CAN NEURAL NETWORKS IMPROVE CLASSICAL OPTIMIZATION OF INVERSE PROBLEMS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Finding the values of model parameters from data is an essential task in science. While iterative optimization algorithms like BFGS can find solutions to inverse problems with machine precision for simple problems, their reliance on local information limits their effectiveness for complex problems involving local minima, chaos, or zero-gradient regions. This study explores the potential for overcoming these limitations by jointly optimizing multiple examples. To achieve this, we employ neural networks to reparameterize the solution space and leverage the training procedure as an alternative to classical optimization. This approach is as versatile as traditional optimizers and does not require additional information about the inverse problems, meaning it can be added to existing general-purpose optimization libraries. We evaluate the effectiveness of this approach by comparing it to traditional optimization on various inverse problems involving complex physical systems, such as the incompressible Navier-Stokes equations. Our findings reveal significant improvements in the accuracy of the obtained solutions.

1 INTRODUCTION

Estimating model parameters by solving inverse problems (Tarantola, 2005) is a central task in scientific research, from detecting gravitational waves (George and Huerta, 2018) to controlling plasma flows (Maingi et al., 2019) to searching for neutrinoless double-beta decay (Agostini et al., 2013; Aalseth et al., 2018). Iterative optimization algorithms, such as limited-memory BFGS (Liu and Nocedal, 1989) or Gauss-Newton (Gill and Murray, 1978), are often employed for solving unconstrained parameter estimation problems (Press et al., 2007). These algorithms offer advantages such as ease of use, broad applicability, quick convergence, and high accuracy, typically limited only by noise in the observations and floating point precision. However, they face several fundamental problems that are rooted in the fact that these algorithms rely on local information, i.e., objective values $L(x_k)$ and derivatives close to the current solution estimate x_k, such as the gradient $\partial L/\partial x|_{x_k}$ and the Hessian matrix $\partial^2 L/\partial x^2|_{x_k}$. Acquiring non-local information can be done in low-dimensional solution spaces, but the curse of dimensionality prevents this approach for high-dimensional problems. These limitations lead to poor performance or failure in various problem settings:

- *Local optima* attract the optimizer in the absence of a counter-acting force. Although using a large step size or adding momentum to the optimizer can help to traverse small local minima, local optimizers are fundamentally unable to avoid this issue.
- *Flat regions* can cause optimizers to become trapped along one or multiple directions. Higher-order solvers can overcome this issue when the Hessian only vanishes proportionally with the gradient, but all local optimizers struggle in zero-gradient regions.
- *Chaotic regions*, characterized by rapidly changing gradients, are extremely hard to optimize. Iterative optimizers typically decrease their step size to compensate, which prevents the optimization from progressing on larger scales.

In many practical cases, a *set* of observations is available, comprising many individual parameter estimation problems, e.g., when repeating experiments multiple times or collecting data over a time frame (Carleo et al., 2019; Delaquis et al., 2018; George and Huerta, 2018; Agostini et al., 2013; Murase et al., 2013) and, even in the absence of many recorded samples, synthetic data can be generated to supplement the data set. Given such a set of inverse problems, we pose the question:

Can we find better solutions x_i to general inverse problems by optimizing them jointly instead of individually, without requiring additional information about the problems?

To answer this question, we employ neural networks to formulate a joint optimization problem. Neural networks as general function approximators are a natural way to enable joint optimization of multiple a priori independent examples. They have been extensively used in the field of machine learning (Goodfellow et al., 2016), and a large number of network architectures have been developed, from multilayer perceptrons (MLPs) (Haykin, 1994) to convolutional networks (CNNs) (Krizhevsky et al., 2012) to transformers (Vaswani et al., 2017). Overparameterized neural network architectures typically smoothly interpolate the training data (Belkin et al., 2018; Balestriero et al., 2021), allowing them to generalize, i.e., make predictions about data the network was not trained on.

