016. Scalable gradient-based tuning of continuous regularization hyperparameters. In *International conference on machine learning*, 2952–2960. PMLR.

Moore, E. H. 1920. On the reciprocal of the general algebraic matrix. *Bulletin of the American Mathematical Society*, 26: 394–395.

Netzer, Y.; Wang, T.; Coates, A.; Bissacco, A.; Wu, B.; and Ng, A. Y. 2011. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Onken, D.; and Ruthotto, L. 2020. Discretize-Optimize vs. Optimize-Discretize for Time-Series Regression and Continuous Normalizing Flows. *arXiv preprint arXiv:2005.13420*.

Onken, D.; Wu Fung, S.; Li, X.; and Ruthotto, L. 2021. OT-Flow: Fast and Accurate Continuous Normalizing Flows via Optimal Transport. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10): 9223–9232.

Osher, S.; Shi, Z.; and Zhu, W. 2017. Low dimensional manifold model for image processing. *SIAM Journal on Imaging Sciences*, 10(4): 1669–1690.

Ottem, J. 2011. Why are hypergeometric series important and do they have a geometric or heuristic motivation? https://mathoverflow.net/q/58089.

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, 406–413. Cambridge University Press.

Peyré, G. 2009. Manifold models for signals and images. *Computer vision and image understanding*, 113(2): 249–260.

Revay, M.; and Manchester, I. 2020. Contracting implicit recurrent neural networks: Stable models with improved trainability. In *Learning for Dynamics and Control*, 393–403. PMLR.

Ruthotto, L.; and Haber, E. 2019. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 1–13.

Ruthotto, L.; and Haber, E. 2021. An Introduction to Deep Generative Modeling. *arXiv preprint arXiv:2103.05180*.

Ruthotto, L.; Osher, S. J.; Li, W.; Nurbekyan, L.; and Fung, S. W. 2020. A machine learning framework for solving high-dimensional mean field game and mean field control problems. *Proceedings of the National Academy of Sciences*, 117(17): 9183–9193.

Sokolić, J.; Giryes, R.; Sapiro, G.; and Rodrigues, M. R. 2017. Robust large margin deep neural networks. *IEEE Transactions on Signal Processing*, 65(16): 4265–4280.

Udell, M.; and Townsend, A. 2019. Why are big data matrices approximately low rank? *SIAM Journal on Mathematics of Data Science*, 1(1): 144–160.

Van der Maaten, L.; and Hinton, G. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, 9(11).

Von Neumann, J. 1959. On the theory of games of strategy. *Contributions to the Theory of Games*, 4: 13–42.

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

Winston, E.; and Kolter, J. Z. 2020. Monotone operator equilibrium networks. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 10718–10728. Curran Associates, Inc.

Zhang, Q.; Gu, Y.; Mateusz, M.; Baktashmotlagh, M.; and Eriksson, A. 2020. Implicitly defined layers in neural networks. *arXiv preprint arXiv:2003.01822*.

Appendix

A Proofs

This section provides proofs for results of Section 3. For the reader's convenience, we restate all results before proving them.

Lemma A.1. *If Assumption 2.1 and 3.1 hold, then $\mathcal{J}_\Theta$ in (12) exists and*

$$\langle u, \mathcal{J}_\Theta u\rangle \geq (1-\gamma)\|u\|^2, \quad \textit{for all } u \in \mathcal{U}. \tag{23}$$

Additionally, $\mathcal{J}_\Theta$ is invertible, and its inverse $\mathcal{J}_\Theta^{-1}$ satisfies the coercivity inequality

$$\left\langle u, \mathcal{J}_\Theta^{-1} u\right\rangle \geq \frac{1-\gamma}{(1+\gamma)^2}\|u\|^2, \quad \textit{for all } u \in \mathcal{U}. \tag{24}$$

Proof. We proceed in the following manner. First we establish the coercivity inequality (23) (Step 1). This is used to show $\mathcal{J}_\Theta$ is invertible (Step 2). The previous two results are then combined to establish the inequality (24) (Step 3). All unproven results that are quoted below about operators are standard and may be found standard functional analysis texts (*e.g.* (Kreyszig 1978)).

Step 1. To obtain our coercivity inequality, we identify a bound on the operator norm for $\partial T_\Theta / \partial u$. Fix any unit vector $v \in \mathcal{U}$. Then, by the definition of differentiation,

$$\frac{\mathrm{d}T_\Theta}{\mathrm{d}u} v = \lim_{\varepsilon \to 0^+} \frac{T_\Theta(u^\star + \varepsilon v; d) - T_\Theta(u^\star; d)}{\|(u^\star + \varepsilon v) - u^\star\|} = \lim_{\varepsilon \to 0^+} \frac{T_\Theta(u^\star + \varepsilon v; d) - T_\Theta(u^\star; d)}{\varepsilon}. \tag{25}$$

Thus,

$$\left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u} v\right\| = \left\|\lim_{\varepsilon \to 0^+} \frac{T_\Theta(u^\star + \varepsilon v; d) - T_\Theta(u^\star; d)}{\varepsilon}\right\| = \lim_{\varepsilon \to 0^+} \frac{\|T_\Theta(u^\star + \varepsilon v; d) - T_\Theta(u^\star; d)\|}{\varepsilon}, \tag{26}$$

where the first equality follows from (25) and the second holds by the continuity of norms. Combining (27) with the Lipschitz assumption (9) gives the upper bound

$$\left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u} v\right\| \leq \lim_{\varepsilon \to 0^+} \frac{\gamma\|(u^\star + \varepsilon v) - u^\star\|}{\varepsilon} = \gamma. \tag{27}$$

Because the upper bound relation in (27) holds for an arbitrary unit vector $v \in \mathcal{U}$, we deduce

$$\left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u}\right\| \triangleq \sup\left\{\left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u} v\right\| \ : \ \|v\| = 1\right\} \leq \gamma. \tag{28}$$

