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 $\exists \tau > 0 \,:\, \forall x \in \mathbb{R}^d \,:\, L_i(x) - L_i(x + U_i(x)) \geq \tau \left(L(x) - L(x^*)\right)$ 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 $I = \{x \in \mathbb{R}^d \,:\, L(x) < L(x_n)\}$.

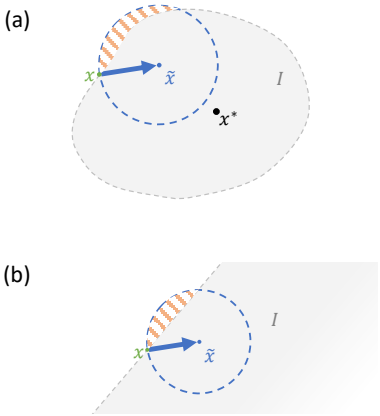

Figure 8: (a) Convergence visualization for the probable loss decrease in the case $n = 1$. (b) Zoomed view for small η. All shown objects are visualized in x space for one example i.

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, available at `https://github.com/tum-pbs/SIP`.

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 $\Delta x = -\eta \cdot \frac{\partial x}{\partial y} \cdot (y - y^*) = -\eta \cdot \nabla^2 (y - y^*)$ 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

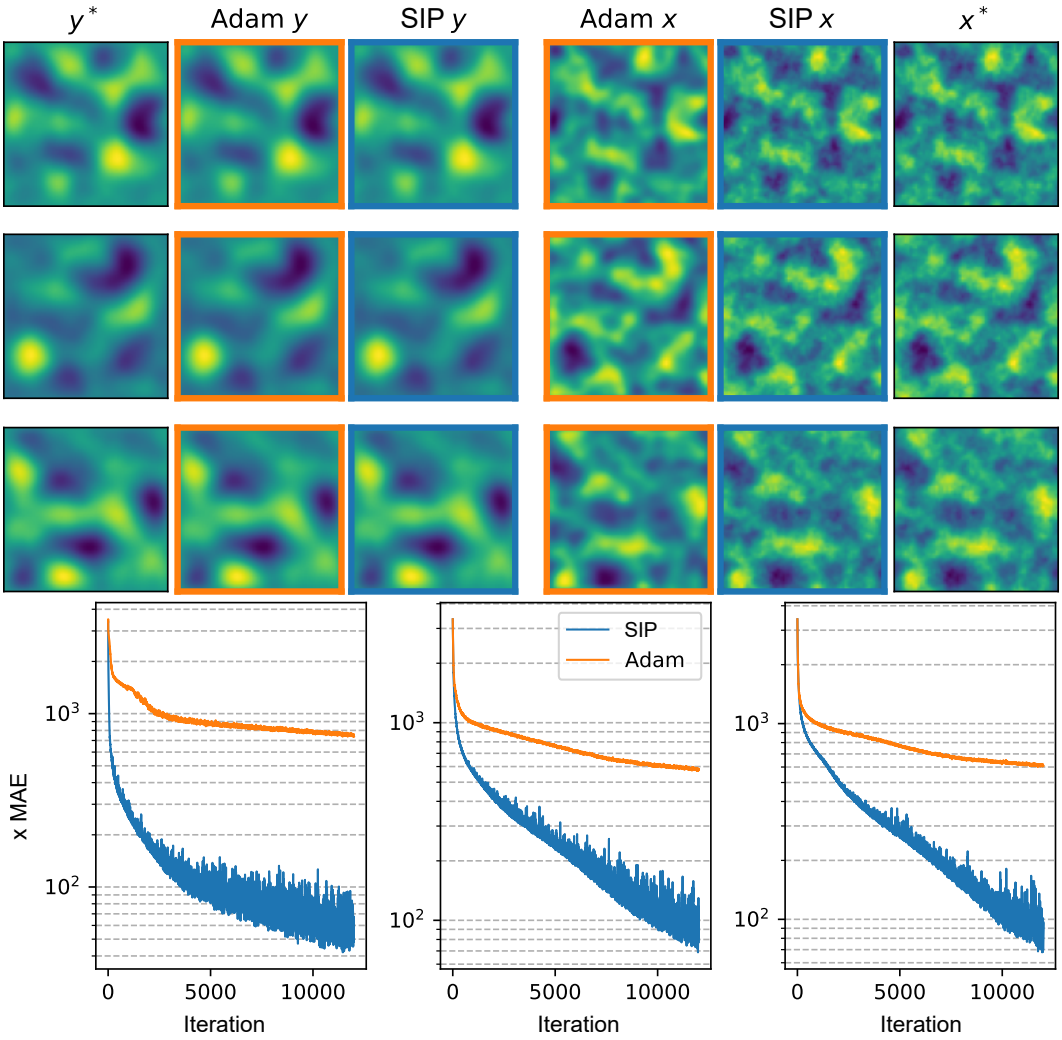

Figure 9: Inverse problems involving Poisson's equation. **Top:** Three examples from the data set, from left to right: observed target (y^*), simulated observations resulting from network predictions (Adam y, SIP y), predicted solutions (Adam x, SIP x), ground truth solution (x^*). Networks were trained 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 $L(x) = \frac{1}{2}||\mathcal{P}(x) - y^*||_2^2 = \frac{1}{2}||\mathcal{F}^{-1}\left(\mathcal{F}(x) \cdot e^{-k^2(t_*-t_0)}\right) - y^*||_2^2$.

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 $s \sim \mathcal{N}(0, \delta \cdot e^{-k^2})$ with $\delta > 0$. Then we can estimate the probability of an observed value arising from the signal using Bayes' theorem $p(s|y) = \frac{p(y \mid s) \cdot p(s)}{p(y \mid s) \cdot p(s) + p(y \mid n) \cdot p(n)}$ 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

