Scale-invariant Learning by Physics Inversion

Anonymous Author(s)
Affiliation
Address
email

Abstract

Solving inverse problems, such as parameter estimation and optimal control, is a vital part of science. Many experiments repeatedly collect data and rely on machine learning algorithms to quickly infer solutions to the associated inverse problems. We find that state-of-the-art training techniques are not well-suited to many problems that involve physical processes. The highly nonlinear behavior, common in physical processes, results in strongly varying gradients that lead first-order optimizers like SGD or Adam to compute suboptimal optimization directions. We propose a novel hybrid training approach that combines higher-order optimization methods with machine learning techniques. We take updates from a scale-invariant inverse problem solver and embed them into the gradient-descent-based learning pipeline, replacing the regular gradient of the physical process. We demonstrate the capabilities of our method on a variety of canonical physical systems, showing that it yields significant improvements on a wide range of optimization and learning problems.

1 Introduction

Inverse problems that involve physical systems play a central role in computational science. This class of problems includes parameter estimation [54] and optimal control [57]. Among others, solving inverse problems is integral in detecting gravitational waves [22], controlling plasma flows [38], searching for neutrinoless double-beta decay [3, 1], and testing general relativity [19, 35].

Decades of research in optimization have produced a wide range of iterative methods for solving inverse problems [47]. Higher-order methods such as limited-memory BFGS [36] have been especially successful. Such methods compute or approximate the Hessian of the optimization function in addition to the gradient, allowing them to locally invert the function and find stable optimization directions. Gradient descent, in contrast, only requires the first derivative but converges more slowly, especially in ill-conditioned settings [49].

Despite the success of iterative solvers, many of today's experiments rely on machine learning methods, and especially deep neural networks, to find unknown parameters given the observations [11, 15, 22, 3]. While learning methods typically cannot recover a solution up to machine precision, they have a number of advantages over iterative solvers. First, their computational cost for inferring a solution is usually much lower than with iterative methods. This is especially important in time-critical applications, such as the search for rare events in data sets comprising of billions of individual recordings for collider physics. Second, learning-based methods do not require an initial guess to solve a problem. With iterative solvers, a poor initial guess can prevent convergence to a global optimum or lead to divergence (see Appendix C.1). Third, learning-based solutions can be less prone to finding local optima than iterative methods because the parameters are shared across a large collection of problems [30]. Combined with the stochastic nature of the training process, this allows gradients from other problems to push a prediction out of the basin of attraction of a local optimum.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Practically all state-of-the-art neural networks are trained using only first order information, mostly due to the computational cost of evaluating the Hessian w.r.t. the network parameters. Despite the many breakthroughs in the field of deep learning, the fundamental shortcomings of gradient descent persist and are especially pronounced when optimizing non-linear functions such as physical systems. In such situations, the gradient magnitudes often vary strongly from example to example and parameter to parameter.

In this paper, we show that inverse physics solvers can be embedded into the traditional deep learning pipeline, resulting in a hybrid training scheme that aims to combine the fast convergence of higher-order solvers with the low computational cost of backpropagation for network training. Instead of using the adjoint method to backpropagate through the physical process, we replace that gradient by the update computed from a higher-order solver which can encode the local nonlinear behavior of the physics. These physics updates are then passed on to a traditional neural network optimizer which computes the updates to the network weights using backpropagation. Thereby our approach maintains compatibility with acceleration schemes [18, 33] and stabilization techniques [32, 31, 7] developed for training deep learning models. The replacement of the physics gradient yields a simple mathematical formulation that lends itself to straightforward integration into existing machine learning frameworks.

In addition to a theoretical discussion, we perform an extensive empirical evaluation on a wide variety of inverse problems including the highly challenging Navier-Stokes equations. We find that using higher-order or domain-specific solvers can drastically improve convergence speed and solution quality compared to traditional training without requiring the evaluation of the Hessian w.r.t. the model parameters.

2 Scale-invariance in Optimization

We consider unconstrained inverse problems that involve a differentiable physical process $\mathcal{P} : X \subset \mathbb{R}^{d_x} \to Y \subset \mathbb{R}^{d_y}$ which can be simulated. Here X denotes the physical parameter space and Y the space of possible observed outcomes. Given an observed or desired output $y^* \in Y$, the inverse problem consists of finding optimal parameters

$$x^* = \arg\min_x L(x) \quad \text{with} \quad L(x) = \frac{1}{2}\|\mathcal{P}(x) - y^*\|_2^2. \tag{1}$$

Such problems are classically solved by starting with an initial guess x_0 and iteratively applying updates $x_{k+1} = x_k + \Delta x_k$. Newton's method [5] and many related methods [10, 36, 23, 41, 46, 8, 12, 6] approximate L around x_k as a parabola $\tilde{L}(x) = L(x_k) + \frac{\partial L(x_k)}{\partial x_k}(x - x_k) + H_k(x - x_k)^2$ where H_k denotes the Hessian or an approximation thereof. Inverting $\tilde{L}$ and walking towards its minimum with step size η yields

$$\Delta x_k = -\eta \cdot H_k^{-1} \left(\frac{\partial L(x_k)}{\partial x_k} \right)^T. \tag{2}$$

The inversion of H results in scale-invariant updates, i.e. when rescaling x or any component of x, the optimization will behave the same way, leaving $L(x_k)$ unchanged. An important consequence of scale-invariance is that minima will be approached equally quickly in terms of L no matter how wide or deep they are.

Newton-type methods have one major downside, however. The inversion depends on the Hessian H which is expensive to compute exactly, and hard to approximate in typical machine learning settings where mini-batches are ubiquitous [51] and parameter spaces very high-dimensional [25].

Instead, practically all state-of-the-art deep learning relies on first-order information only. Setting H to the identity in Eq. 2 yields gradient descent updates $\Delta x = -\eta \cdot \left(\frac{\partial L}{\partial x}\right)^T$ which are not scale-invariant. Rescaling x by λ also scales Δx by λ, inducing a factor of λ^2 in the first-order-accurate loss change $L(x) - L(x + \Delta x) = -\eta \cdot (\frac{\partial L}{\partial x})^2 + \mathcal{O}(\Delta x^2)$. Gradient descent prescribes small updates to parameters that require a large change to decrease L and vice-versa, typically resulting in slower convergence than Newton updates [56]. This behavior is the cause of exploding gradients or vanishing gradients in deep neural networks. The step size η alone cannot remedy this behavior whenever $\frac{\partial L}{\partial x}$

varies along x. Figure 1a shows the optimization trajectories for the simple problem $\mathcal{P}(x) = (x_1, x_2^2)$ to illustrate this problem.

When training a neural network, the effect of scaling-variance can be reduced through normalization in intermediate layers [32, 7] and regularization [37, 53] but this level of control is not present in most other optimization tasks, such as inverse problems (Eq. 1). More advanced first-order optimizers try to solve the scaling issue by approximating higher-order information [33, 29, 18, 39, 40, 55, 45, 50], such as Adam where $H \approx \mathrm{diag}\left(|\frac{\partial L}{\partial x}|\right)$, decreasing the loss scaling factor from λ^2 to λ. However, these methods lack the exact higher-order information which limits the accuracy of the resulting update steps when optimizing nonlinear functions.

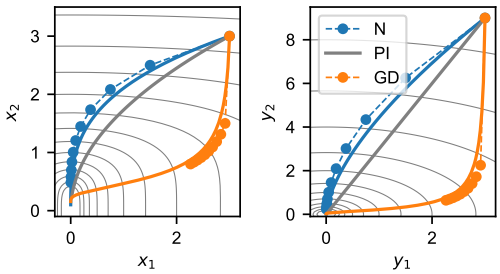

Figure 1: Minimization (Eq. 1) with $y \equiv \mathcal{P}(x) = (x_1, x_2^2)$. L contours in gray. Trajectories of gradient descent (GD), Newton's method (N), and perfect physics inversion (PI) shown as lines (infinitesimal η) and circles (10 iterations with constant η).

3 Scale-invariant Physics and Deep Learning

We are interested in finding solutions to Eq. 1 using a neural network, $x^* = \mathrm{NN}(y^* \,|\, \theta)$, parameterized by θ. Let $\mathcal{Y}^* = \{y_i^* \,|\, i = 1, ..., N\}$ denote a set of N inverse problems, all sharing the same physical process $\mathcal{P}$. Then training the network means finding

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathcal{P}\big(\mathrm{NN}(y_i^* \,|\, \theta)\big) - y_i^*\|_2^2 \,. \tag{3}$$

Assuming a large parameter space θ and the use of mini-batches, higher-order methods are difficult to apply to this problem, as described above. Additionally, the scale-variance issue of gradient descent is especially pronounced in this setting because only the network part of the joint problem $\mathrm{NN} \circ \mathcal{P}$ can be properly normalized while the physical process $\mathcal{P}$ is fixed. Therefore, the traditional approach of computing the gradient $\frac{\partial L}{\partial \theta}$ using the adjoint method (backpropagation) can lead to undesired behavior.

Consider the problem $\mathcal{P}(x) = e^x$ with observed data $y^* \in (0, 1]$ (Appendix C.2). Due to the exponential form of $\mathcal{P}$, the curvature around the solutions $x^* = \log(y^*)$ strongly depends on y^*. This causes first-order network optimizers such as SGD or Adam to fail in approximating the correct solutions for small y^* because their gradients are overshadowed by larger y^* (see Fig. 2). Scaling the gradients to unit length in x drastically improves the prediction accuracy, which hints at a possible solution: If we could employ a scale-invariant physics solver, we would be able to optimize all examples, independent of the curvature around their respective minima.

3.1 Derivation

If $\mathcal{P}$ has a unique inverse and is sufficiently well-behaved, we can split the joint optimization problem (Eq. 3) into two stages. First, solve all inverse problems individually, constructing a set of unique solutions $\mathcal{X}^{\mathrm{sv}} = \{x_i^{\mathrm{sv}} = \mathcal{P}^{-1}(y_i^*)\}$ where $\mathcal{P}^{-1}$ denotes the inverse problem solver. Second, use $\mathcal{X}^{\mathrm{sv}}$ as labels for supervised training

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathrm{NN}(y_i^* \,|\, \theta) - x_i^{\mathrm{sv}}\|_2^2. \tag{4}$$

This enables scale-invariant physics (SIP) inversion while a fast first-order method can be used to train the network which can be constructed to be normalized using state-of-the-art procedures [32, 7, 53].

Unfortunately, this two-stage approach is not applicable in multimodal settings, where x^{sv} depends on the initial guess x_0 used in the first stage. This causes the network to interpolate between possible

solutions, leading to subpar convergence and generalization performance. To avoid these problems, we alter the training procedure from Eq. 4 in two ways. First, we reintroduce $\mathcal{P}^{-1}$ into the training loop, yielding

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathrm{NN}(y_i^* \,|\, \theta) - \mathcal{P}^{-1}(y_i^*)\|_2^2 \,. \tag{5}$$

Next, we condition $\mathcal{P}^{-1}$ on the neural network prediction by using it as an initial guess, $\mathcal{P}^{-1}(y^*) \rightarrow \mathcal{P}^{-1}(y^* \,|\, \mathrm{NN}(y^* \,|\, \theta))$. This makes training in multimodal settings possible because the embedded solver $\mathcal{P}^{-1}$ searches for minima close to the prediction of the NN. Therefore θ can exit the basin of attraction of other minima and does not need to interpolate between possible solutions. Also, since all inverse problems from $\mathcal{Y}^*$ are optimized jointly, this reduces the likelihood of any individual solution getting stuck in a local minimum, as discussed earlier.

The obvious downside to this approach is that $\mathcal{P}^{-1}$ must be run for each training step. When $\mathcal{P}^{-1}$ is an iterative solver, we can write it as $P^{-1}(y^* \,|\, \mathrm{NN}(y^* \,|\, \theta)) = \mathrm{NN}(y^* \,|\, \theta) + \Delta x_0 + ... + \Delta x_n$. We denote the first update $\Delta x_0 \equiv U(y^* \,|\, \theta)$.

Instead of computing all updates Δx, we approximate P^{-1} with its first update U. Inserting this into Eq. 5 with $(\circ) \equiv (y_i^* \,|\, \theta)$ yields

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathrm{NN}(y_i^* \,|\, \theta) - (\mathrm{NN}(\circ) + U(\circ))\,\|_2^2 . \tag{6}$$

This can be further simplified to $\sum_{i=1}^{N} \frac{1}{2} \|U(y_i^* \,|\, \theta)\|_2^2$ but this form is hard to optimize as is requires back-propagation through the solver U. In addition, its minimization is not sufficient, because all fixed points of U, such as maxima or saddle points of L, can act as attractors.