That is, the operator norm is bounded by γ. Together the Cauchy-Schwarz inequality and (28) imply

$$\left\langle u, \frac{\mathrm{d}T_\Theta}{\mathrm{d}u} u\right\rangle \leq \left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u}\right\| \|u\|^2 \leq \gamma\|u\|^2, \quad \text{for all } u \in \mathcal{U}. \tag{29}$$

Thus, the bilinear form $\langle \cdot \, , \ \mathcal{J}_\Theta \cdot \rangle$ is $(1-\gamma)$ coercive, *i.e.*

$$\langle u, \mathcal{J}_\Theta u\rangle = \|u\|^2 - \left\langle u, \frac{\mathrm{d}T_\Theta}{\mathrm{d}u} u\right\rangle \geq (1-\gamma)\|u\|^2, \quad \text{for all } u \in \mathcal{U}. \tag{30}$$

Step 2. Consider any kernel element $w \in \ker(\mathcal{J}_\Theta)$. Then (30) implies

$$(1-\gamma)\|w\|^2 \leq \langle w, \mathcal{J}_\Theta w\rangle = \langle w, 0\rangle = 0 \quad \Longrightarrow \quad (1-\gamma)\|w\|^2 \leq 0 \quad \Longrightarrow \quad w = 0. \tag{31}$$

Consequently, the kernel of $\mathcal{J}_\Theta$ is trivial, *i.e.*

$$\ker(\mathcal{J}_\Theta) \triangleq \{u : \mathcal{J}_\Theta u = 0\} = \{0\}, \tag{32}$$

and wherefore the linear operator $\mathcal{J}_\Theta$ is invertible.

Step 3. By (27) and an elementary result in functional analysis,

$$\|\mathcal{J}_\Theta^\top \mathcal{J}_\Theta\| = \|\mathcal{J}_\Theta\|^2 \leq \left(\|\mathrm{I}\| + \left\|\frac{\mathrm{d}T_\Theta}{\mathrm{d}u}\right\|\right)^2 \leq (1+\gamma)^2. \tag{33}$$

Hence

$$\|u\|^2 = \langle u, u\rangle = \left\langle \mathcal{J}_\Theta^{-1} u, (\mathcal{J}_\Theta^\top \mathcal{J}_\Theta)\mathcal{J}_\Theta^{-1} u\right\rangle \leq (1+\gamma)^2 \left\|\mathcal{J}_\Theta^{-1} u\right\|^2, \quad \text{for all } u \in \mathcal{U}. \tag{34}$$

Combining (30) and (34) reveals

$$\frac{1-\gamma}{(1+\gamma)^2}\langle u, u\rangle \leq (1-\gamma)\|\mathcal{J}_\Theta^{-1} u\|^2 \leq \left\langle \mathcal{J}_\Theta^{-1} u, \mathcal{J}_\Theta(\mathcal{J}_\Theta^{-1} u)\right\rangle = \left\langle \mathcal{J}_\Theta^{-1} u, u\right\rangle, \quad \text{for all } u \in \mathcal{U}. \tag{35}$$

This establishes (24), and we are done. □

Lemma A.2. *If $A \in \mathbb{R}^{t\times t}$ is symmetric with positive eigenvalues,*

$$\overline{\lambda} \triangleq \frac{\lambda_{\max}(A) + \lambda_{\min}(A)}{2} \quad \textit{and} \quad S \triangleq \overline{\lambda}\mathrm{I} - A, \tag{36}$$

then

$$\|S\| = \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{2}. \tag{37}$$

Proof. Since A is symmetric, the spectral theorem asserts it possesses a set of eigenvectors that form an orthogonal basis for $\mathbb{R}^t$. This same basis forms the set of eigenvectors for $\overline{\lambda}\mathrm{I} - A$, with eigenvalues of A denoted by $\{\lambda_i\}_{i=1}^t$. So, there exists orthogonal $P \in \mathbb{R}^{t\times t}$ and diagonal Λ with entries given by each of the eigenvalues λ_i such that

$$S = \overline{\lambda}\mathrm{I} - P^\top \Lambda P = P^\top \left(\overline{\lambda}\mathrm{I} - \Lambda\right) P. \tag{38}$$

Substituting this equivalence into the definition of the operator norm yields

$$\|S\| \triangleq \sup\left\{\|S\xi\| : \ \|\xi\| = 1\right\} = \sup\left\{\|P^\top(\overline{\lambda}I - \Lambda)P\xi\| : \ \|\xi\| = 1\right\}. \tag{39}$$

Leveraging the fact P is orthogonal enables the supremum above to be restated via

$$\|S\| = \sup\left\{\|(\overline{\lambda}I - \Lambda)P\xi\| : \ \|\xi\| = 1\right\} = \sup\left\{\|(\overline{\lambda}I - \Lambda)\zeta\| : \ \|\zeta\| = 1\right\}. \tag{40}$$

Because $\overline{\lambda}\mathrm{I} - \Lambda$ is diagonal, (40) implies

$$\|S\| = \max_{i\in[t]} |\overline{\lambda} - \lambda_i| = \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{2}, \tag{41}$$

and the proof is complete. $\square$

Theorem 3.1. *If Assumptions 2.1, 3.1, 3.2, and 3.3 hold for given weights* Θ *and data* d*, then*

$$p_\Theta \triangleq -\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=u_d^\star} \tag{42}$$

forms a descent direction for $\ell(y_d, \mathcal{N}_\Theta(d))$ *with respect to* Θ*.*

Proof. To complete the proof, it suffices to show

$$\left\langle \frac{\mathrm{d}\ell}{\mathrm{d}\Theta}, p_\Theta \right\rangle < 0, \quad \text{for all } \frac{\mathrm{d}\ell}{\mathrm{d}\Theta} \neq 0. \tag{43}$$

Let any weights Θ and data d be given, and assume the gradient $\mathrm{d}\ell/\mathrm{d}\Theta$ is nonzero. We proceed in the following manner. First we show p_Θ is equivalent to a preconditioned gradient (Step 1). We then
show $M^\top \mathrm{d}\ell/\mathrm{d}\Theta$ is nonzero, with M as in (17) of Assumption 3.3 (Step 2). These two results are then combined to verify the descent inequality (43) for the provided Θ and d (Step 3).