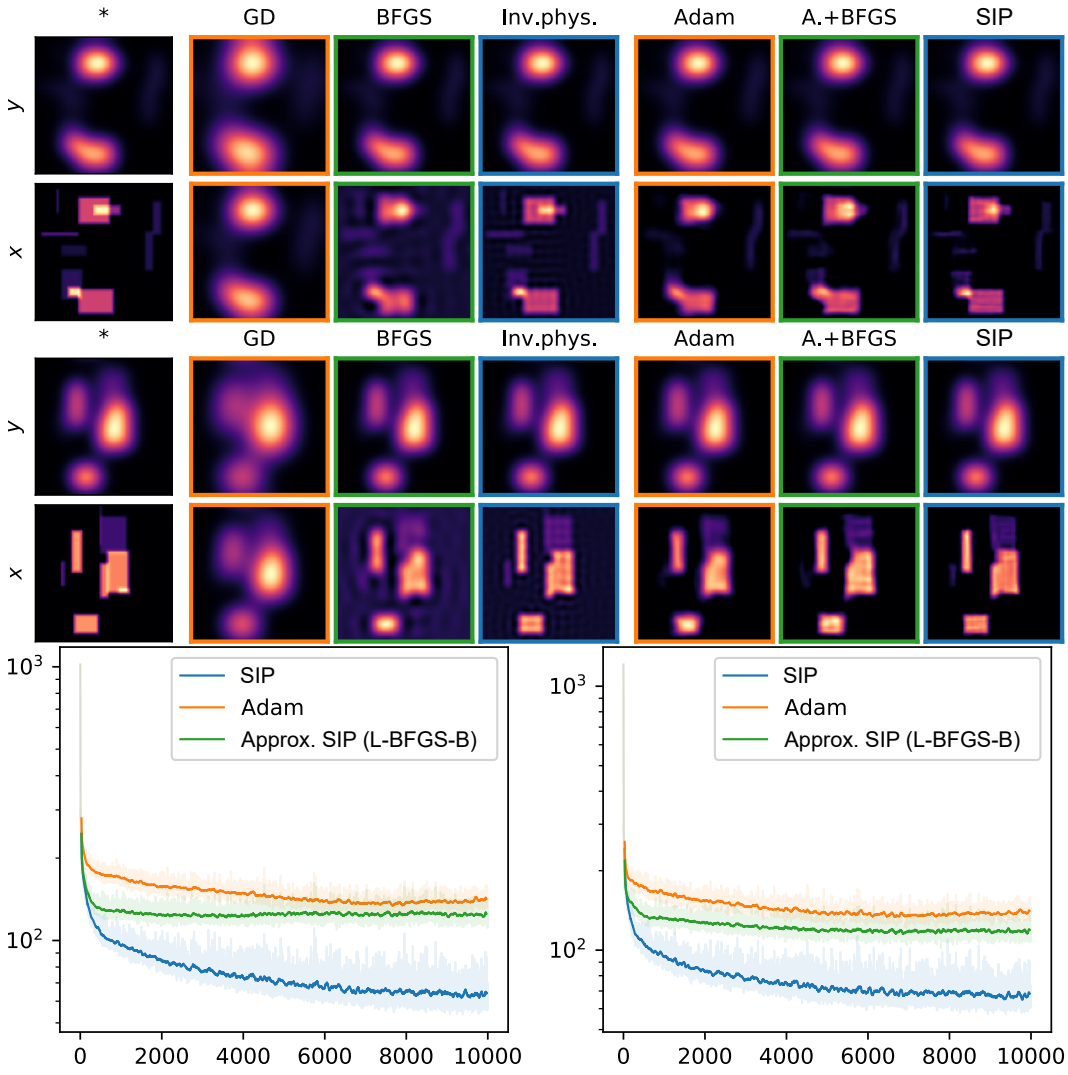

Figure 10: Inverse problems involving the heat equation. **Top:** Two examples from the data set. The top row shows observed target (y^*) and simulated observations resulting from inferred solutions. The bottom row shows the ground truth solution (x^*) and inferred solutions. From left to right: ground 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.