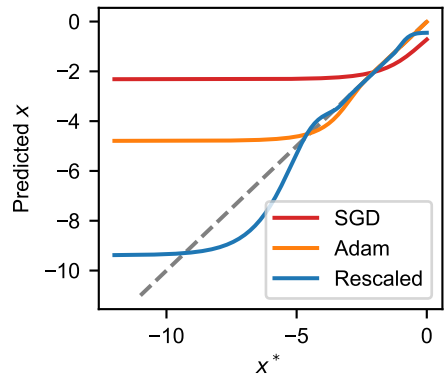

Figure 2: Networks trained according to Eq. 3 with $\mathcal{P}(x) = e^x$. Stochastic gradient descent (SGD) and Adam fail to approximate solutions for small values due to scale variance. Normalizing the gradient in x space (Rescaled) improves solution accuracy by decreasing scale variance.

Instead, we assume $\frac{\partial \mathcal{P}^{-1}}{\partial y} = 0$, which removes the $(\circ)$ dependencies in Eq. 6 and leaves only the dependency of the first term $\mathrm{NN}(y_i^* \,|\, \theta)$. This avoids both of the problems above. It also allows us to factor the optimization into a network and a physics graph (see Fig. 3), so that all necessary derivative computations only require data from one of them.

3.2 Update Rule

The factorization described above results in the following update rule for training with SIP updates, shown in Fig. 3.

1. Pass the network prediction x_0 to the physics graph and compute $\Delta x_0 \equiv U(y^* \,|\, x_0)$ for all examples in the mini-batch.
2. Send $\tilde{x} \equiv x_0 + \Delta x_0$ back and compute the network loss $\tilde{L} = \frac{1}{2}||x_0 - \tilde{x}||_2^2$.
3. Update θ using any neural network optimizer, such as SGD or Adam, treating $\tilde{x}$ as a constant.

To see what updates $\Delta\theta$ result from this algorithm, we compute the gradient w.r.t. θ,

$$\frac{\partial \tilde{L}}{\partial \theta} = \sum_{i=1}^{N} U(y^* \,|\, x_0) \cdot \frac{\partial \mathrm{NN}}{\partial \theta} \tag{7}$$

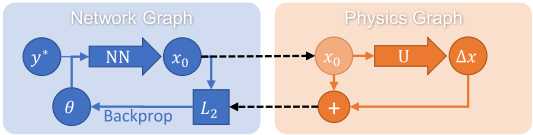

Figure 3: Neural network (NN) training procedure with embedded physics solver (Opt).

Comparing this to the gradient resulting from optimizing the joint problem $\mathrm{NN} \circ \mathcal{P}$ with a single optimizer (Eq. 3),

$$\frac{\partial L}{\partial \theta} = \Delta y_i \cdot \frac{\partial \mathcal{P}}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta} \tag{8}$$

where $\Delta y = \mathcal{P}\big(\mathrm{NN}(y_i^* \,|\, \theta)\big) - y_i^*$, we see that U takes the place of $\Delta y_i \cdot \frac{\partial \mathcal{P}}{\partial x}$, the adjoint vector that would otherwise be computed by backpropagation. Unlike the adjoint vector, however, $\tilde{x}$ encodes an actual physical configuration. Since $\tilde{x}$ can stem from a higher-order solver, the resulting updates $\Delta\theta$ can also encode non-linear information about the physics without computing the Hessian w.r.t. θ.

3.3 Convergence

It is a priori not clear whether SIP training will converge, given that $\frac{\partial \mathcal{P}^{-1}}{\partial y}$ is not computed. We start by proving convergence for two special choices of the inverse problem solver $\mathcal{P}^{-1}$ before considering the general case. We assume that NN is expressive enough to fit our problem and that it is able to converge to every point x for all examples using gradient descent:

$$\exists \eta > 0 \,:\, \forall i \,\forall x \,\forall \epsilon > 0 \,\exists n \in \mathbb{N} : ||\mathrm{NN}_{\theta_n} - x||_2 \leq \epsilon \tag{9}$$

where θ_n is the sequence of gradient descent steps

$$\theta_{n+1} = \theta_n - \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta} \right)^T (\mathrm{NN}_{\theta_n} - x) \tag{10}$$

with $\eta > 0$. Large enough networks fulfill this property under certain assumptions [17] and the universal approximation theorem guarantees that such a configuration exists [13].

In the first special case, $U(x) \equiv U(y^* \,|\, x)$ points directly towards a solution x^*. This models the case of a known ground truth solution in a unimodal setting. For brevity, we will drop the example indices and the dependencies on y^*.

Theorem 1. *If $\forall x \,\exists \lambda \in (0, 1] \,:\, U(x) = \lambda(x^* - x)$, then the gradient descent sequence NN_{θ_n} with $\theta_{n+1} = \theta_n + \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta}\right)^T U(x)$ converges to x^*.*

Proof. Rewriting $U(x) = -\frac{\partial}{\partial x}\left(\frac{\lambda}{2}||x - x^*||_2^2\right)$ yields the update $\theta_{n+1} - \theta_n = -\eta \left(\frac{\partial \hat{L}}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta}\right)^T$ where $\hat{L} = \frac{\lambda}{2}||x - x^*||_2^2$ which describes gradient descent towards x^* with the gradients scaled by λ. Since $\lambda \in (0, 1]$, the convergence proof of gradient descent applies. □

The second special case has $U(x)$ pointing in the direction of steepest gradient descent in x space.

Theorem 2. *If $\forall x \,\exists \lambda \in (0, 1] \,:\, U(x) = -\lambda \left(\frac{\partial L}{\partial x}\right)^T$, then the gradient descent sequence NN_{θ_n} with $\theta_{n+1} = \theta_n + \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta}\right)^T U(x)$ converges to minima of L.*

Proof. This is equivalent to gradient descent in $L(\theta) \equiv (\mathrm{NN} \circ L)(\theta)$. Rewriting the update yields $\theta_{n+1} - \theta_n = -\eta\lambda \left(\frac{\partial L}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta}\right)^T$ which is the gradient descent update scaled by λ. □

Next, we consider the general case of an arbitrary $\mathcal{P}^{-1}$ and U. We require that U decreases L by a minimum relative amount specified by τ,

$$\exists \tau > 0 \,:\, \forall x \,:\, L(x) - L(x + U(x)) \geq \tau \left(L(x) - L(x^*)\right). \tag{11}$$

To guarantee convergence to a region, we also require

$$\exists K > 0 \,:\, \forall x \,:\, ||U(x)|| \leq K(L(x) - L(x^*)). \tag{12}$$

Theorem 3. *There exists an update strategy $\theta_{n+1} = S(\theta_n)$ based on a single evaluation of U for which $L(\mathrm{NN}_{\theta_n}(y))$ converges to a minimum x^* or minimum region of L for all examples.*

Proof. We denote $\tilde{x}_n \equiv x_n + U(x_n)$ and $\Delta L^* = L(x_n) - L(x^*)$. Let I_2 denote the open set of all x for which $L(x) - L(x_n) > \frac{\tau}{2}\Delta L^*$. Eq. 11 provides that $\tilde{x}_n \in I_2$. Since I_2 is open, $\exists \epsilon > 0 : \forall x \in B_\epsilon(\tilde{x}_n) : L(x_n) - L(x) > \frac{\tau}{2}\Delta L^*$ where $B_\epsilon(x)$ denotes the open set containing all $x' \in \mathbb{R}^d$ for which $||x' - x||_2 < \epsilon$, i.e. there exists a small ball around $\tilde{x}_n$ which is fully contained in I_2 (see sketch in Fig. 4).

Using the Eq. 9, we can find a finite $n \in \mathbb{N}$ for which $\mathrm{NN}_{\theta_n} \in B_\epsilon(\tilde{x}_n)$ and therefore $L(x_n) - L(\mathrm{NN}_{\theta_n}) > \frac{\tau}{2}\Delta L^*$. We can thus use the following strategy S for minimizing L: First, compute

$\tilde{x}_n = x_n + U(x_n)$. Then perform gradient descent steps in θ with the effective objective function $\frac{1}{2}||\mathrm{NN}_\theta - \tilde{x}_n||_2^2$ until $L(x_n) - L(\mathrm{NN}_\theta) \geq \frac{\tau}{2}\Delta L^*$. Each application of S reduces the loss to $\Delta L^*_{n+1} \leq (1 - \frac{\tau}{2})\Delta L^*_n$ so any arbitrary value of $L > L(x^*)$ can be reached within a finite number of steps. Eq. 12 ensures that $||U(x)|| \to 0$ as the optimization progresses which guarantees that the optimization converges to a minimum region. □

While this theorem guarantees convergence, it requires potentially many gradient descent steps in θ for each physics update U. This can be advantageous in special circumstances, e.g. when U is more expensive to compute than an update in θ, or when θ is far away from a solution. However, in many cases, we want to re-evaluate U after each update to θ. Without additional assumptions about U and NN_θ, there is no guarantee that L will decrease every iteration, even for infinitesimally small step sizes. Despite this, there is good reason to assume that the optimization will decrease L over time. This can be seen visually in Fig. 4 where the next prediction x_{n+1} is guaranteed to lie within the blue circle. The region of increasing loss is shaded orange and always fills less than half of the volume of the circle, assuming we choose a sufficiently small step size. We formalize this argument in appendix A.2.

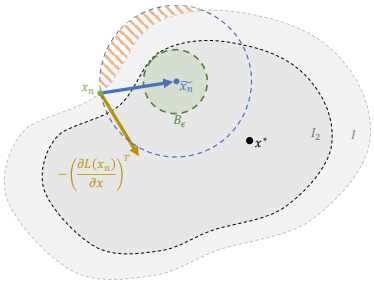

Figure 4: Convergence visualization for the proof of theorem 3. All shown objects are visualized in x space for one example i.

Remarks and lemmas We considered the case that $U \equiv \Delta x_0$. This can be trivially extended to the case that $U \equiv \Delta x_0 + ... + \Delta x_m$ for any $m \in \mathbb{N}$. When we let the solver run to convergence, i.e. m large enough, theorem 1 guarantees convergence if $\mathcal{P}^{-1}$ consistently converges to the same x_i^*. Also note that any minimum θ^* found with SIP training fulfills $U = 0$ for all examples.

3.4 Experimental Characterization

We first investigate the convergence behavior of SIP training depending on characteristics of $\mathcal{P}$. We construct the synthetic two-dimensional inverse process

$$\mathcal{P}(x) = (\sin(\hat{x}_1)/\xi,\ \hat{x}_2 \cdot \xi) \quad \text{with} \quad \hat{x} = \gamma \cdot R_\phi \cdot x,$$

where $R_\phi \in \mathrm{SO}(2)$ denotes a rotation matrix and $\gamma > 0$. The parameters ξ and ϕ allow us to continuously change the characteristics of the system. The value of ξ determines the conditioning of $\mathcal{P}$ with large ξ representing ill-conditioned problems while ϕ describes the coupling of x_1 and x_2. When $\phi = 0$, the off-diagonal elements of the Hessian vanish and the problem factors into two independent problems. Fig. 5a shows one example loss landscape.

We train a fully-connected neural network to invert this problem (Eq. 3), comparing SIP training using a saddle-free Newton solver [14] to various state-of-the-art network optimizers. We select the best learning rate for each optimizer independently. For $\xi = 0$, when the problem is perfectly conditioned, all network optimizers converge, with Adam converging the quickest (Fig. 5b). Note that the relatively slow convergence of SIP mostly stems from it taking significantly more time per iteration than the other methods, on average 3 times as long as Adam. While the evaluation of the Hessian inherently requires more computations, the per-iteration time of SIP could likely be significantly reduced by optimizing the computations.

When increasing ξ with $\phi = 0$ fixed (Fig. 5d), the accuracy of all traditional network optimizers decreases because the gradients scale with $(1/\xi, \xi)$ in x, becoming longer in x_2, the direction that requires more precise values. SIP training avoids this using the Hessian, inverting the scaling behavior and producing updates that align with the flat direction in x. This allows SIP training to retain its relative accuracy over a wide range of ξ. At $\xi = 32$, only SIP and Adam succeed in optimizing the network to a significant degree (Fig. 5c).

Varying ϕ with $\xi = 32$ fixed (Fig. 5e), sheds light on how Adam manages to learn in ill-conditioned settings. Its diagonal approximation of the Hessian reduces the scaling effect when x_1 and x_2 lie on different scales, but when the parameters are coupled, the lack of off-diagonal terms prevents this. SIP training has no problem with coupled parameters since its updates are based on the full-rank Hessian $\frac{\partial^2 L}{\partial x^2}$.