Step 1. Denote the dimension of each component of the gradient $\mathrm{d}\ell/\mathrm{d}\Theta$ using[9]

$$\frac{\partial T_\Theta}{\partial \Theta} \in \mathbb{R}^{p\times n}, \quad \mathcal{J}_\Theta^{-1} \in \mathbb{R}^{n\times n}, \quad \frac{\partial S_\Theta}{\partial \Theta} \in \mathbb{R}^{p\times c}, \quad \frac{\mathrm{d}S_\Theta}{\mathrm{d}u} \in \mathbb{R}^{n\times c}, \quad \frac{\partial \ell}{\partial y} \in \mathbb{R}^{c\times 1}. \tag{44}$$

Combining each of these terms yields the gradient expression[10]

$$\frac{\mathrm{d}\ell}{\mathrm{d}\Theta} = \left[\frac{\partial T_\Theta}{\partial \Theta}\mathcal{J}_\Theta^{-1}\frac{\mathrm{d}S_\Theta}{\mathrm{d}u} + \frac{\mathrm{d}S_\Theta}{\mathrm{d}\Theta}\right]\frac{\partial \ell}{\partial y}. \tag{45}$$

By Assumption 3.2, S_Θ and T_Θ depend on separate components of $\Theta = (\theta_S, \theta_T)$. Thus,

$$\frac{\mathrm{d}\ell}{\mathrm{d}\Theta} = \begin{bmatrix} \frac{\partial S_\Theta}{\partial \theta_S} \\ \frac{\partial T_\Theta}{\partial \theta_T}\mathcal{J}_\Theta^{-1}\frac{\mathrm{d}S_\Theta}{\mathrm{d}u}\end{bmatrix}\frac{\partial \ell}{\partial y} = \underbrace{\begin{bmatrix} \frac{\partial S_\Theta}{\partial \theta_S} & 0 \\ 0 & \frac{\partial T_\Theta}{\partial \theta_T}\end{bmatrix}}_{M}\underbrace{\begin{bmatrix} \mathrm{I} & 0 \\ 0 & \mathcal{J}_\Theta^{-1}\end{bmatrix}}_{\tilde{\mathcal{J}}_\Theta^{-1}}\underbrace{\begin{bmatrix} \mathrm{I} \\ \frac{\mathrm{d}S_\Theta}{\mathrm{d}u}\end{bmatrix}\frac{\partial \ell}{\partial y}}_{v}, \tag{46}$$

where we define[11] $M \in \mathbb{R}^{p\times(n+c)}$, $\tilde{\mathcal{J}}_\Theta^{-1} \in \mathbb{R}^{(n+c)\times(n+c)}$, and $v \in \mathbb{R}^{(n+c)\times 1}$ to be the underbraced quantities. This enables the gradient to be concisely expressed via the relation

$$\frac{\mathrm{d}\ell}{\mathrm{d}\Theta} = M\tilde{\mathcal{J}}_\Theta^{-1}v, \tag{47}$$

and our proposed gradient alternative in (42) is given by

$$p_\Theta = -Mv. \tag{48}$$

Because M has full column rank (by Assumption 3.3), $M^+M = \mathrm{I}$, enabling us to rewrite p_Θ via

$$p_\Theta = -M\tilde{\mathcal{J}}_\Theta M^+ M\mathcal{J}_\Theta^{-1}v = -(M\tilde{\mathcal{J}}_\Theta M^+)\frac{\mathrm{d}\ell}{\mathrm{d}\Theta}. \tag{49}$$

Hence p_Θ is a preconditioned gradient (*n.b.* the preconditioner is not necessarily symmetric).

Step 2. Set

$$w \triangleq M^\top \frac{\mathrm{d}\ell}{\mathrm{d}\Theta} = M^\top M\tilde{\mathcal{J}}_\Theta^{-1}v. \tag{50}$$

The fact that M has full column rank implies it has a trivial kernel. In particular,

$$0 \neq \frac{\mathrm{d}\ell}{\mathrm{d}\Theta} = M\tilde{\mathcal{J}}_\Theta^{-1}v \implies 0 \neq \tilde{\mathcal{J}}_\Theta^{-1}v. \tag{51}$$

[9]We assumed each space is a real-valued finite dimensional Hilbert space, making it equivalent to some Euclidean space. So, it suffices to show everything in Euclidean spaces.

[10]In the main text, the ordering was used to make clear application of the chain rule, but here we reorder terms to get consistent dimensions in each matrix operation.

[11]Note this choice of M coincides with the matrix M in Assumption 3.3.