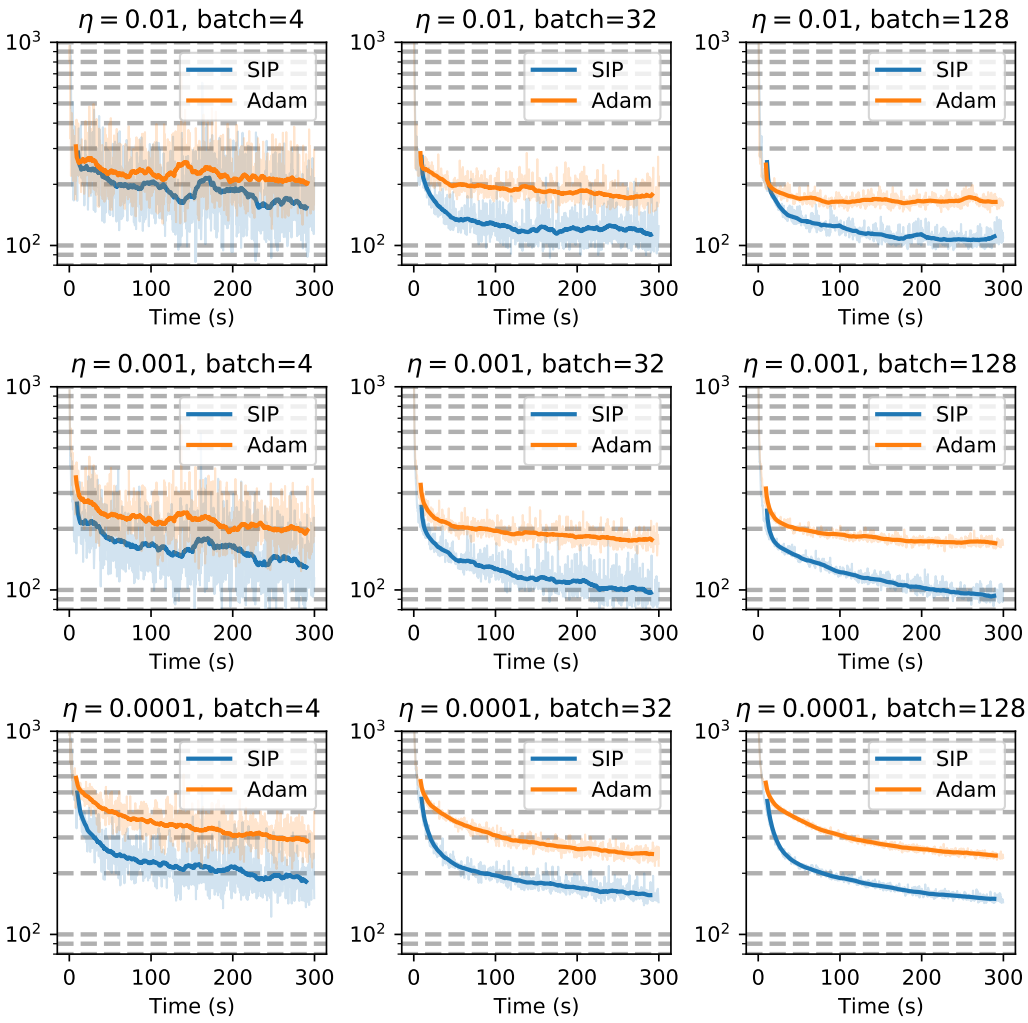

Figure 11: Hyperparameter study for the heat equation experiment. The learning rate η varies vertically from 10^{-4} to 10^{-2} and the batch size varies horizontally from 4 to 128 examples per mini-batch. The solid curves are averaged over 64 mini-batches. A learning rate of 10^{-3} with large batch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