It has been shown that this generalization capability or *inductive bias* benefits the optimization of individual problems with grid-like solution spaces by implicitly adding a prior to the optimization based on the network architecture (Ulyanov et al., 2018; Hoyer et al., 2019). However, these effects have yet to be investigated for general inverse problems or in the context of joint optimization. We propose using the training process of a neural network as a drop-in component for traditional optimizers like BFGS without requiring additional data, configuration, or tuning. Instead of making predictions about new data after training, our objective is to solve only the given problems, i.e., the training itself produces the solutions to the inverse problems, and the network is never used for inference. These solutions can also be combined with an iterative optimizer to improve accuracy. Unlike related machine learning applications (Kim et al., 2019; Sanchez-Gonzalez et al., 2020; Stachenfeld et al., 2021; Rasp and Thuerey, 2021; Schnell et al., 2022; Holl et al., 2021; Schenck and Fox, 2018; Ren et al., 2020; Allen et al., 2022), where a significant goal is accelerating time-intensive computations, we accept a higher computational demand if the resulting solutions are more accurate.

To quantify the gains in accuracy that can be obtained, we compare this approach to classical optimization as well as related techniques on four experiments involving difficult inverse problems: (i) a curve fit with many local minima, (ii) a billiards-inspired rigid body simulation featuring zero-gradient areas, (iii) a chaotic system governed by the Kuramoto–Sivashinsky equation and (iv) an incompressible fluid system that is only partially observable. We compare joint optimization to direct iterative methods and related techniques in each experiment.

2 Related work

Neural networks have become popular tools to model physical processes, either completely replacing physics solvers (Kim et al., 2019; Sanchez-Gonzalez et al., 2020; Stachenfeld et al., 2021; Rasp and Thuerey, 2021) or improving them (Tompson et al., 2017; Um et al., 2020; Kochkov et al., 2021). This can improve performance since network evaluations and solvers may be run at lower resolution while maintaining stability and accuracy. Additionally, it automatically yields a differentiable forward process which can then be used to solve inverse problems (Schenck and Fox, 2018; Ren et al., 2020; Allen et al., 2022), similar to how style transfer optimizes images (Gatys et al., 2016).

Alternatively, neural networks can be used as regularizers to solve inverse problems on sparse tomography data (Li et al., 2020) or employed recurrently for image denoising and super-resolution (Putzky and Welling, 2017). Recent works have also explored them for predicting solutions to inverse problems (Holl et al., 2021; Schnell et al., 2022) or aiding in finding solutions (Khalil et al., 2017; Dai et al., 2021). In these settings, neural networks are trained offline and then used to infer solutions to new inverse problems, eliminating the iterative optimization process at test time.

Underlying many of these approaches are differentiable simulations required to obtain gradients of the inverse problem. These can be used in iterative optimization or to train neural networks. Many recent software packages have demonstrated this use of differentiable simulations, with general frameworks (Hu et al., 2020; Schoenholz and Cubuk, 2019; Holl et al., 2020) and specialized simulators (Takahashi et al., 2021; Liang et al., 2019).

Physics-informed neural networks (Raissi et al., 2019) encode solutions to optimization problems in the network weights themselves. They model a continuous solution to an ODE or PDE and are trained by formulating a loss function based on the differential equation, and have been explored for a variety of directions (Yang et al., 2019; Lu et al., 2021; Krishnapriyan et al., 2021). However, as these approaches rely on loss terms formulated with neural network derivatives, they do not

apply to general inverse problems. The training process of neural networks themselves can also be framed as an inverse problem, and employing learning models to aid this optimization is referred to as *meta-learning* (Vilalta and Drissi, 2002). However, due to the large differences, meta-learning algorithms strongly differ from methods employed for inverse problems in physics.