Again leveraging the full column rank of M, we know $M^\top M$ is invertible and, thus, has trivial kernel as well. This fact together with (51) reveals

$$0 \neq (M^\top M)\tilde{\mathcal{J}}_\Theta^{-1} v = w. \tag{52}$$

Step 3. Inserting the definition of w and p_Θ formulation of (49) into the scalar product in (43) yields

$$\left\langle \frac{\mathrm{d}\ell}{\mathrm{d}\Theta}, p_\Theta \right\rangle = -\left\langle M^\top M \tilde{\mathcal{J}}_\Theta^{-1} v, \tilde{\mathcal{J}}_\Theta M^+ M \tilde{\mathcal{J}}_\Theta^{-1} v \right\rangle = -\left\langle w, \tilde{\mathcal{J}}_\theta (M^\top M)^{-1} w \right\rangle, \tag{53}$$

noting $M^+ = (M^\top M)^{-1} M^\top$. Let λ_+ and λ_- be the maximum and minimum eigenvalues of $(M^\top M)^{-1}$, respectively. Note $(M^\top M)$ is positive definite since the full column rank of M implies

$$\langle \xi, M^\top M \xi \rangle = \|M\xi\|^2 > 0, \quad \text{for all nonzero } \xi \in \mathbb{R}^{n+c}. \tag{54}$$

Thus, $(M^\top M)^{-1}$ is positive definite, making $\lambda_+, \lambda_- > 0$. Let $\overline{\lambda}$ be the average of these terms, *i.e.*

$$\overline{\lambda} \triangleq \frac{\lambda_+ + \lambda_-}{2}. \tag{55}$$

Substituting in this choice of $\overline{\lambda}$ to (53) by adding and subtracting $\overline{\lambda}\mathrm{I}$ gives the inequality

$$-\left\langle w, \tilde{\mathcal{J}}_\theta (M^\top M)^{-1} w \right\rangle \leq -\overline{\lambda}(1-\gamma)\|w\|^2 + \left\langle w, \tilde{\mathcal{J}}_\Theta (\overline{\lambda}\mathrm{I} - (M^\top M)^{-1}) w \right\rangle, \tag{56}$$

noting $\tilde{\mathcal{J}}_\Theta$ is $1-\gamma$ coercive because it is the block diagonal composition of $\mathcal{J}_\Theta$, which is $1-\gamma$ coercive by (23) in Lemma A.1, and the identity matrix, which is 1-coercive. Application of the Cauchy Schwarz inequality to the right hand side of (56) reveals

$$-\left\langle w, \tilde{\mathcal{J}}_\theta (M^\top M)^{-1} w \right\rangle \leq -\overline{\lambda}(1-\gamma)\|w\|^2 + \|\tilde{\mathcal{J}}_\Theta\| \|\overline{\lambda}\mathrm{I} - (M^\top M)^{-1})\| \|w\|^2. \tag{57}$$

By Lemma A.2,

$$\|\overline{\lambda}\mathrm{I} - (M^\top M)^{-1}\| = \frac{\lambda_+ - \lambda_-}{2}. \tag{58}$$

Similar block diagonal argument as used above to verify $\tilde{\mathcal{J}}_\Theta$ is coercive can also be applied to bound the operator norm of $\tilde{\mathcal{J}}_\Theta$. Indeed, (28) implies

$$\|\mathcal{J}_\Theta\| \leq 1+\gamma \quad \Longrightarrow \quad \|\tilde{\mathcal{J}}_\Theta\| \leq 1+\gamma. \tag{59}$$

Hence (53), (57), (58), and (59) together yield

$$\left\langle \frac{\mathrm{d}\ell}{\mathrm{d}\Theta}, p_\Theta \right\rangle \leq -\frac{1}{2}\big((1-\gamma)(\lambda_+ + \lambda_-) - (1+\gamma)(\lambda_+ - \lambda_-)\big)\|w\|^2 = -2(\lambda_- - \gamma\lambda_+)\|w\|^2. \tag{60}$$

The right hand expression in (60) is negative since (52) shows $w \neq 0$ and the conditioning inequality (18) in Assumption 3.3 implies $(\lambda_- - \gamma\lambda_+)$ is positive. This verifies (43), completing the proof. □

Corollary 3.1. *Given weights Θ and data d, there exists $\varepsilon > 0$ such that if $u^\varepsilon \in \mathcal{U}$ satisfies $\|u_d^\varepsilon - u_d^\star\| \leq \varepsilon$ and the assumptions of Theorem 3.1 hold, then*

$$p_\Theta^\varepsilon \triangleq - - \frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=u_d^\varepsilon} \tag{61}$$

is a descent direction for the loss function $\ell(y_d, \mathcal{N}_\Theta(u_d^\star, d))$ with respect to Θ.

Proof. For notational convenience, for all $\tilde{u} \in \mathcal{U}$, define

$$p_\Theta(\tilde{u}) \triangleq -\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(T_\Theta(u,d))\Big]_{u=\tilde{u}} \tag{62}$$

noting $p_\Theta^\varepsilon = p_\Theta(u_d^\varepsilon)$. Also define the quantity

$$\nabla \triangleq \frac{\mathrm{d}}{\mathrm{d}\Theta}\left[\ell(y_d, \mathcal{N}_\Theta(d))\right]. \tag{63}$$

Assuming $\nabla \neq 0$, it suffices to show

$$\langle p_\Theta^\varepsilon, \nabla\rangle < 0. \tag{64}$$

By the smoothness of ℓ, S_Θ, and T_Θ (see Assumption 3.1), there exists $\delta > 0$ such that

$$\|u - u_d^\star\| \leq \delta \quad \Longrightarrow \quad \|p_\Theta(u) - p_\Theta(u_d^\star)\| \leq \frac{(\lambda_- - \gamma\lambda_+)\|M^\top \nabla\|^2}{\|\nabla\|}, \tag{65}$$

where λ_+ and λ_- are the maximum and minimum eigenvalues of $(M^\top M)^{-1}$, respectively. Also note $M^\top \nabla \neq 0$ since $M^\top$ has full column rank.[12] Substituting the inequality (60) in the proof of Theorem 3.1 into (64) reveals

$$\langle p_\Theta(u), \nabla\rangle = \langle p_\Theta(u_d^\star), \nabla\rangle + \langle p_\Theta(u) - p_\Theta(u_d^\star), \nabla\rangle \tag{66a}$$

$$\leq -2(\lambda_- - \gamma\lambda_+)\|M^\top \nabla\|^2 + \langle p_\Theta(u) - p_\Theta(u_d^\star), \nabla\rangle. \tag{66b}$$