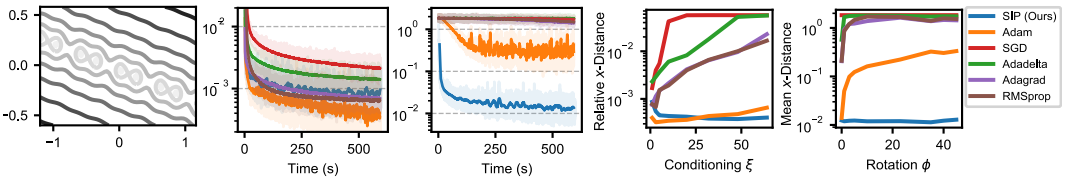

Figure 5: (a) Example loss landscape with $y^* = (0.3, -0.5)$, $\xi = 1$, $\phi = 15°$. (b,c) Learning curves with $\phi = \frac{\pi}{4}$, averaged over 256 min-batches. For (b) $\xi = 1$, and (c) $\xi = 32$. (d) Dependence on problem conditioning ξ (with $\phi = 0$). (e) Dependence on parameter coupling ϕ (with $\xi = 32$).

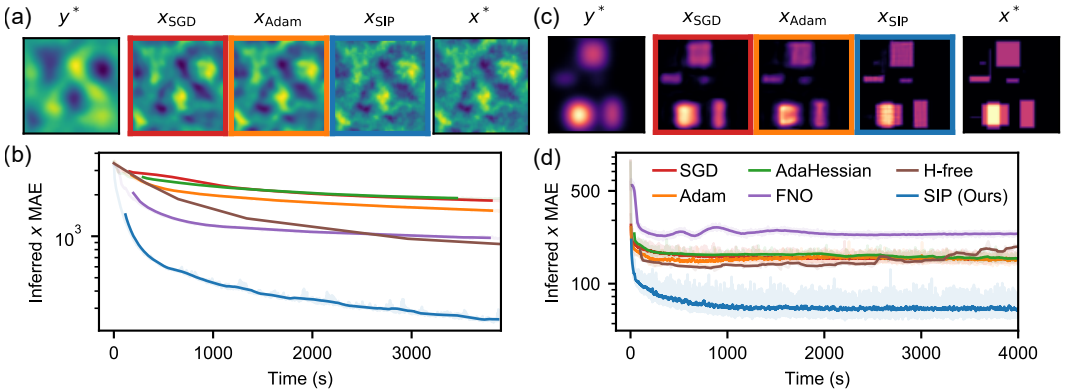

Figure 6: Poisson's equation (left) and the heat equation (right). (a,c) Example from the data set: observed distribution (y^*), inferred solutions, and ground truth solution (x^*). (b,d) Learning curves, running average over 64 mini-batches (except for H-free).

3.5 Application to high-dimensional Problems

Explicitly evaluating the Hessian is not feasible for high-dimensional problems. However, scale-invariant updates can still be computed, e.g. by inverting the gradient or via domain knowledge. We test SIP training on three high-dimensional physical systems described by partial differential equations: Poisson's equation, the heat equation, and the Navier-Stokes equations. This selection covers ubiquitous physical processes with diffusion, transport, and strongly non-local effects, featuring both explicit and implicit solvers. All code required to reproduce our results is provided in the supplemental material and will be made public. A detailed description of all experiments along with additional visualizations and performance measurements can be found in appendix B.

Poisson's equation Poisson's equation, $\mathcal{P}(x) = \nabla^{-2}x$, plays an important role in electrostatics, Newtonian gravity, and fluid dynamics [4]. It has the property that local changes in x can affect $\mathcal{P}(x)$ globally. Here we consider a two-dimensional system and train a U-net [48] to solve inverse problems (Eq. 1) on pseudo-randomly generated y^*. We compare SIP training to SGD with momentum, Adam, AdaHessian [55], Fourier neural operators (FNO) [34] and Hessian-free optimization (H-free) [39]. Fig. 6b shows the learning curves. The training with SGD, Adam and AdaHessian drastically slows after around 200 and 300 iterations, respectively. FNO and H-free both improve upon this behavior, reaching twice the accuracy before slowing. For SIP, we construct scale-invariant Δx based on the analytic inverse of Poisson's equation and use Adam to compute $\Delta\theta$. The curve closely resembles an exponential curve, which indicates linear convergence, the ideal case for first-order methods optimizing an L_2 objective. During all of training, the SIP variant converges exponentially faster than the traditional optimizers, its relative performance difference compared to Adam continually increasing from a factor of 3 at iteration 60 to a full order of magnitude after 5k iterations. This difference can be seen in the inferred solutions (Fig. 6a) which are noticeably more detailed.

Heat equation Next, we consider a system with fundamentally non-invertible dynamics. The heat equation, $\frac{\partial u}{\partial t} = \nu \cdot \nabla^2 u$, models heat flow in solids but also plays a part in many diffusive systems [16]. It gradually destroys information as the temperature equilibrium is approached [26],

causing $\nabla\mathcal{P}$ to become near-singular. Inspired by heat conduction in microprocessors, we generate examples x_{GT} by randomly scattering four to ten heat generating rectangular regions on a plane and simulating the heat profile $y^* = \mathcal{P}(x_{\mathrm{GT}})$ as observed from outside a heat-conducting casing. The learning curves for the corresponding inverse problem are shown in Fig. 6d.

When training with SGD, Adam or AdaHessian, we observe that the distance to the solution starts rapidly decreasing before decelerating between iterations 30 and 40 to a slow but mostly stable convergence. The sudden deceleration is rooted in the adjoint problem, which is also a diffusion problem. Backpropagation through $\mathcal{P}$ removes detail from the gradients, which makes it hard for first-order methods to recover the solution. H-free initially finds better solutions but then stagnates with the solution quality slowly deteriorating. FNO performs poorly on this task, likely due to the sharp edges in x^*.

The information loss in $\mathcal{P}$ prevents direct numerical inversion of the gradient or Hessian. Instead, we add a dampening term to derive a numerically stable scale-invariant solver which we use for SIP training. Unlike SGD and Adam, the convergence of SIP training does not decelerate as early as the other methods, resulting in an exponentially faster convergence. At iteration 100, the predictions are around 34% more accurate compared to Adam, and the difference increases to 130% after 10k iterations, making the reconstructions noticeably sharper than with traditional training methods (Fig. 6c). To test the dependence of SIP on hyperparameters like batch size or learning rate, we perform this experiment with a range of values (see appendix B.3). Our results indicate that SIP training and Adam are impacted the same way by non-optimal hyperparameter configurations.

Navier-Stokes equations Fluids and turbulence are among the most challenging and least understood areas of physics due to their highly nonlinear behavior and chaotic nature [21]. We consider a two-dimensional system governed by the incompressible Navier-Stokes equations: $\frac{\partial v}{\partial t} = \nu\nabla^2 v - v \cdot \nabla v - \nabla p$, $\nabla \cdot v = 0$, where p denotes pressure and ν the viscosity. At $t = 0$, a region of the fluid is randomly marked with a massless colorant m_0 that passively moves with the fluid, $\frac{\partial m}{\partial t} = -v \cdot \nabla m$. After time t, the marker is observed again to obtain m_t. An example observation pair $y^* = \{m_0, m_t\}$ is shown in Fig. 7a. The task is to find an initial fluid velocity $x \equiv v_0$ such that the fluid simulation $\mathcal{P}$ matches m_t at time t. Since $\mathcal{P}$ is deterministic, x encodes the complete fluid flow from 0 to t. We define the objective in frequency space with lower frequencies being weighted more strongly. This definition considers the match of the marker distribution on all scales, from the coarse global match to fine details, and is compatible with the definition in Eq. 1. We train a U-net [48] to solve these inverse problems; the learning curves are shown in Fig. 7b.

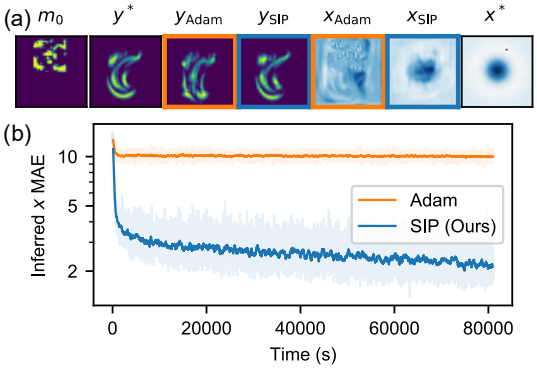

Figure 7: Incompressible fluid flow reconstruction. (a) Example from the data set: initial marker distribution (m_0); simulated marker distribution after time t using ground-truth velocity (y^*) and network predictions ($y_\circ$); predicted initial velocities ($x_\circ$); ground truth velocity (x^*). (b) Learning curves, running average over 64 mini-batches.

When training with Adam, the error decreases for the first 100 iterations while the network learns to infer velocities that lead to an approximate match. The error then proceeds to decline at a much lower rate, nearly coming to a standstill. This is caused by an overshoot in terms of vorticity, as visible in Fig. 7a right. While the resulting dynamics can roughly approximate the shape of the observed m_t, they fail to match its detailed structure. Moving from this local optimum to the global optimum is very hard for the network as the distance in x space is large and the gradients become very noisy due to the highly non-linear physics. A similar behavior can also be seen when optimizing single inverse problems with gradient descent where it takes more than 20k iterations for GD to converge on single problems.

For SIP training, we run the simulation in reverse, starting with m_t, to estimate x in a scale-invariant manner. When used for network training, we observe vastly improved convergence behavior compared to first-order training. The error rapidly decreases during the first 200 iterations, at which point the

inferred solutions are more accurate than pure Adam training by a factor of 2.3. The error then continues to improve at a slower but still exponentially faster rate, reaching a relative accuracy advantage of 5x after 20k iterations. To match the network trained with pure Adam for 20K iterations, the SIP variant only requires 55 iterations. This improvement is possible because the inverse-physics solver and associated SIP updates do not suffer from the strongly-varying gradient magnitudes and directions, which drown the first-order signal in noise. Instead, the SIP updates behave much more smoothly, both in magnitude and direction.

Comparing the inferred solutions from the network to an iterative approach shows a large difference in inference time (table 1). To reach the same solution quality as the neural network prediction, $\mathcal{P}_{\text{NS}}^{-1}$ needs 7 iterations on average, which takes more than 10,000 times as long, and gradient descent (GD) does not reach the same quality even after 20k iterations. This difference is caused by P_{NS}^{-1} having to run the full forward and backward simulation for each iteration. This cost is also required for each training iteration of the network but once converged, its inference is extremely fast, solving around 9000 problems per second in batch mode. For both iterative solver and network, we used a batch size of 64 and divide the total time by the batch size.

Table 1: Time to reach equal solution quality in the fluid experiment, measured as MAE in x space. The inference time is given per example in batch mode, followed by the number of iterations in parentheses.

Method	Training time	Inference time
NN	17.6 h (15.6 k)	0.11 ms
$\mathcal{P}_{\text{NS}}^{-1}$	n/a	2.2 s (7)
GD	n/a	$>$ 4h (20k)

3.6 Limitations and Discussion

While SIP training manages to find vastly more accurate solutions for the examples above, there are some caveats to consider. First, an approximately scale-invariant physics solver is required. While in low-dimensional x spaces Newton's method is a good candidate, high-dimensional spaces require another form of inversion. Some equations can locally be inverted analytically but for complex problems, domain-specific knowledge may be required. However, this is a widely studied field and many specialized solvers have been developed [9].

Second, SIP uses traditional first-order optimizers to determine $\Delta\theta$. As discussed, these solvers behave poorly in ill-conditioned settings which can also affect SIP performance when the network outputs lie on different scales. Some recent works address this issue and have proposed network optimization based on inversion [40, 20].

Third, while SIP training generally leads to more accurate solutions, measured in x space, the same is not always true for the loss $L = \sum_i L_i$. SIP training weighs all examples equally, independent of the curvature $|\frac{\partial^2 L}{\partial x^2}|$ near a chosen solution. This can cause small errors in examples with large curvatures to dominate L. In these cases, or when the accuracy in x space is not important, like in some control tasks, traditional training methods may perform better than SIP training.

4 Conclusions

We have introduced SIP training, a novel neural network training scheme for learning solutions to nonlinear inverse problems. SIP training leverages physics inversion to compute scale-invariant updates in the solution space. It provably converges when enough network updates $\Delta\theta$ are performed per solver evaluation and we have shown that it converges with a single $\Delta\theta$ update for a wide range of physics experiments. The scale-invariance allows it to find solutions exponentially faster than traditional learning methods for many physics problems while keeping the computational cost relatively low. While this work targets physical processes, SIP training could also be applied to other coupled nonlinear optimization problems, such as differentiable rendering or training invertible neural networks.