3 REPARAMETERIZING INVERSE PROBLEMS WITH NEURAL NETWORKS

We consider a set of n similar inverse problems where we take *similar* to mean we can express all of them using a function $F(\xi_i \,|\, x_i)$ conditioned on a problem-specific vector x_i with $i = 1, ..., n$. Each inverse problem then consists of finding optimal parameters ξ_i^* such that a desired or observed output y_i is reproduced, i.e.

$$\xi_i^* = \arg\min_{\xi_i} \mathcal{L}(F(\xi_i \,|\, x_i), y_i), \tag{1}$$

where $\mathcal{L}$ denotes an error measure, such as the squared L^2 norm $||\cdot||_2^2$. We assume that F is differentiable and can be approximately simulated, i.e., the observed output y_i may not be reproducible exactly using F due to hidden information or stochasticity.

A common approach to finding ξ_i^* is performing a nonlinear optimization, minimizing $\mathcal{L}$ using the gradients $\frac{\partial \mathcal{L}}{\partial F}\frac{\partial F}{\partial \xi_i}$. In strictly convex optimization, many optimizers guarantee convergence to the global optimum in these circumstances. However, when considering more complex problems, generic optimizers often fail to find the global optimum due to local optima, flat regions, or chaotic regions. Trust region methods (Yuan, 2000) can be used on low-dimensional problems but scale poorly to higher-dimensional problems. Without further domain-specific knowledge, these methods are limited to individually optimizing all n inverse problems.

Instead of improving the optimizer itself, we want to investigate whether better solutions can be found by jointly optimizing all problems. However, without domain-specific knowledge, it is unknown which parameters of ξ_i are shared among multiple problems. We therefore first reparameterize the full solution vectors ξ_i using a set of functions $\hat{\xi}_i$, setting $\xi_i \equiv \hat{\xi}_i(\theta)$ where θ represents a set of shared parameters. With this change, the original parameters ξ_i become functions of θ, allowing θ to be jointly optimized over all problems. Here, the different $\hat{\xi}_i$ can be considered transformation functions mapping θ to the actual solutions ξ_i, similar to transforming Cartesian to polar coordinates. Second, we sum the errors of all examples to define the overall objective function $L = \sum_{i=1}^{n} \mathcal{L}_i$.

For generality, all $\hat{\xi}_i(\theta)$ should be able to approximate arbitrary functions. We implement them as an artificial neural network $\mathcal{N}$ with weights θ: $\hat{\xi}_i(\theta) \equiv \mathcal{N}(x_i, y_i \,|\, \theta)$. Inserting these changes into Eq. 1 yields the reparameterized optimization problem

$$\xi_i^* = \hat{\xi}_i(\theta^*)\,, \quad \theta^* = \text{argmin}_\theta \sum_{i=1}^{n} \mathcal{L}(F(\mathcal{N}(x_i, y_i \,|\, \theta) \,|\, x_i), y_i). \tag{2}$$

We see that the joint optimization with reparameterization strongly resembles standard formulations of neural network training where (x_i, y_i) is the input to the network and $F \circ L$ represents the effective loss function. However, from the viewpoint of optimizing inverse problems, the network is not primarily a function of (x_i, y_i) but rather a set of transformation functions of θ, each corresponding to a fixed and discrete (x_i, y_i). Figure 1 shows the computational graph corresponding to Eq. 2.

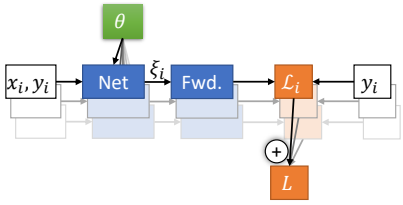


Figure 1: Reparameterized optimization

While the tasks of optimizing inverse problems and learning patterns from data may seem unrelated at first, there is a strong connection between the two. The inductive bias of a chosen network architecture, which enables generalization, also affects the update direction of classical optimizers under reparameterization. This can be seen most clearly if we consider gradient descent steps where the updates are $\Delta\xi_i = -\eta \frac{\partial \mathcal{L}_i}{\partial \xi_i}$ with step size η. After reparameterization, the updates are $\Delta\theta = -\eta \sum_i \frac{\partial \mathcal{L}_i}{\partial \xi_i}\frac{\partial \mathcal{N}}{\partial \theta}$. As we can see, $\frac{\partial \mathcal{N}}{\partial \theta}$ now contributes a large part to the update direction, allowing for cross-talk between the different optimization problems.