But, the Cauchy Schwarz inequality and (65) enable us to obtain the upper bound

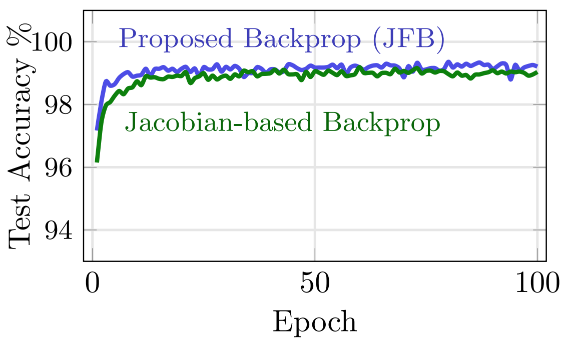
$$|\langle p_\Theta(u) - p_\Theta(u_d^\star), \nabla\rangle| \leq (\lambda_- - \gamma\lambda_+)\|M^\top \nabla\|^2, \quad \text{for all } u \in B(u_d^\star, \delta), \tag{67}$$

where $B(u_d^\star, \delta)$ is the ball of radius δ centered about $u_d^\star$. Combining (66) and (67) yields

$$\langle p_\Theta(u), \nabla\rangle \leq -(\lambda_- - \gamma\lambda_+)\|M^\top \nabla\|^2, \quad \text{for all } u \in B(u_d^\star, \delta). \tag{68}$$

In particular, this shows (64) holds when we set $\varepsilon = \delta$. ☐

B Classification Accuracy Plots



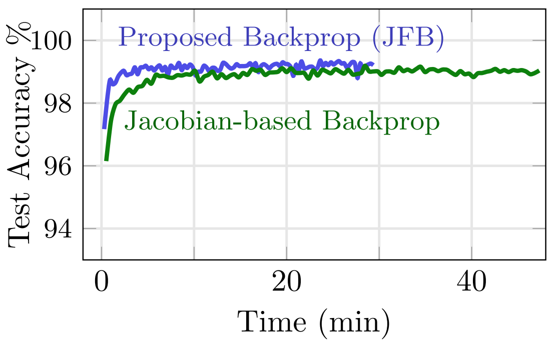


Figure 7: MNIST performance using nearly identical architectures/configurations, but with two backpropagation schemes: our proposed method (blue) and the standard Jacobian-based backpropagation in (14) (red), with fixed point tolerance $\epsilon = 10^{-4}$. The difference in the architecture/configurations comes from the use of batch normalization in the latent space when using JFB (see Appendix E for more details). Our method is faster and yields better test accuracy.

[12]See w in Step 2 of the proof of Theorem 3.1.

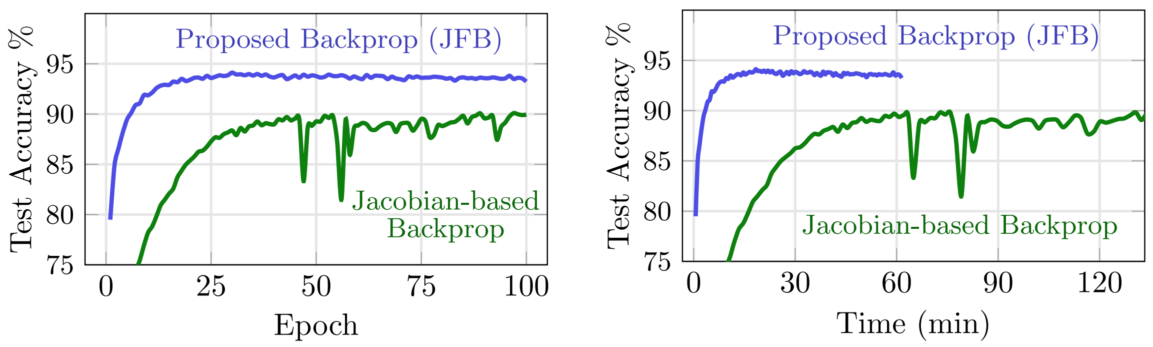


Figure 8: SVHN performance using identical architecture/configurations, but with two backpropagation schemes: our proposed method (blue) and the standard Jacobian-based backpropagation in (14) (red), with fixed point tolerance $\epsilon = 10^{-4}$. The difference in the architectures/configurations comes from the use of batch normalization in the latent space when using JFB (see Appendix E for more details). Our method is faster and yields better test accuracy.

C Implementation of Jacobian-based Backpropagation

Implementation Notes

In this section, we provide some notes to help understand the code/implementation of the Jacobian-based backpropagation in PyTorch. Assume we have the fixed point $\tilde{u}$ at hand. For brevity, we will omit the dependence of R_Θ and $\tilde{u}$ on d. We wish to compute

$$\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(R_\Theta(\tilde{u}_\Theta))\Big] = \frac{\mathrm{d}\ell}{\mathrm{d}S}\left[\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}\frac{\mathrm{d}\tilde{u}}{\mathrm{d}\Theta} + \frac{\partial S}{\partial \Theta}\right], \tag{69}$$

where

$$\frac{\mathrm{d}\tilde{u}}{\mathrm{d}\Theta} = \mathcal{J}^{-1}\frac{\mathrm{d}R_\Theta(\tilde{u})}{\mathrm{d}\Theta}, \tag{70}$$

and the argument $\tilde{u}$ inside of R is treated as a constant. This implies that

$$\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(R_\Theta(\tilde{u}_\Theta))\Big] = \frac{\mathrm{d}\ell}{\mathrm{d}S}\left[\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}\mathcal{J}^{-1}\frac{\mathrm{d}R_\Theta(\tilde{u})}{\mathrm{d}\Theta} + \frac{\partial S}{\partial \Theta}\right]. \tag{71}$$