Scale-invariant optimizers, such as Newton's method, avoid many of the problems that plague deep learning at the moment. While their application to high-dimensional parameter spaces is currently limited, we hope that our method will help establish them as commonplace tools for training neural networks in the future.

References

[1] Craig E Aalseth, N Abgrall, Estanislao Aguayo, SI Alvis, M Amman, Isaac J Arnquist, FT Avignone III, Henning O Back, Alexander S Barabash, PS Barbeau, et al. Search for neutrinoless double-β decay in ge 76 with the majorana demonstrator. *Physical review letters*, 120(13):132502, 2018.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] M Agostini, M Allardt, E Andreotti, AM Bakalyarov, M Balata, I Barabanov, M Barnabé Heider, N Barros, L Baudis, C Bauer, et al. Pulse shape discrimination for gerda phase i data. *The European Physical Journal C*, 73(10):2583, 2013.

[4] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

[5] Kendall E Atkinson. *An introduction to numerical analysis*. John wiley & sons, 2008.

[6] Mordecai Avriel. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. In *International Conference on Learning Representations (ICLR)*, 2016.

[8] Ernst R Berndt, Bronwyn H Hall, Robert E Hall, and Jerry A Hausman. Estimation and inference in nonlinear structural models. In *Annals of Economic and Social Measurement, Volume 3, number 4*, pages 653–665. NBER, 1974.

[9] A. Borzì, G. Ciaramella, and M. Sprengel. *Formulation and Numerical Solution of Quantum Control Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2017.

[10] Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[11] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4):045002, 2019.

[12] Andrew R Conn, Nicholas IM Gould, and Ph L Toint. Convergence of quasi-newton matrices generated by the symmetric rank one update. *Mathematical programming*, 50(1-3):177–195, 1991.

[13] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[14] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.

[15] S Delaquis, MJ Jewell, I Ostrovskiy, M Weber, T Ziegler, J Dalmasson, LJ Kaufman, T Richards, JB Albert, G Anton, et al. Deep neural networks for energy and position reconstruction in exo-200. *Journal of Instrumentation*, 13(08):P08023, 2018.

[16] Jerome Droniou. Finite volume schemes for diffusion equations: introduction to and review of modern methods. *Mathematical Models and Methods in Applied Sciences*, 24(08):1575–1619, 2014.

[17] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[18] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[19] F. W. Dyson, A. S. Eddington, and C. Davidson. A Determination of the Deflection of Light by the Sun's Gravitational Field, from Observations Made at the Total Eclipse of May 29, 1919, January 1920.

[20] Israel Elias, José de Jesús Rubio, David Ricardo Cruz, Genaro Ochoa, Juan Francisco Novoa, Dany Ivan Martinez, Samantha Muñiz, Ricardo Balcazar, Enrique Garcia, and Cesar Felipe Juarez. Hessian with mini-batches for electrical demand prediction. *Applied Sciences*, 10(6):2036, 2020.

[21] Giovanni Galdi. *An introduction to the mathematical theory of the Navier-Stokes equations: Steady-state problems*. Springer Science & Business Media, 2011.

[22] Daniel George and EA Huerta. Deep learning for real-time gravitational wave detection and parameter estimation: Results with advanced ligo data. *Physics Letters B*, 778:64–70, 2018.

[23] Philip E Gill and Walter Murray. Algorithms for the solution of the nonlinear least-squares problem. *SIAM Journal on Numerical Analysis*, 15(5):977–992, 1978.

[24] Alexander Gluhovsky and Christopher Tong. The structure of energy conserving low-order models. *Physics of Fluids*, 11(2):334–343, 1999.

[25] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

[26] Matthew A Grayson. The heat equation shrinks embedded plane curves to round points. *Journal of Differential geometry*, 26(2):285–314, 1987.

[27] FH Harlow. The marker-and-cell method. *Fluid Dyn. Numerical Methods*, 38, 1972.

[28] Francis H Harlow and J Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The physics of fluids*, 8(12):2182–2189, 1965.

[29] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[30] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. In *International Conference on Learning Representations (ICLR)*, 2020.

[31] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv:1502.03167*, February 2015.

[33] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[34] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

[35] GV Kraniotis and SB Whitehouse. Compact calculation of the perihelion precession of mercury in general relativity, the cosmological constant and jacobi's inversion problem. *Classical and Quantum Gravity*, 20(22):4817, 2003.

[36] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[37] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[38] Rajesh Maingi, Arnold Lumsdaine, Jean Paul Allain, Luis Chacon, SA Gourlay, CM Greenfield, JW Hughes, D Humphreys, V Izzo, H McLean, et al. Summary of the fesac transformative enabling capabilities panel report. *Fusion Science and Technology*, 75(3):167–177, 2019.

[39] James Martens et al. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[40] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[41] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[42] Patrick Mullen, Keenan Crane, Dmitry Pavlov, Yiying Tong, and Mathieu Desbrun. Energy-preserving integrators for fluid animation. *ACM Transactions on Graphics (TOG)*, 28(3):1–8, 2009.

[43] Mykola Novik. torch-optimizer – collection of optimization algorithms for PyTorch., 1 2020.

[44] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Advances in Neural Information Processing Systems*, pages 1–8, 2017.

[45] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[46] Michael Powell. A hybrid method for nonlinear equations. In *Numerical methods for nonlinear algebraic equations*. Gordon and Breach, 1970.

[47] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3 edition, 2007.

[48] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[49] Sirpa Saarinen, Randall Bramley, and George Cybenko. Ill-conditioning in neural network training problems. *SIAM Journal on Scientific Computing*, 14(3):693–714, 1993.

[50] Patrick Schnell, Philipp Holl, and Nils Thuerey. Half-inverse gradients for physical deep learning. *arXiv preprint arXiv:2203.10131*, 2022.

[51] Nicol N Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-newton method for online convex optimization. In *Artificial intelligence and statistics*, pages 436–443, 2007.

[52] Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3):350–371, June 2008.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[54] Albert Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, 2005.

[55] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W Mahoney. Adahessian: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719*, 2020.

[56] Nan Ye, Farbod Roosta-Khorasani, and Tiangang Cui. Optimization methods for inverse problems. In *2017 MATRIX Annals*, pages 121–140. Springer, 2019.

[57] Kemin Zhou, John Comstock Doyle, Keith Glover, et al. *Robust and optimal control*, volume 40. Prentice hall New Jersey, 1996.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] We do not expect any negative impacts.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] We declare all assumptions the main text and provide an additional compact list in the appendix.
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We provide the full source code in the supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] We list learning rates and our training procedure in the appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We show learning curves with different seeds in the appendix.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Our hardware is listed in the appendix.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] All used software libraries are given in the appendix.
   (b) Did you mention the license of the assets? [N/A] No proprietary licenses are involved.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We include our source code.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Convergence

Here we give a more detailed explanation of some of the convergence arguments made in the paper.

A.1 Assumptions

The following is a sufficient, self-contained list of all assumptions required for the theorems in the main text.

1. Let $L = \{L_i : \mathbb{R}^d \to \mathbb{R} \,|\, i = 1, ..., N\}$ be a finite set of functions and let $x_i^* \in \mathbb{R}^d$ be any global minima of L_i.
2. Let $U = \{U_i : \mathbb{R}^d \to \mathbb{R}^d \,|\, i = 1, ..., N\}$ be a set of update functions for L_i such that $\exists \tau > 0 \,:\, \forall x \in \mathbb{R}^d \,:\, L_i(x) - L_i(x + U_i(x)) \geq \tau\,(L(x) - L(x^*))$ and $\exists K > 0 \,:\, \forall x \in \mathbb{R}^d \,:\, ||U_i(x)|| \leq K(L(x) - L(x^*))$.
3. Let $\mathrm{NN}_\theta : \mathbb{R}^m \to \mathbb{R}^d$ be differentiable w.r.t. θ with the property that $\exists \eta > 0 \,:\, \forall i \in 1, ..., N\ \forall x_i \in \mathbb{R}^d\ \forall \epsilon > 0\ \exists n \in \mathbb{N} : ||\mathrm{NN}_{\theta_n^{x_i}}(y_i) - x_i||_2 \leq \epsilon$ where θ_n^x is the sequence of gradient descent steps with $\theta_{n+1}^x = \theta_n^x - \eta \left(\frac{\partial \mathrm{NN}_\theta}{\partial \theta}\right)^T (\mathrm{NN}_{\theta_n} - x)$, $\eta > 0$.

Given these assumptions, SIP training with sufficiently many network optimization steps n is guaranteed to converge to an optimum point or region. In the two special cases mentioned in the main text, it is guaranteed to converge even for $n = 1$.

If we are interested in convergence to a global optimum, we may additionally require that L is convex.

A.2 Probable Loss Decrease for the Case $n = 1$

To show that the loss decreases on average, we require the the assumption that an update of the neural network weights θ does not systematically distort the update direction in x. More formally, we update θ to minimize

$$\tilde{L} = \frac{1}{2}||\mathrm{NN}_{\theta_n^{x_i}}(y) - \tilde{x}||_2^2,$$

where $\tilde{x}$ denotes the target in x space which is computed using U as defined above. This can be done with gradient descent,

$$\Delta\theta = -\eta \left(\frac{\partial \mathrm{NN}_\theta}{\partial \theta}\right)^T (\mathrm{NN}_\theta - \tilde{x}),$$

or any other method. In fact, recent works have shown that the Jacobian $\frac{\partial \mathrm{NN}_\theta}{\partial \theta}$ can be inverted numerically to compute more precise θ updates [50]. The updated network will predict a new vector $\mathrm{NN}_{\theta+\Delta\theta}(y)$ and we will denote the shift in x space resulting from the update $\Delta\theta$ as $\Delta\mathrm{NN} \equiv \mathrm{NN}_{\theta+\Delta\theta}(y) - \mathrm{NN}_\theta(y)$. This shift is guaranteed to lie closer to $\tilde{x}$ (within the blue circle in Fig. 8) as long as we choose η appropriately. We also know that at $\tilde{x}$ itself, as well as in a small region around $\tilde{x}$, the loss L is smaller than $L(x)$. We denote the region of decreased loss $I = \{x \in \mathbb{R}^d \,:\, L(x) < L(x_n)\}$.

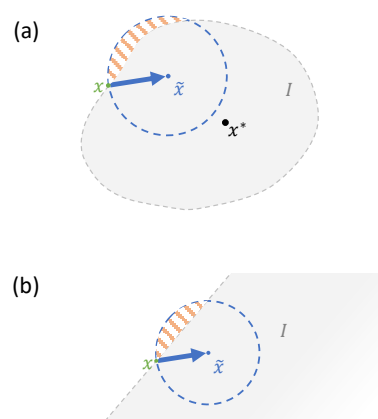

Figure 8: (a) Convergence visualization for the probable loss decrease in the case $n = 1$. (b) Zoomed view for small η. All shown objects are visualized in x space for one example i.

Note that the circle radius $||\tilde{x} - x||_2$ scales with the step size of the higher-order optimizer since $\tilde{x} = x + U(x)$. Factoring out this step size and integrating it into η allows us to arbitrarily scale our problem in x space. In particular, when choosing η to be small, we can linearly approximate the loss landscape (Fig. 8b). Its boundary becomes a straight line separating increased and decreased regions of the loss, and must cut the circle so that more than half of it lies within I. More importantly, the expectation of L integrated over the circle is smaller than $L(x)$.

Therefore, if $\Delta\mathrm{NN}$ points towards $\tilde{x}$ on average, i.e. $\mathbb{E}\left[\Delta\mathrm{NN}\right] = \lambda(\tilde{x} - x)$ with $\lambda \in (0, 1]$, and we choose η small enough, the loss must decrease on average. While the assumption that updates of θ do not systematically distort the direction in x may not hold for all problems or network architectures, we have empirically observed it to be true over a wide range of tests.

B Detailed Description of the Experiments

Here, we give a more detailed description of our experiments including setup and analysis. The implementation of our experiments is based on the Φ_{Flow} (PhiFlow) framework [30] and uses TensorFlow [2] and PyTorch [44] for automatic differentiation. All experiments were run on an NVidia GeForce RTX 2070 Super. Our code is open source and is included in the supplemental material.

In all of our experiments, we compare SIP training to standard neural network optimizers. To make this comparison as fair as possible, we try to avoid other factors that might prevent convergence, such as overfitting. Therefore, we perform our experiments with effectively infinite training sets, sampling target observations y^* on-the-fly. This setup ensures that all optimizers can, in principle, converge to zero loss. Since a numerical simulation of the forward problem is assumed to be available, unlimited amounts of synthetic training data can always be generated, further justifying this assumption. To check for possible biases introduced by this procedure, we also performed our experiments with finite training sets but did not observe a noticeable change in performance as long as the data set sizes were reasonable, e.g. 1000 samples for the characterization (sine) experiment.