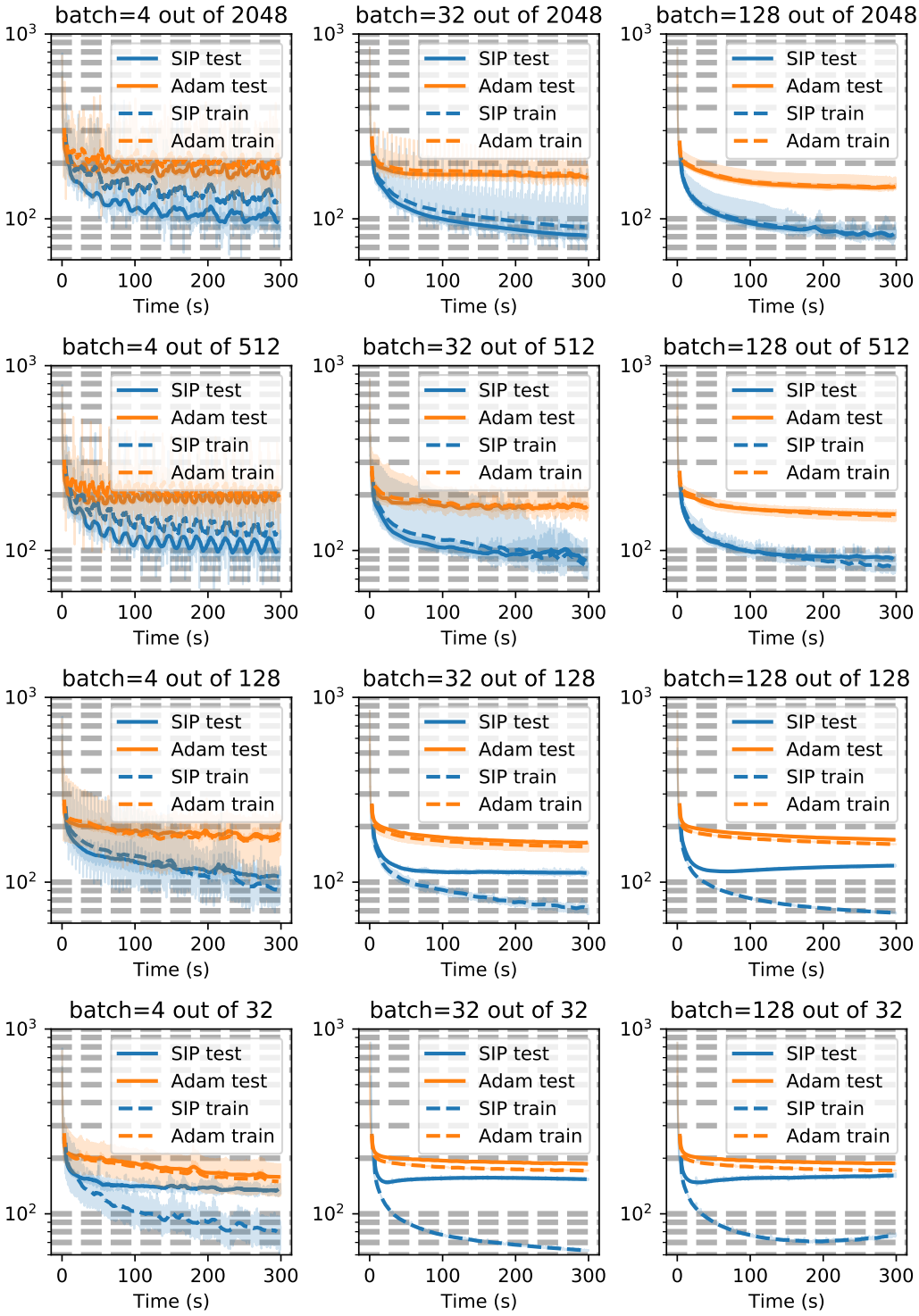

Figure 12: Heat equation experiment with different training set sizes. The training set size $|\mathcal{T}|$ varies vertically from 32 to 2048 and the batch size b varies horizontally from 4 to 128 examples per mini-batch. The test set is of fixed size 128. The solid curves are averaged over 64 mini-batches. 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.

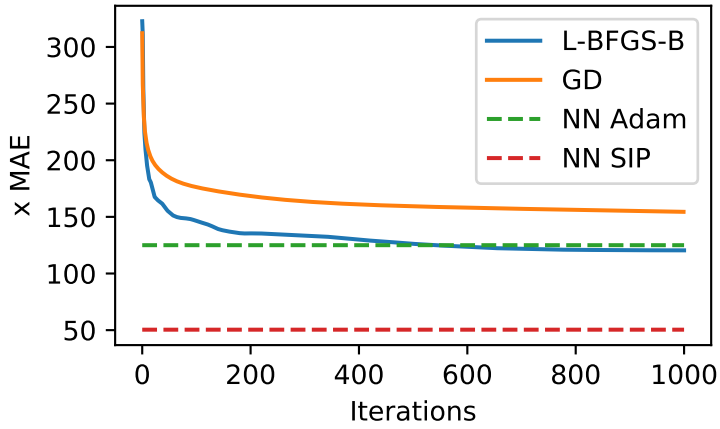

Figure 13: Iterative optimization of individual examples from a test set of 128 heat examples. Gradient descent (GD) and L-BFGS-B directly optimize the initial state x_0 (64x64 grid), independently for each example. The predictions of the trained and frozen neural networks (NN Adam, NN SIP) are evaluated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