Despite the similarities to machine learning, the different use case of this setup leads to differences in the training procedure. For example, while overfitting is usually seen as undesirable in machine learning, we want the solutions to our inverse problems to be as accurate as possible, i.e. we want to "overfit" to the data. Consequently, we do not have to worry about the curvature at θ^* and will not use mini-batches for training the reparameterization network.

Supervised training. Our main goal is obtaining an optimization scheme that works exactly like classical optimizers, only requiring the forward process F, x_i in the form of a numerical simulator, and desired outputs y_i. However, if we additionally have a prior on the solution space $P(\xi)$, we can generate synthetic training data $\{(x_j, y_i), \xi_j\}$ with $y_j = F(x_j, \xi_j)$ by sampling $\xi_i \sim P(\xi)$. Using this data set, we can alternatively train $\mathcal{N}$ with the supervised objective

$$\tilde{L} = \sum_j ||\mathcal{N}(x_j, y_j) - \xi_j||_2^2. \tag{3}$$

Since $\mathcal{N}$ has the same inputs and outputs, we can use the same network architecture as above and the solutions to the original inverse problems can be obtained as $\xi_i = \mathcal{N}(x_i, y_i)$. While this method requires domain knowledge in the form of the distributions $P(x)$ and $P(\xi)$, it has the distinct advantage of being independent of the characteristics of F. For example, if F is chaotic, directly optimizing through F can yield very large and unstable gradients, while the loss landscape of $\tilde{L}$ can still be smooth. However, we cannot expect the inferred solutions to be highly accurate as the network is not trained on the inverse problems we want to solve and, thus, has to interpolate. Additionally, this method is only suited to unimodal problems, i.e. inverse problems with a unique global minimum. On multimodal problems, the network cannot be prevented from learning an interpolation of possible solutions, which may result in poor accuracy.

Refinement Obtaining a high accuracy on the inverse problems of interest is generally difficult when the training set size is limited, which can result in suboptimal solutions. This is especially problematic when the global minima are narrow and no direct feedback from F is available, as in the case of supervised training. To ensure that all learned methods have the potential to compete with gradient-based optimizers like BFGS, we pass the solution estimates for ξ to a secondary refinement stage where they are used as an initial guess for BFGS. The refinement uses the true gradients of F to find a nearby minimum of $\mathcal{L}$.

4 EXPERIMENTS

We perform a series of numerical experiments to test the convergence properties of the reparameterized joint optimization. An overview of the experiments is given in Tab. 1 and experimental details can be found in Appendix B. An additional experiment is given in Appendix B.6. We run each experiment and method multiple times, varying the neural network initializations and data sets to obtain statistically significant results.

To test the capabilities of the algorithms as a black-box extension of generic optimizers, all experiments use off-the-shelf neural network architectures and only require hyperparameter tuning in terms of decreasing the Adam (Kingma and Ba, 2015) learning rate until stable convergence is reached. We then compare the reparameterized optimization to BFGS (Liu and Nocedal, 1989), a popular classical solver for unconstrained optimization problems, and to the neural adjoint method, which has been shown to outperform various other neural-network-based approaches for solving inverse problems (Ren et al., 2020).

Table 1: Overview of numerical experiments.