B.1 Characterization with two-dimensional sine function

In this experiment, we consider the nonlinear process

$$\mathcal{P}(x) = \left(\frac{\sin(\hat{x}_1)}{\xi}, \xi \cdot \hat{x}_2 \right) \quad \text{with} \quad x\hat{x} = \gamma \cdot R_\phi \cdot x \quad \text{and} \quad R_\phi = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$$

where γ denotes a global scaling factor. We set $\gamma = 10$ as this value leads to the fastest convergence when training the network with traditional optimizers like Adam.

The conditioning factor ξ continuously scales any compact solution manifold, elongating it along one axis and compressing it along the other. Therefore, the range of values required for a solution manifold scales linearly with ξ, in both x_1 and x_2 if $\phi \neq 0$. To take this into account, we show the relative accuracy $\frac{||\mathrm{x}-\mathrm{x}^*||_2}{\xi}$ in Fig. 5d, as described in the main text. To measure $||x - x^*||_2$, we find the true solution x^* analytically using $\sin^{-1}$ and determining which solution is closest.

Saddle-free Newton For SIP training, we use a variant of Newton's method. Without modification, Newton's method also approaches maxima and saddle points which exist in this experiment. To avoid these, we project the Newton direction and gradient into the eigenspace of the Hessian, similar to [14]. There, we can easily flip the Newton update along eigenvectors of the Hessian to ensure that the update always points in the direction of decreasing loss.

Neural network training We sample target states $y^* = (y_1^*, y_2^*)$ uniformly in the range $[-1, 1]$ each, and feed y^* to a fully-connected neural network that predicts $x = (x_1, x_2)$. The network consists of three hidden layers with 32, 64 and 32 neurons, respectively, and uses ReLU activations. The final layer has no activation function and outputs two values that are interpreted as x. We always train the network with a batch size of 100 and chose the best learning rate for each optimizer. We determined the following learning rates to work the best for this problem: SIP training with Adam $\eta = 10^{-3}$, Adam $\eta = 10^{-3}$, SGD $\eta = 10^{-2}/\xi^2$, Adadelta $\eta = 3 \cdot 10^{-3}$, Adagrad $\eta = 3 \cdot 10^{-3}$, RMSprop $\eta = 3 \cdot 10^{-5}$. For Fig. 5d and e, we run each optimizer for 10 minutes for each sample point which is enough time for more than 50k iterations with all traditional optimizers and around 15k iterations with SIP training. We then average the last 10% of recorded distances for the displayed value.

B.2 Poisson's equation

We consider Poisson's equation, $\nabla^2 y = x$ where x is the initial state and y is the output of the simulator. We set up a two-dimensional simulation with 80 by 60 cubic cells. Our simulator computes $y = \mathcal{P}(x) = \nabla^{-2} x$ implicitly via the conjugate gradient method. The inverse problem consists of finding an initial value x^* for a given target y^* such that $\nabla^2 y^* = x^*$. We formulate this problem as minimizing $L(x) = \frac{1}{2}||\mathcal{P}(x) - y^*||_2^2 = \frac{1}{2}||\nabla^{-2}(x - x^*)||_2^2$. We now investigate the computed updates Δx of various optimization methods for this problem.

Gradient descent Gradient descent prescribes the update $\Delta x = -\eta \cdot \left(\frac{\partial L}{\partial x}\right)^T = -\eta \cdot \nabla^{-2}\left(y - y^*\right)$ which requires an additional implicit solve for each optimization step. This backward solve produces much larger values than the forward solve, causing GD-based methods to diverge from oscillations unless η is very small. We found that GD requires $\eta \leq 2 \cdot 10^{-5}$, while the momentum in Adam allows for larger η. For both GD and Adam, the optimization converges extremely slowly, making GD-based methods unfeasible for this problem.

SIP Gradients via analytic inversion Poisson's equation can easily be inverted analytically, yielding $x = \nabla^2 y$. Correspondingly, we formulate the update step as $\Delta x = -\eta \cdot \frac{\partial x}{\partial y} \cdot (y - y^*) = -\eta \cdot \nabla^2 \left(y - y^*\right)$ which directly points to x^* for $\eta = 1$. Here the Laplace operator appears in the computation of the optimization direction. This is much easier to compute numerically than the Poisson operator used by gradient descent. Consequently, no additional implicit solve is required for the optimization and the cost per iteration is less than with gradient descent. This computational advantages also carries over to neural network training where this method can be integrated into the backpropagation pipeline as a gradient.

Neural network training We first generate ground truth solutions x^* by adding fluctuations of varying frequencies with random amplitudes. From these x^*, we compute $y^* = \mathcal{P}(x^*)$ to form the set of target states $\mathcal{Y}$. This has the advantage that learning curves are representative of both test performance as well as training performance. The top of Fig. 9 shows some examples generated this way. All training methods except for the Fourier neural operators (FNO) use a a U-net [48] with a total of 4 resolution levels and skip connections. The network receives the feature map y^* as input. Max pooling is used for downsampling and bilinear interpolation for upsampling. After each downsampling or upsampling operation, two blocks consisting of 2D convolution with kernel size of 3x3, batch normalization and ReLU activation are performed. For training with AdaHessian, we also tested $\tanh$ for activation but observed no difference in performance. All of these convolutions output 16 feature maps and a final 1x1 convolution brings the output down to one feature map. The network contains a total of 37,697 trainable parameters. We use a mini-batch size of 128 for all methods.

For SGD and Adam training, the composite gradient of $\mathrm{NN} \circ \mathcal{P}$ is computed with TensorFlow or PyTorch, enabling an end-to-end optimization. The learning rate is set to $\eta = 10^{-3}$ with Adam and $\eta = 3 \cdot 10^{-12}$ for SGD. The extremely small learning rate for SGD is required to balance out the large gradients and is consistent with the behavior of gradient-descent optimization on single examples where an $\eta = 2 \cdot 10^{-5}$ was required. We use a typical value of 0.9 for the momentum of SGD and Adam.

For training with AdaHessian [55], we use the implementation from `torch-optimizer` [43] and found the best learning rate to be $\eta = 10^{-6}$ which, similar to SGD, is much smaller than what is used on well-conditioned problems. However for $\eta >= 10^{-4}$ AdaHessian diverges on the Poisson equation, independent of which activation function is used. The Hessian power is another hyperparameter of the AdaHessian optimizer and we set it to the standard value of 0.5 which converges much better than a value of 1.0.

For training with the Hessian-free optimizer [39], we use the `PyTorchHessianFree` implementation from `https://github.com/ltatzel/PyTorchHessianFree`. We use the generalized Gauss-Newton matrix to approximate the local curvature because it yields more stable results than the Hessian approximation in our tests. The optimizer uses a learning rate of $\eta = 1.0$ and adaptive damping during training. On this problem, the Hessian-free optimizer takes between 300 and 1700 seconds to compute a single update step. This is largely due to the CG loop, performing between 50 and 250 iterations for each update. The solution error drops to 900 within the plotted time frame in Fig. 6b, corresponding to less then 6 iterations. Convergence speed then slows continuously, reaching a MAE of 500 after about 6 hours.

For the training using Adam with SIP gradients, we compute Δx as described above and keep $\eta = 10^{-3}$. For each case, we set the learning rate to the maximum value that consistently converges. The learning curves for three additional random network initializations are shown at the bottom of Fig. 9, while Fig. 15 shows the computation time per iteration.

Training a Fourier neural operator network [34] on the same task requires a different network architecture. We use a standard 2D FNO architecture with width 32 and 12 modes along x and y. This results in a much larger network consisting of 1,188,353 parameters. Despite its size, its performance

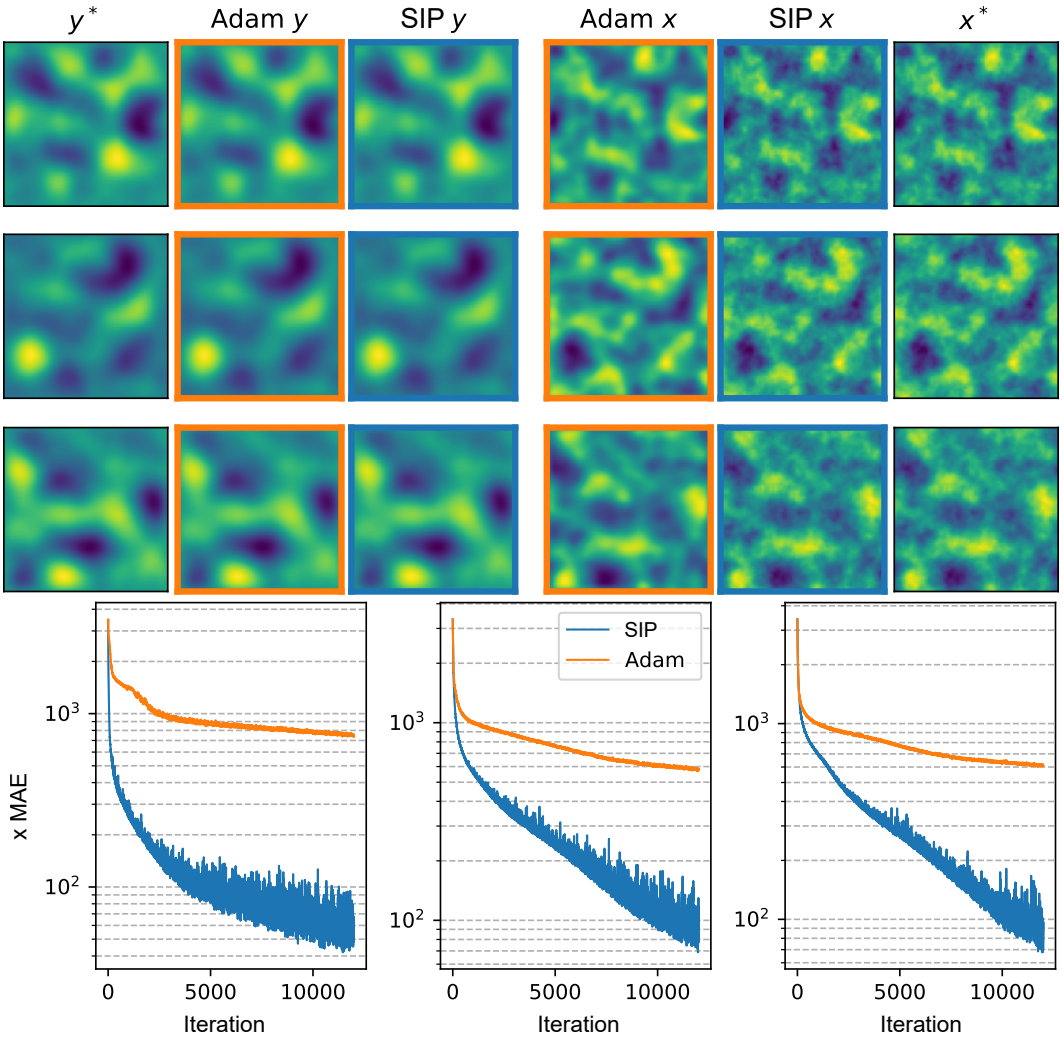

Figure 9: Inverse problems involving Poisson's equation. **Top:** Three examples from the data set, from left to right: observed target (y^*), simulated observations resulting from network predictions (Adam y, SIP y), predicted solutions (Adam x, SIP x), ground truth solution (x^*). Networks were trained for 12k iterations. **Bottom:** Neural network learning curves for three random network initializations, measured as $||x - x^*||_1$.

measured against wall clock time is superior to that of smaller versions we tested. We train the FNO using Adam with $\eta = 0.003$, which yields the best results, converging in a stable manner.

B.3 Heat equation

We consider a two-dimensional system governed by the heat equation $\frac{\partial u}{\partial t} = \nu \cdot \nabla^2 u$. Given an initial state $x = u_0$ at t_0, the simulator computes the state at a later time t_* via $y = u_* = \mathcal{P}(x)$. Exactly inverting this system is only possible for $t \cdot \nu = 0$ and becomes increasingly unstable for larger $t \cdot \nu$ because initially distinct heat levels even out over time, drowning the original information in noise. Hence the Jacobian of the physics $\frac{\partial y}{\partial x}$ is near-singular. In our experiment we set $t \cdot \nu = 8$ on a domain consisting of 64x64 cells of unit length. This level of diffusion is challenging, and diffuses most details while leaving the large-scale structure intact.