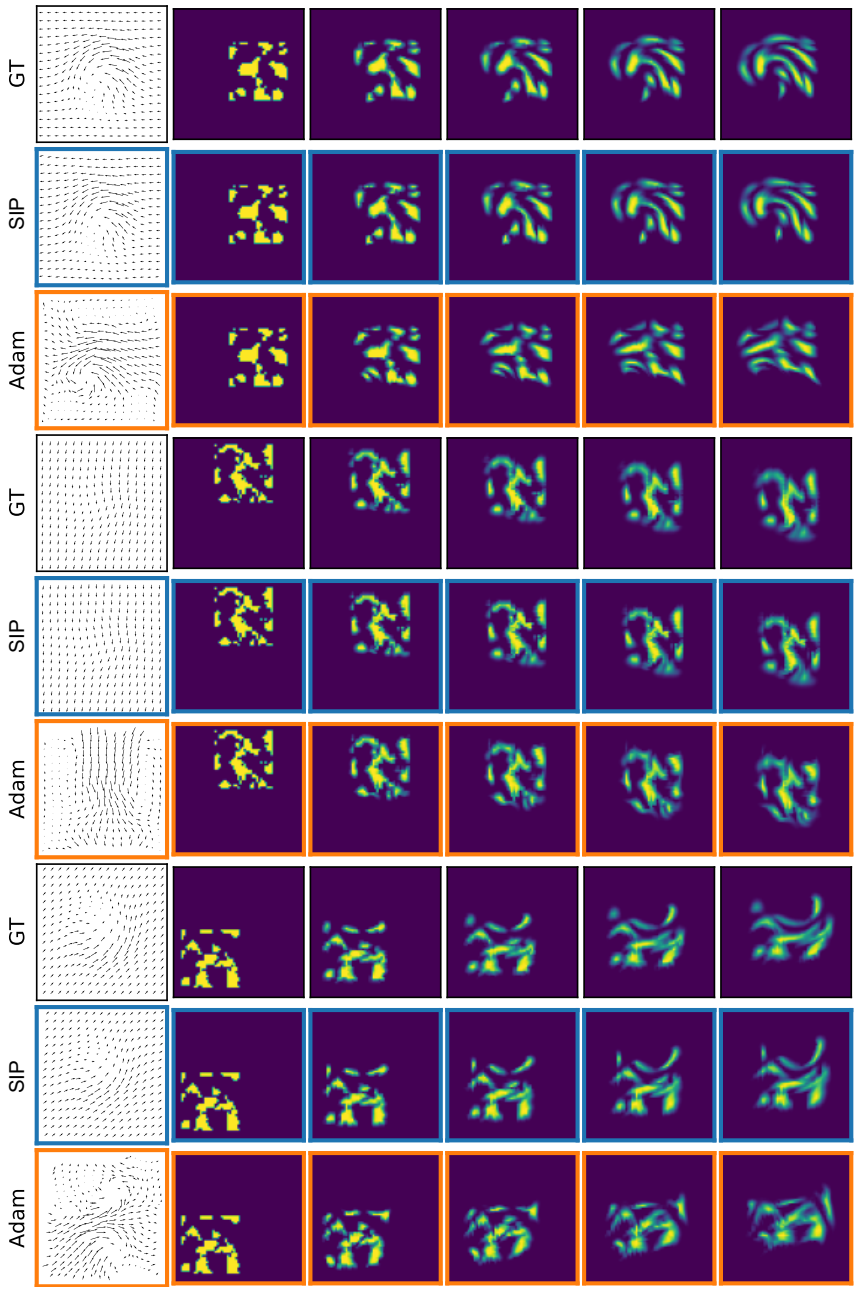

Figure 14: Three example inverse problems involving the Navier-Stokes equations. For each example, the ground truth (GT) and neural network reconstructions using Adam with SIP gradient (SIP) and pure Adam training (Adam) are displayed as rows. Each row shows the initial velocity $v_0 \equiv x$ as well 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.