In our PyTorch implementation, we do not build J^{-1} explicitly. Instead, we solve a linear system as follows. We would like to compute w defined by

$$w = \frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}\mathcal{J}^{-1}. \tag{72}$$

To do this, we solve the following linear system

$$w\mathcal{J} = \frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}} \tag{73}$$

Note, we consider multiplication by matrices from the right as this is more natural to implement in PyTorch. We also note that building the matrix $\mathcal{J}$ explicitly is inefficient, thus any matrix-factorization methods (*e.g.* the LU decomposition) cannot be used. As explained in Section 4, we use a CG method and require a symmetric coefficient matrix. To this end, we symmetrize the system by multiplying by $J^\top$ on both sides to obtain the normal equations (Golub and Van Loan 2013)

$$w\mathcal{J}\mathcal{J}^\top = \frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}\mathcal{J}^\top. \tag{74}$$

Once we solve for w, we can then arrive at the gradient by computing

$$\frac{\mathrm{d}}{\mathrm{d}\Theta}\Big[\ell(y_d, S_\Theta(R_\Theta(\tilde{u}_\Theta))\Big] = w\frac{\mathrm{d}R_\Theta(\tilde{u})}{\mathrm{d}\Theta} + \frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\partial S}{\partial \Theta}. \tag{75}$$

Coding right-hand-side

To code the right-hand-side of the normal equations, we can code $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}$ in the following line of code:

Computing $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}$

```
Qd = net.data_space_forward(d)
Ru = net.latent_space_forward(u, Qd)
S_Ru = net.map_latent_to_inference(Ru)
loss = criterion(S_Ru, labels)
dldu = torch.autograd.grad(outputs=loss, inputs=Ru,
                           retain_graph=True, create_graph=True,
                           only_inputs=True)[0]
```

Next, we would like to multiply `dldu` by $J^\top$ from the right side. To do this, we need to use a vector-Jacobian trick in Pytorch as follows:

Computing $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}J^\top$

```
dldu_dRdu = torch.autograd.grad(outputs=Ru, inputs = u, grad_outputs=dldu,
                                retain_graph=True, create_graph=True,
                                only_inputs=True)[0]
dldu_J = dldu - dldu_dRdu

dldu_JT = torch.autograd.grad(outputs=dldu_J, inputs=dldu, grad_outputs=dldu,
                              retain_graph=True, create_graph=True,
                              only_inputs=True)[0]

rhs = dldu_JT
```

Here, to multiply by $J^\top$ from the right, we note that for any vector v,

$$v\frac{\mathrm{d}(vJ)}{\mathrm{d}v} = vJ^\top. \tag{76}$$

The vector-Jacobian trick uses autograd once to compute vJ, and then autograd once more compute $vJ^\top$ as in Equation (76). Thus, we have that `rhs` takes the value of $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\mathrm{d}S}{\mathrm{d}\tilde{u}}J^\top$.

Coding right matrix-vector multiplication by $JJ^\top$

Next, we want to implement a function that computes right matrix-vector multiplication by $JJ^\top$. This function, along with the right-hand-side, is then fed into the conjugate gradient algorithm to solve Equation (74).

Given a vector v, the task is to return $vJJ^\top$. First, we use one autograd call to obtain vJ. Then we use another autograd call to multiply by $J^\top$ to obtain $vJJ^\top$. The function which multiplies by $JJ^\top$ from the right can thus be coded as

Computing multiplication by $JJ^\top$

```
v_dRdu = torch.autograd.grad(outputs=Ru, inputs=u, grad_outputs=v,
                             retain_graph=True, create_graph=True,
                             only_inputs=True)[0]
v_J = v - v_dRdu

v_JJT = torch.autograd.grad(outputs=v_J, inputs=v, grad_outputs=v_J,
                            retain_graph=True, create_graph=True,
                            only_inputs=True)[0]
```

We emphasize here that the third line returns $vJJ^\top$ by setting the variable `grad_outputs` to be vJ. Finally, we feed the computed right-hand-side and the function that multiplies by $JJ^\top$ into the conjugate gradient method to solve for w in Equation (74).

Coding $w\frac{\mathrm{d}R_\Theta(\tilde{u})}{\mathrm{d}\Theta}$ and $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\partial S}{\partial \Theta}$

Once w is obtained from the linear solve, we have two remaining tasks to obtain the gradient: computation of $w\frac{\mathrm{d}R_\Theta(\tilde{u})}{\mathrm{d}\Theta}$ and $\frac{\mathrm{d}\ell}{\mathrm{d}S}\frac{\partial S}{\partial \Theta}$. These can be computed as follows in the PyTorch framework. Suppose the solution to the normal equations is saved in the variable `normal_eq_sol`

Update gradients

```
Ru.backward(normal_eq_sol)
S_Ru = net.map_latent_to_inference(Ru.detach())
loss = criterion(S_Ru, labels)
loss.backward()
```

This is only one (perhaps the most straightforward) way to code the Jacobian-based backpropagation. But as can be seen, coding the Jacobian-based backpropagation is not trivial, unlike our proposed JFB.

D Comparison with Neumann RBP

Below is a comparison of JFB with 5th and 10th order Neumann series approximations of gradients for the SVHN dataset.