We apply periodic boundary conditions and compute the result in frequency space where the physics can be computed analytically as $\hat{y} = \hat{x} \cdot e^{-k^2(t_* - t_0)}$ where $\hat{y}_k \equiv \mathcal{F}(y)_k$ denotes the k-th element of the Fourier-transformed vector y. Here, high frequencies are dampened exponentially. The inverse problem can thus be written as minimizing $L(x) = ||\mathcal{P}(x) - y^*||_2^2 = ||\mathcal{F}^{-1}\left(\mathcal{F}(x) \cdot e^{-k^2(t_* - t_0)}\right) - y^*||_2^2$.

Gradient descent Using the analytic formulation, we can compute the gradient descent update as

$$\Delta x = -\eta \cdot \mathcal{F}^{-1}\left(e^{-k^2(t_* - t_0)} \mathcal{F}(y - y^*)\right).$$

GD applies the forward physics to the gradient vector itself, which results in updates that are stable but lack high frequency spatial information. Consequently, GD-based optimization methods converge slowly on this task after fitting the coarse structure and have severe problems in recovering high-frequency details. This is not because the information is fundamentally missing but because GD cannot adequately process high-frequency details.

Stable SIP gradients The frequency formulation of the heat equation can be inverted analytically, yielding $\hat{x}_k = \hat{y}_k \cdot e^{k^2(t_* - t_0)}$. This allows us to define the update

$$\Delta x = -\eta \cdot \mathcal{F}^{-1}\left(e^{k^2(t_* - t_0)} \mathcal{F}(y - y^*)\right).$$

Here, high frequencies are multiplied by exponentially large factors, resulting in numerical instabilities. When applying this formula directly to the gradients, it can lead to large oscillations in Δx. This is the opposite behavior compared to Poisson's equation where the GD updates were unstable and the SIP stable.

The numerical instabilities here can, however, be avoided by taking a probabilistic viewpoint. The observed values y contain a certain amount of noise n, with the remainder constituting the signal $s = y - n$. For the noise, we assume a normal distribution $n \sim \mathcal{N}(0, \epsilon \cdot y)$ with $\epsilon > 0$ and for the signal, we assume that it arises from reasonable values of x so that $y \sim \mathcal{N}(0, \delta \cdot e^{-k^2})$ with $\delta > 0$. With this, we can estimate the probability of an observed value arising from the signal using Bayes' theorem $p(s|v) = \frac{p(v|s) \cdot p(s)}{p(v|s) \cdot p(s) + p(v|n) \cdot p(n)}$ where we assume the priors $p(s) = p(n) = \frac{1}{2}$. Based on this probability, we dampen the amplification of the inverse physics which yields a stable inverse. Gradients computed in this way hold as much high-frequency information as can be extracted given the noise that is present. This leads to a much faster convergence and more precise solution than any generic optimization method.

Neural network training For training, we generate x^* by randomly placing between 4 and 10 hot rectangles of random size and shape in the domain and computing $y = \mathcal{P}(x^*)$. For the neural network, we use the same U-net and FNO architectures as in the previous experiment. We train all methods with a batch size of 128. A learning rate of $\eta = 10^{-3}$ yields the best results for SGD, Adam, AdaHessian, FNO and SIP training. Unlike the Poisson experiment, where such large learning rates lead to divergence due to the large gradient magnitudes, the heat equation produces relatively small and predictable gradients. Larger η can still result in divergence, especially with AdaHessian. For the Hessian-free optimizer, we use $\eta = 1.0$ with adaptive damping during training. Due to the

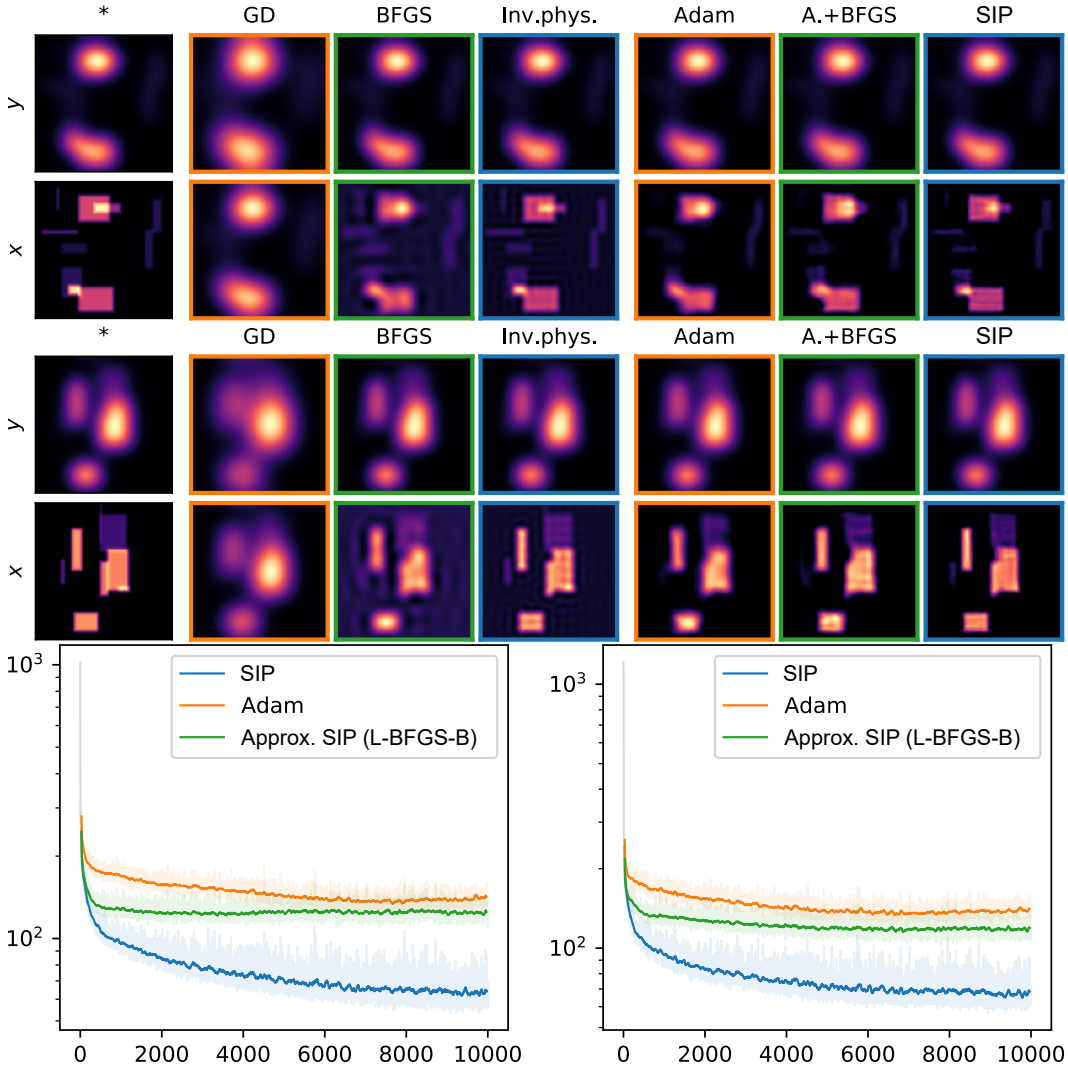

Figure 10: Inverse problems involving the heat equation. **Top:** Two examples from the data set. The top row shows observed target (y^*) and simulated observations resulting from inferred solutions. The bottom row shows the ground truth solution (x^*) and inferred solutions. From left to right: ground truth; gradient descent (GD), L-BFGS-B (BFGS) and inverse physics (Inv.phys.), running for 100 iterations each, starting with $x_0 = 0$; Networks trained for 10k iterations. **Bottom:** Neural network learning curves for two random network initializations, measured in terms of $||x - x^*||_1$.

more compute-intensive updates, we plot the running average over 8 mini-batches for Hessian-free in Fig. 6, instead of the usual 64.

Fig. 10 shows two examples from the data set, along with the corresponding inferred solutions, as well as the network learning curves for two network initializations. The measured computation time per iteration is shown in Fig. 15.

We perform additional hyperparameter studies on this experiment to better gauge how SIP training compares to traditional network optimization in a variety of settings. Fig. 11 shows the learning curves for a varying learning rate η and batch size b. The best performance of both methods is achieved at $\eta = 10^{-3}$ and $b = 128$.

Additionally, we evaluate the methods on finite data sets with sizes between 32 and 2048 examples while also varying the batch size. Fig. 12 shows the performance on both the training and the test set. Both SIP training and Adam exhibit overfitting under the same conditions. For data set sizes of 128 and less, both methods start to overfit early on. This is especially pronounced for large batch sizes where less randomness is involved. In the cases where the batch size is equal to or larger than the data set size, no mini-batching is used. We tile the data set where needed. For very small data sets, SIP training seems to exhibit stronger overfitting but this can be explained by its superior performance. The test performance of SIP training always surpasses the Adam variant, even on small data sets.

We also run iterative optimizers on individual examples of the data set. Fig. 13 shows the optimization curves of gradient descent and L-BFGS-B and compares them to the network predictions. L-BFGS-B converges faster than gradient descent but both iterative optimizers progress slowly on this inverse problem due to its ill-conditioned nature. After 500 iterations, L-BFGS-B matches the solution accuracy of the neural network trained with Adam. We have run both traditional optimizers for 1000 iterations, representing 102 seconds for BFGS and 36 seconds for gradient descent. This is about 1000 times longer than the network predictions, which finish within 64 ms.

B.4 Navier-Stokes equations

Here, we give additional details on the simulation, data generation, SIP gradients and network training procedure for the fluid experiment.

Simulation details We simulate the fluid dynamics using a direct numerical solver. We adopt the marker-in-cell (MAC) method [28, 27] which guarantees stable simulations even for large velocities or time increments. The velocity vectors are sampled in staggered form at the face centers of grid cells while the marker density is sampled at the cell centers. The initial velocity v_0 is specified at cell centers and resampled to a staggered grid for the simulation. Our simulation employs a second-order advection scheme [52] to transport both the marker and the velocity vectors. This step introduces significant amount of numerical diffusion which can clearly be seen in the final marker distributions. Hence, we do not numerically solve for adding additional viscosity. Incompressibility is achieved via Helmholz decomposition of the velocity field using a conjugate gradient solve.

Neither pressure projection nor advection are energy-conserving operations. While specialized energy-conserving simulation schemes for fluids exist [24, 42], we instead enforce energy conservation by normalizing the velocity field at each time step to the total energy of the previous time step. Here, the energy is computed as $E = \int_{\mathbb{R}^2} dx\, v(x)^2$ since we assume constant fluid density.

Data generation The data set consists of marker pairs $\{m_0, m_t\}$ which are randomly generated on-the-fly. For each example, a center position for m_0 is chosen on a grid of 64x64 cells. m_0 is then generated from discretized noise fluctuations to fill half the domain size in each dimension. The number of marked cells is random.

Next, a ground truth initial velocity v_0 is generated from three components. First, a uniform velocity field moves the marker towards the center of the domain to avoid boundary collisions. Second, a large vortex with random strength and direction is added. The velocity magnitude of the vortex falls off with a Gaussian function depending on the distance from the vortex center. Third, smaller-scale vortices of random strengths and sizes are added additionally perturb the flow fields. These are generated by assigning a random amplitude and phase to each frequency making up the velocity field. The range from which the amplitudes are sampled depends on the magnitude frequency.