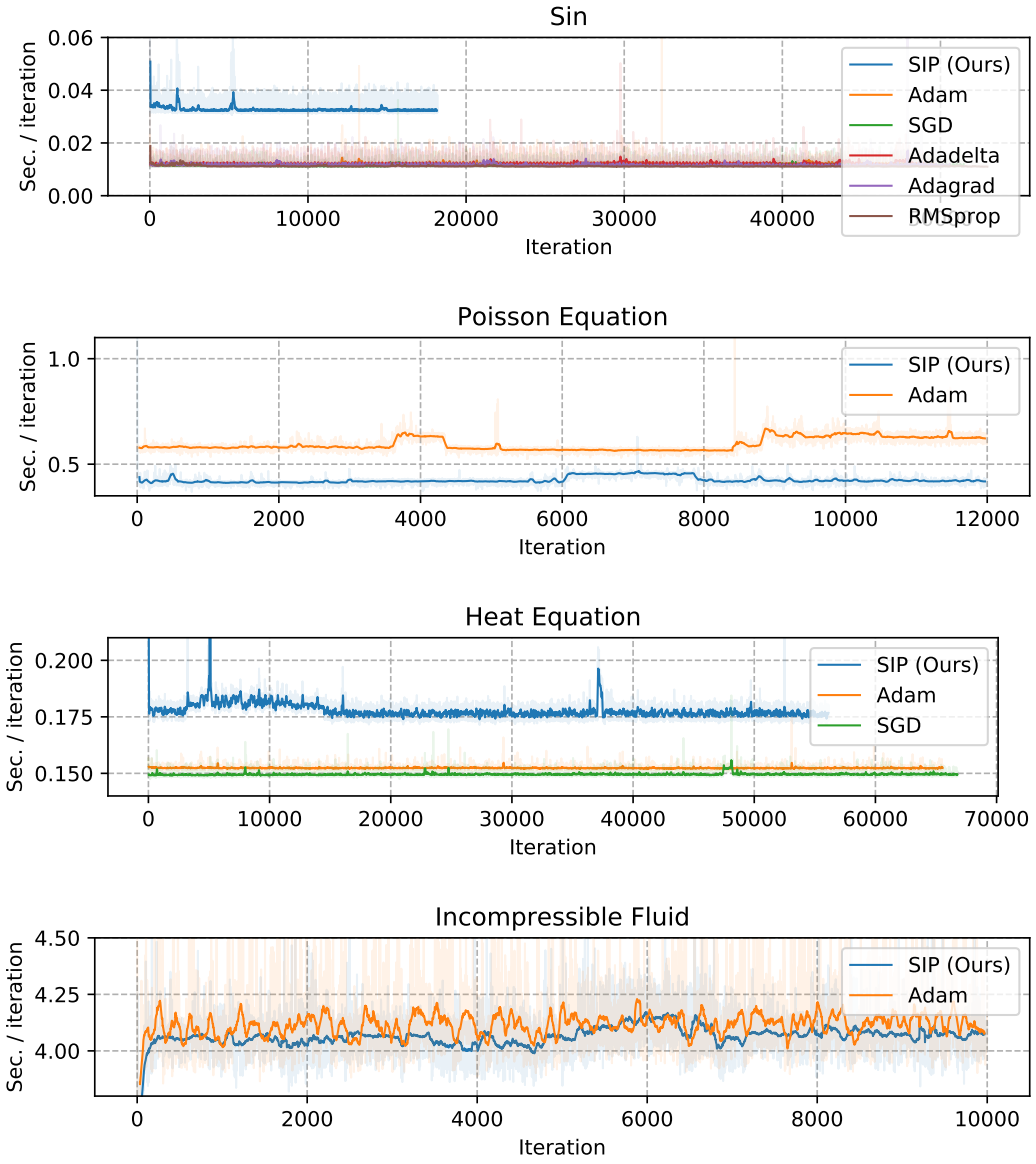

Figure 15: Measured time per neural network training iteration for all experiments, averaged over 64 mini-batches. Step times were measured using Python's `perf_counter()` function and include data generation, gradient evaluation and network update. In all experiments, the computational cost difference 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.