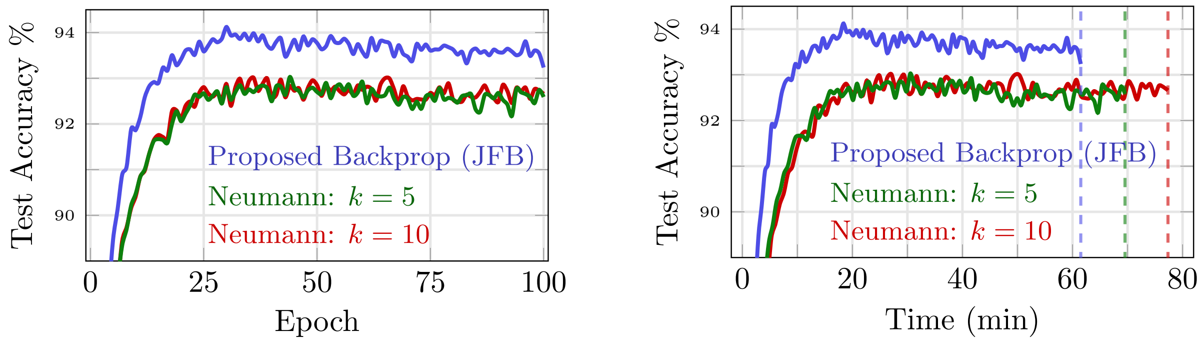


Figure 9: SVHN using different Neumann approximations of the inverse Jacobian.

Neumann Gradient Implementation

To compute the Neumann-based gradient, we use a similar approach that explained in Appendix C. In particular, we use a for-loop to accumulate the Neumann sum in the variable `dldu_Jinv_approx`.

Computing Neumann gradient

```
for i in range(1, neumann_order+1):

    dldu_dRdu_k.requires_grad = True

    # compute dldu_dRdu_k+1 = dldu_dRdu_k * dRdu
    dldu_dRdu_kplus1 = torch.autograd.grad(outputs=Ru,
                                           inputs=u,
                                           grad_outputs=dldu_dRdu_k,
                                           retain_graph=True,
                                           create_graph=True,
                                           only_inputs=True)[0]

    dldu_Jinv_approx = dldu_Jinv_approx + dldu_dRdu_kplus1.detach()

    dldu_dRdu_k = dldu_dRdu_kplus1.detach()

Ru.backward(dldu_Jinv_approx)
```

Similar to the Jacobian-based approach, we multiply from right by $\frac{\partial R}{\partial \Theta}$ from the right in the last line.

E Experimental Settings

We present the experimental settings and describe the architecture used for each dataset. We used ResNets with batch normalization in the latent space portion of the networks, *i.e.* , $R_\Theta(d)$. While batch normalization prevents us from completely guaranteeing the network is γ-contractive in its latent variable, we found the networks automatically behave in a contractive manner. Specifically, every time the network is evaluated during training, we check whether our network violates the γ-contractive property and print a warning when this is the case. This warning was never called in our experiments. As mentioned in (4), the Jacobian-based version failed to converge (even with tighter tolerance and more iterations) when batch normalization was present in the latent space - this is an issue also observed in other implicit networks literature (Bai, Koltun, and Kolter 2020). Consequently, we remove the batch normalization for the Jacobian-based runs. We train all of our networks with the Adam optimizer (Kingma and Ba 2015) and use the cross entropy loss function.

MNIST

We use two convolutions with leaky relu activation functions and max pooling for the data-space portion of the network $Q_\Theta(d)$. In the latent space portion, $R_\Theta(d)$, we use 2-layer ResNet-based architecture, with the ResNet block containing two convolution operators with batch normalization. Finally, we map from latent space to inference space using one convolution and one fully connected layer. For the fixed point stopping criterion, we stop whenever consecutive iterates satisfy $\|u^{k+1} - u^k\| < \epsilon = 10^{-4}$ or 50 iterations have occurred. We use a constant learning rate of 10^{-4}.

SVHN

We use three 1-layer ResNets with residual blocks containing two convolutions with leaky relu activation functions and max pooling for the data-space portion of the network $Q_\Theta(d)$. Similarly to the ResNet-based network in MNIST, we use a ResNet block containing two convolution operators with batch normalization in the latent space portion $R_\Theta(d)$. We map from latent space to inference space using one convolution and one fully connected layer. For the fixed point stopping criterion, we stop whenever consecutive iterates satisfy $\|u^{k+1} - u^k\| < \epsilon = 10^{-4}$ or 200 iterations have occurred. We use constant learning rate of 10^{-4} with weight decay of 2×10^{-4}.

CIFAR10

We use a ResNet with residual blocks containing two convolutions for the data-space portion of the network $Q_\Theta(d)$. We use a ResNet for the latent space portion of $R_\Theta(d)$, with each ResNet block containing two convolution operators with batch normalization. Approximately 70% of the weights are in Q_Θ and 30% of the weights are in R_Θ. We map from latent space to inference space using one convolution and one fully connected layer. For the JFB fixed point stopping criterion, we stop whenever consecutive iterates satisfy $\|u^{k+1} - u^k\| < \epsilon = 10^{-1}$ or 50 iterations have occurred. For the Jacobian-based approach, however, we observed that we needed to tighten the tolerance in order for the gradients to be computed accurately. Particularly, we stop whenever consecutive iterates satisfy $\|u^{k+1} - u^k\| < \epsilon = 10^{-4}$ or 500 iterations have occurred.

F Toy Implicit Example

This section provides rigorous justification of the toy example provided in Section 1 for solving $y = d + y^5$ with a given $d \in [-1/2, 1/2]$. See Figure 10 for an illustration. We first outline its implicit solution in the following lemma, which also establishes this equation has a unique solution in $[-10^{-1/4}, 10^{-1/4}]$. This is followed by a brief discussion of the explicit series representations of solutions to (4).