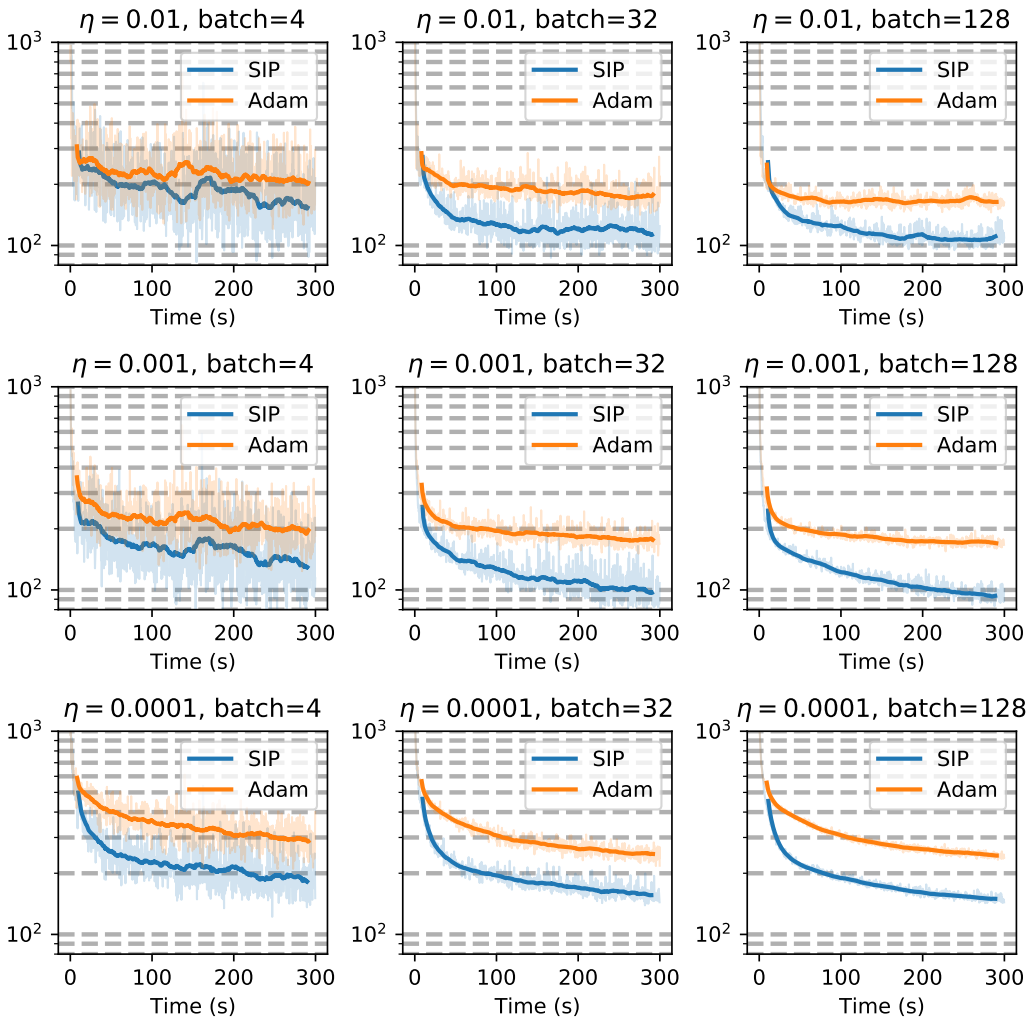

Figure 11: Hyperparameter study for the heat equation experiment. The learning rate η varies vertically from 10^{-4} to 10^{-2} and the batch size varies horizontally from 4 to 128 examples per mini-batch. The solid curves are averaged over 64 mini-batches. A learning rate of 10^{-3} with large batch sizes yields best performance.

Given m_0 and v_0, a ground truth simulation is run for $t = 2$ with $\Delta t = 0.25$. The resulting marker density is then used as the target for the optimization. This ensures that there exists a solution for each example.

Computation of SIP gradients To compute the SIP gradients for this example, we construct an explicit formulation $\hat{v}_0 = \mathcal{P}^{-1}(m_0, m_t \,|\, x_0)$ that produces an estimate for v_0 given an initial guess x_0 by locally inverting the physics. From this information, it fits the coarse velocity, i.e. the uniform velocity and the vortex present in the data. This use of domain knowledge, i.e., enforcing the translation and rotation components of the velocity field as a prior, is what allows it to produce a much better estimate of v_0 than the regular gradient. More formally, it assumes that the solution lies on a manifold that is much more low-dimensional than v_0. On the other hand, this estimator ignores the small-scale velocity fluctuations which limits the accuracy it can achieve. However, the difficulty of fitting the full velocity field without any assumptions outweighs this limitation. Nevertheless, GD could eventually lead to better results if trained for an extremely long time.

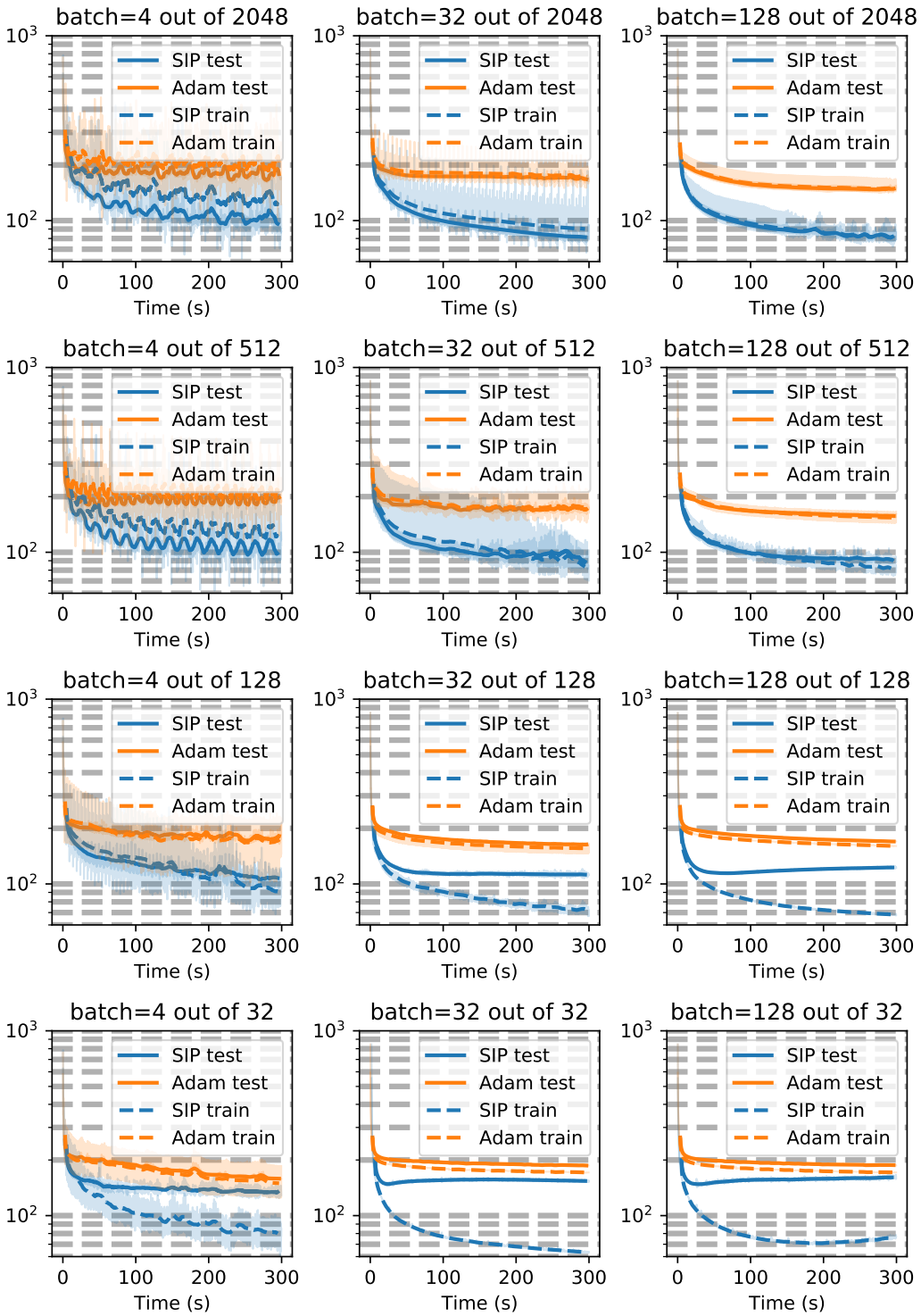

Figure 12: Heat equation experiment with different training set sizes. The training set size $|\mathcal{T}|$ varies vertically from 32 to 2048 and the batch size b varies horizontally from 4 to 128 examples per mini-batch. The test set is of fixed size 128. The solid curves are averaged over 64 mini-batches. Overfitting occurs in both SIP training and Adam when no mini-batches are used, $|\mathcal{T}| \leq b$, and for small b where the convergence is unstable.

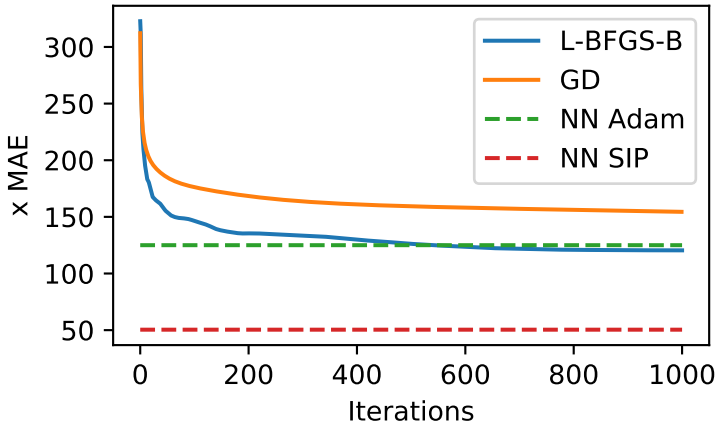

Figure 13: Iterative optimization of individual examples from a test set of 128 heat examples. Gradient descent (GD) and L-BFGS-B directly optimize the initial state x_0 (64x64 grid), independently for each example. The predictions of the trained and frozen neural networks (NN Adam, NN SIP) are evaluated on the same data set for reference.

To estimate the vortex strength, the estimator runs a reverse Navier-Stokes simulation. The reverse simulation is initialized with the marker $m_t^{\text{rev}} = m_t$ and velocity $v_t^{\text{rev}} = v^t$ from the forward simulation. The reverse simulation then computes m^{rev} and v^{rev} for all time steps by performing simulation steps with $\Delta t = -0.25$. Then, the update to the vortex strength is computed from the differences $m^{\text{rev}} - m$ at each time step and an estimate of the vortex location at these time steps.

Neural network training We train a U-net [48] similar to the previous experiments but with 5 resolution levels. The network contains a total of 49,570 trainable parameters. The network is given the observed markers m_0 and m_t, resulting in an input consisting of two feature maps. It outputs two feature maps which are interpreted as a velocity field sampled at cell centers.

The objective function is defined as $|\mathcal{F}(\mathcal{P}(x) - y^*)| \cdot w$ where $\mathcal{F}$ denotes the two-dimensional Fourier transform and w is a weighting vector that factors high frequencies exponentially less than low frequencies.

We train the network using Adam with a learning rate of 0.005 and mini-batches containing 64 examples each, using PyTorch's automatic differentiation to compute the weight updates. We found that second-order optimizers like L-BFGS-B yield no significant advantage over gradient descent, and typically overshoot in terms of high-frequency motions. Example trajectories and reconstructions are shown in Fig. 14 and performance measurements are shown in Fig. 15.

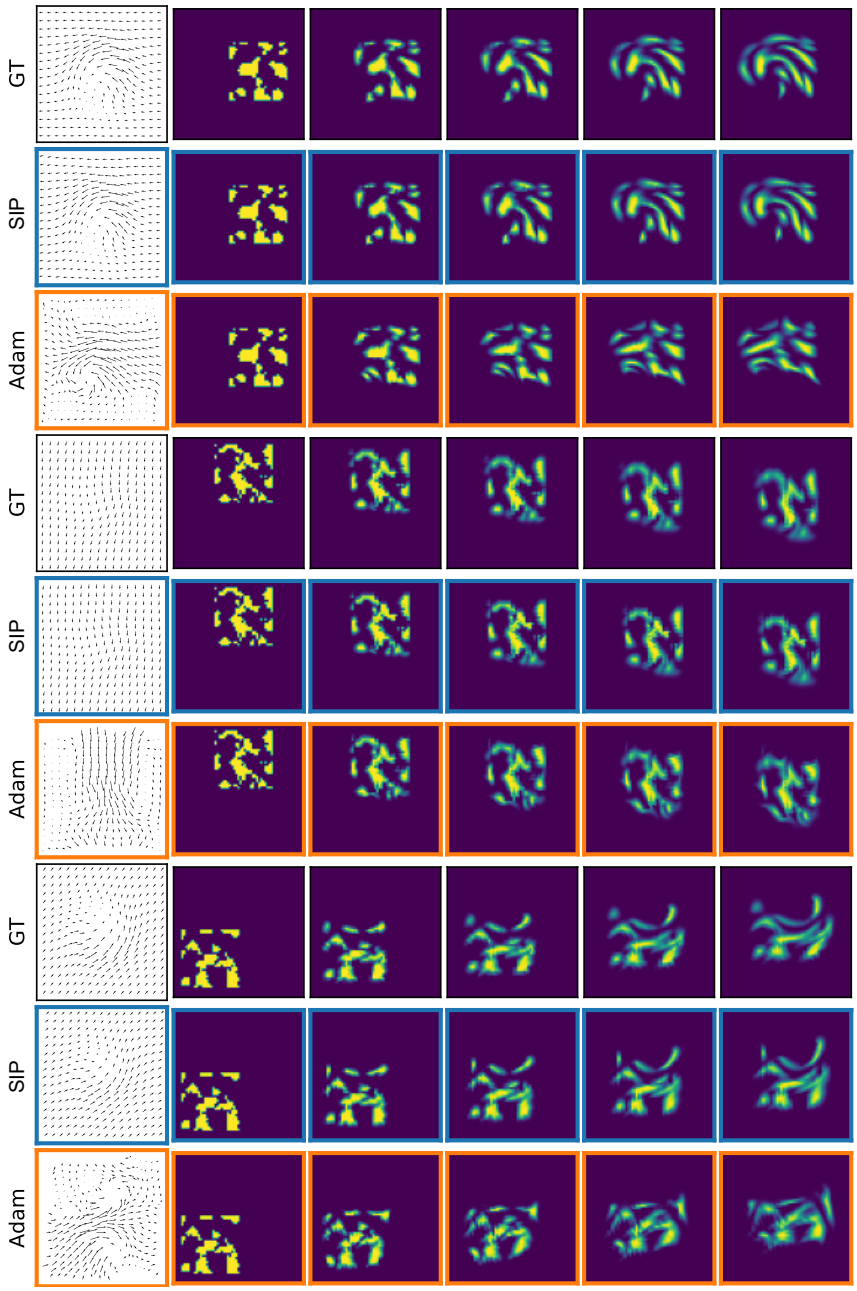

Figure 14: Three example inverse problems involving the Navier-Stokes equations. For each example, the ground truth (GT) and neural network reconstructions using Adam with SIP gradient (SIP) and pure Adam training (Adam) are displayed as rows. Each row shows the initial velocity $v_0 \equiv x$ as well as five frames from the resulting marker density sequence $m(t)$, at time steps $t \in \{0, 0.5, 1, 1.5, 2\}$. The differences of the Adam version are especially clear in terms of v_0.