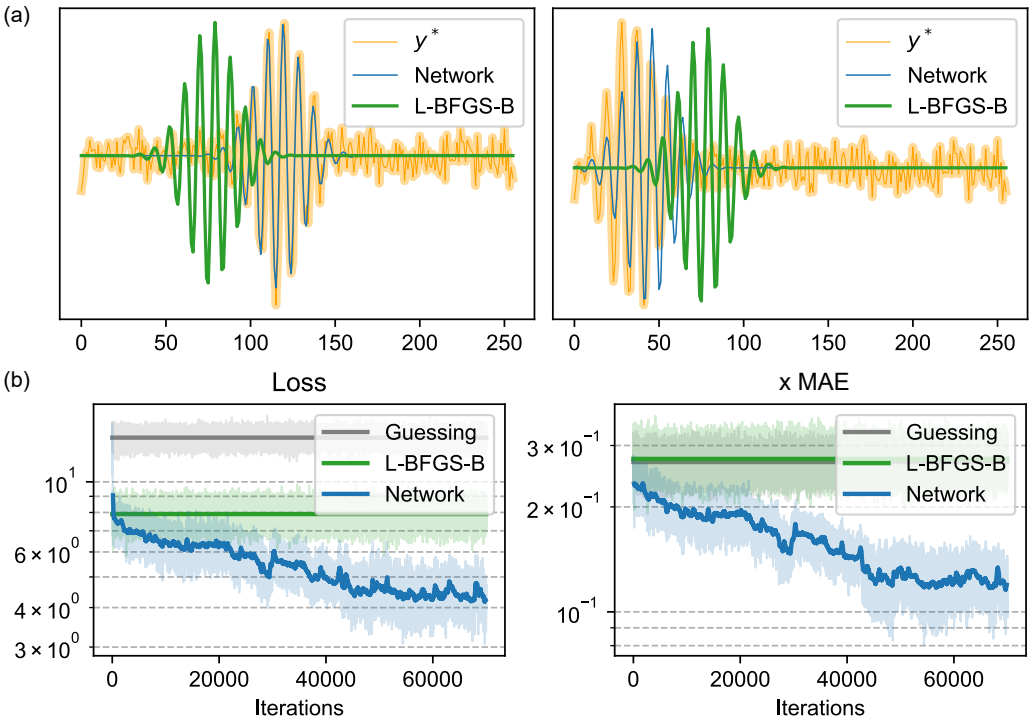

Figure 16: (a) Two examples from the wave packet data set, each showing the simulated data (orange), neural network prediction and L-BFGS-B fit. (b) Learning curves of the network trained to localize wave packets. The performance of L-BFGS-B and random guessing are evaluated on the same data for reference. 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.