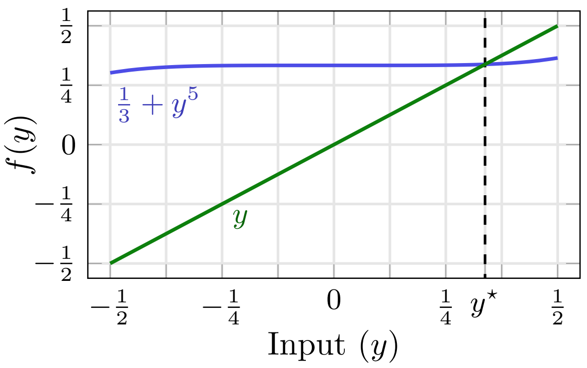


Figure 10: Plot of the functions y and $d + y^5$ with $d = 1/3$.

Lemma F.1. *Let $d \in [-1/2, 1/2]$. If the sequence $\{y_k\} \subset \mathbb{R}$ is defined such that $y_1 = 0$ and*

$$y_{k+1} = T(y_k; d) \triangleq d + y_k^5 \tag{77}$$

then $\{y_k\}$ converges to the unique fixed point of $T(\cdot; d)$ among $y \in [-10^{-1/4}, 10^{-1/4}]$.

Proof. We proceed in the following manner. First $T(\cdot; d)$ is shown to a contraction on a restricted subset of $\mathbb{R}$ (Step 1). Then we show $\{y_k\}$ is a subset of this restricted subset (Step 2). These two facts together enable us to obtain convergence (Step 3) and uniqueness (Step 4), using a special case of Banach's fixed point theorem (Banach 1922).

Step 1. Set $\alpha = 10^{-1/4}$ and let $y, \gamma \in [-\alpha, \alpha]$. By the mean value theorem, there exists ξ between y and γ such that

$$|y^5 - \gamma^5| = |T(y; d) - T(\gamma; d)| = \left| \frac{\mathrm{d}T}{\mathrm{d}u}(\xi; d) \right| |y - \gamma|. \tag{78}$$

Additionally,

$$\sup_{\xi \in [-\alpha, \alpha]} \left| \frac{\mathrm{d}T}{\mathrm{d}u}(\xi; d) \right| = \sup_{\xi \in [-\alpha, \alpha]} 5y^4 = 5\alpha^4 \le 5 \cdot \frac{1}{10} = \frac{1}{2}, \tag{79}$$

and so

$$|y^5 - \gamma^5| \le \frac{1}{2}|y - \gamma|. \tag{80}$$

Because y and γ were arbitrarily chosen in $[-\alpha, \alpha]$, it follows that the restriction of $T(\cdot; d)$ to $[-\alpha, \alpha]$ is a $\frac{1}{2}$-contraction.

Step 2. This step proceeds by induction. Note $y_1 = 0 \in [-\alpha, \alpha]$. Inductively, suppose $y_k \in \mathbb{N}$. This implies

$$|y_{k+1}| = |T(y_k; d)| = |d + y_k^5| \le |d| + |y_k|^5 \le \frac{1}{2} + \alpha^5 < \alpha, \tag{81}$$

and so $y_{k+1} \in [-\alpha, \alpha]$. By the principle of mathematical induction, we deduce $y_k \in [-\alpha, \alpha]$ for all $k \in \mathbb{N}$.

Step 3. We now establish convergence. Applying the results of Step 1 and Step 2 reveals

$$|y_{k+2} - y_{k+1}| = |T(y_{k+1}) - T(y_k)| \le \frac{1}{2}|y_{k+1} - y_k|, \quad \text{for all } k \in \mathbb{N}. \tag{82}$$

Applying this result inductively with the triangle inequality reveals $m > n$ implies

$$|y_m - y_n| \le \sum_{\ell=n}^{m-1} |y_{\ell+1} - y_\ell| \le \sum_{\ell=n}^{m-1} 2^{-\ell}|y_2 - y_1| \le 2^{-n}|y_2 - y_1| \cdot \sum_{\ell=0}^{\infty} 2^{-\ell} \le 2^{1-n}|y_2 - y_1|. \tag{83}$$

Since the right hand side in (83) converges to zero as $n \to \infty$, we see $\{y_k\}$ is Cauchy and, thus, converges to a limit y_∞. Moreover, the limit satisfies

$$y_\infty = \lim_{k \to \infty} y_k = \lim_{k \to \infty} T(y_k; d) = \lim_{k \to \infty} d + y_k^5 = d + y_\infty^5. \tag{84}$$

Step 4. All that remains it to verify the fixed point of $T(\cdot; d)$ is unique over $[-\alpha, \alpha]$. If a fixed point $\tilde{y} \in \text{fix}(T(\cdot; d))$ were to exist such that $\tilde{y} \in [-\alpha, \alpha] - \{y_\infty\}$, then the contractive property of $T(\cdot; d)$ may be applied to deduce

$$|y_\infty - \tilde{y}| = |T(y_\infty; d) - T(\tilde{y}; d)| \le \frac{1}{2}|y_\infty - \tilde{y}| \quad \Longrightarrow \quad 1 < \frac{1}{2}, \tag{85}$$

a contradiction. Hence the fixed point y_∞ is unique. □

Explicit Solution

As is well-known, the solution of a quintic equation cannot be expressed as a function of the coefficients using only the operations of addition, subtraction, multiplication, division and taking roots (Abel 1826). The simplest way to express the unique root to $y = d + y^5$ lying in the interval $[-10^{-1/4}, 10^{-1/4}]$ as a function of d is via a hypergeometric series by writing

$$y = d \left[{}_4F_3 \left(\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}; \frac{1}{2}, \frac{3}{4}, \frac{5}{4}; \frac{3125d^4}{256} \right) \right] = d + d^5 + 10\frac{d^9}{2!} + 210\frac{d^{13}}{3!} + \dots \tag{86}$$

See (Birkeland 1927) or (Ottem 2011) for further information on solving quintic equations using hypergeometric functions.