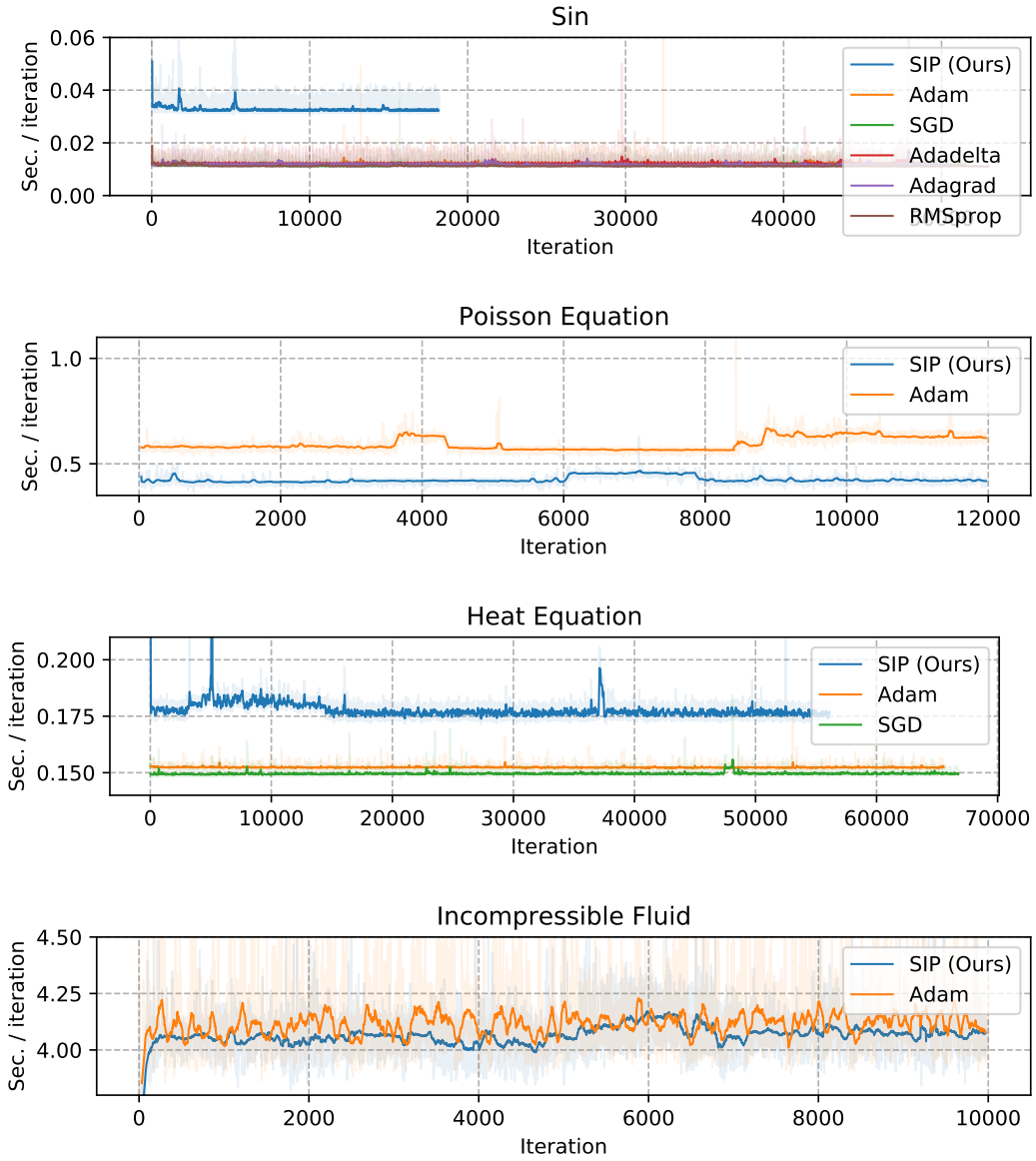

Figure 15: Measured time per neural network training iteration for all experiments, averaged over 64 mini-batches. Step times were measured using Python's `perf_counter()` function and include data generation, gradient evaluation and network update. In all experiments, the computational cost difference between the various gradients is marginal, affecting the overall training time by less than 10%.

C Additional Experiments

Here, we describe the additional experiments that were referenced in sections 1 and 2.

C.1 Wave packet localization

As we state in the introduction, one advantage of solution inference using neural networks is that no initial guess needs to be provided for each problem. Of course the network starts off with some initialization but we observe that the network can explore a much larger area of the solution space than any iterative solver. To our knowledge this claim has not been verified as of yet. However, as it is not directly relevant to our method, we do not discuss it in detail in the main text. Instead, we provide a simple example here.

The wave packet localization experiment is an instance of a generic curve fitting problem. The task is to find an offset parameter t_0 that results in least mean squared error between a noisy recorded curve and the model.

Data generation. We simulate an observed time series y^* from a random ground truth position $x^* = t_0$. Each time series contains 256 entries and consists of the wave packet and superimposed noise. For the wave packet, we sample $t_0 \in [25.6, 128)$ from a uniform distribution. The wave packet has the functional form

$$y(t) = A \cdot \sin(f \cdot (t - t_0)) \cdot \exp\left(-\frac{1}{2}\frac{(t - t_0)^2}{\sigma^2}\right)$$

where we set $A = 1$, $f = 0.7$ and $\sigma = 20$ constant for all data. For the noise, we superimpose random values sampled from the normal distribution $\mathcal{N}(0, 0.1)$ at each point.

Network architecture. We construct the neural network from convolutional blocks, followed by fully-connected layers, all using the ReLU activation function. The input is first processed by five blocks, each containing a max pooling operation and two 1D convolutions with kernel size 3. Each convolution outputs 16 feature maps. The downsampled result is then passed to two fully connected layers with 64 and 32 and 2 neurons, respectively, before a third fully-connected layer produces the predicted t_0 which is passed through a Sigmoid activation function and normalized to the range of possible values.

Training and fitting. We fit the data using L-BFGS-B with a centered initial guess ($t_0 = 76.8$) and the network output is offset by the same amount. Both network and L-BFGS-B minimize the squared loss $||y(t_0) - y^*||_2^2$ and the resulting performance curves along with example fits are shown in Fig. 16. We observe that L-BFGS-B manages to fit the wave packet perfectly when it is located very close to the center where the initial guess predicts it. When the wave packet is located slightly to either side, L-BFGS-B gets stuck in a local optimum that corresponds to an integer phase shift. When the wave packet is located further away from the initial guess, L-BFGS-B does not find it and instead fits the noise near the center.

The neural network is trained using Adam with learning rate $\eta = 10^{-3}$ and a batch size of 100. Despite the simpler first-order updates, the network learns to localize most wave packets correctly, outperforming L-BFGS-B after 30 to 40 training iterations. This improvement is possible because of the network's reparameterization of the problem, allowing for joint parameter optimization using all data. When the prediction for one example is close to a local optimum, updates from different examples can prevent it from converging to that sub-optimal solution.

Conclusion This example shows that neural networks have the capability to explore the solution space much better than iterative solvers, at least in some cases. Employing neural networks should therefore be considered, even when problems can be solved iteratively.

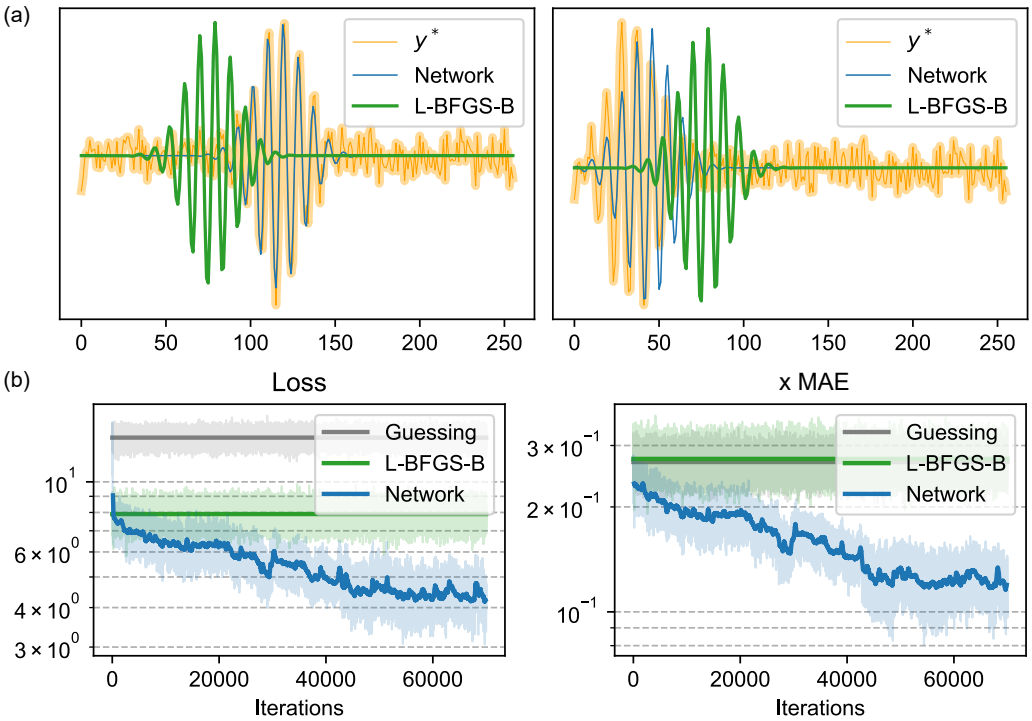

Figure 16: (a) Two examples from the wave packet data set, each showing the simulated data (orange), neural network prediction and L-BFGS-B fit. (b) Learning curves of the network trained to localize wave packets. The performance of L-BFGS-B and random guessing are evaluated on the same data for reference. The left graph shows the objective $||y - y^*||_2^2$ and the right graph shows the x-space (t_0) deviation from the true solution.

C.2 Gradient normalization for the exponential function

The task in this simple experiment is to learn to invert the exponential function $\mathcal{P}(x) = e^x$. As described in the text, both SGD and Adam converge very slowly on this task due to the gradients scaling linearly with e^x.

Gradient normalization We introduce a gradient normalization which first computes the gradient $\frac{\partial L}{\partial x}$. It then normalizes this adjoint vector for each example in the batch to unit length, $\Delta x = \text{sign}\left(\frac{\partial L}{\partial x}\right)$. Δx is then passed on to the network optimizer, replacing the standard adjoint vector for x. Like with SIP training, we implement this using an L_2 loss for the effective network objective $\tilde{L} = \frac{1}{2}||\text{NN}(y^*) - (\text{NN}(\circ) + \Delta x)||_2^2$.

Neural network training For training data, we sample x^* uniformly in the range $[-12, 0]$ and compute $y^* = e^{x^*}$. We train a fully-connected neural network with three hidden layers, each using the Sigmoid activation function and consisting of 16, 64 and 16 neurons, respectively. The network has a single input and output neuron. We train the network for 10k iterations with each method, using the learning rates $\eta = 10^{-3}$ for Adam, $\eta = 10^{-2}$ for SGD and $\eta = 10^{-3}$ for Adam with x normalization. Each mini-batch consists of 100 randomly sampled values.