Experiment	$\nabla = 0$ areas	Chaotic	x_i known	$P(\xi)$ known
Wave packet localization	No	No	No	Yes
Billiards	Yes	No	Yes	No
Kuramoto–Sivashinsky	No	Yes	Yes	Yes
Incompr. Navier-Stokes	No	Yes	No	Yes
Rototic arm (B.6)	No	No	Yes	Yes

Neural adjoint The neural adjoint method relies on an approximation of the forward process by a surrogate neural network $S(x_i, \xi_i \mid \theta)$. We first train the surrogate on an independent data set generated from the same distribution as the inverse problems and contains many examples. We use the same examples as for the supervised approach outlined above but switch the labels to match the network design, $\{(x_i, \xi_i), y_i\}$. After training, the weights θ are frozen and BFGS is used to optimize ξ_i on the proxy process $\tilde{F}(\xi_i \mid x_i) = S(\xi_i, x_i) + B(\xi_i)$ where B denotes a boundary loss term (see Appendix A). With the loss function $\mathcal{L}$ from Eq. 1, this yields the effective objective $\mathcal{L}(F(\xi_i \mid x_i), y_i)$ for solving the inverse problems. Like with the other methods, the result of the surrogate optimization is then used as a starting point for the refinement stage described above.

4.1 Wave packet localization

First, we consider a 1D curve fit. A noisy signal $u(t)$ containing a wave packet centered at t_0 is measured, resulting in the observed data $u(t) = A \cdot \sin(t - t_0) \cdot \exp(-(t - t_0)^2/\sigma^2) + \epsilon(t)$ where $\epsilon(t)$ denotes random noise and $t = 1, ..., 256$. An example waveform is shown in Fig. 2a. For fixed A and σ, the task is to locate the wave packed, i.e. retrieve t_0. This task is difficult for optimization algorithms because the loss landscape (Fig. 2b) contains many local optima that must be traversed. This results in alternating gradient directions when traversing the parameter space, with maximum magnitude near the correct solution.

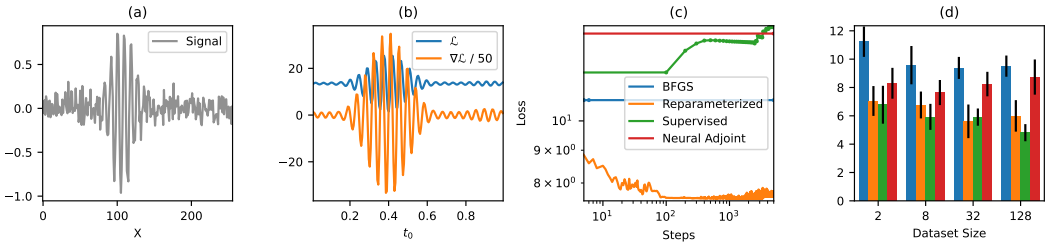


Figure 2: Wave packet localization. **(a)** Example waveform $u(t)$, **(b)** corresponding loss and gradient landscape for t_0, **(c)** optimization curves without refinement, **(d)** refined loss L/n by the number of examples n, mean and standard deviation over multiple network initializations and data sets.

We generate the inverse problems by sampling random t_0 and $\epsilon(t)$ from ground truth prior distributions and simulating the corresponding outputs $u(t) = F\epsilon(t) \mid t_0)$. Because the noise distribution $\epsilon(t)$ is not available to any of the optimization methods, a perfect solution with $\mathcal{L} = 0$ is impossible.

Fig. 2c shows the optimization process. Iterative optimizers like BFGS get stuck in local minima quickly on this task. In most examples, BFGS moves a considerable distance in the first iteration and then quickly halts. However, due to the oscillating gradient directions, this initial step is likely to propel the estimate away from the global optimum, leading many solutions to lie further from the actual optimum than the initial guess.

The neural adjoint method finds better solutions than BFGS for about a third of examples for $n = 256$ (see Tab. 2). In many cases, the optimization progresses towards the boundary and gets stuck once the boundary loss B balances the gradients from the surrogate network.

To reparameterize the problem, we create a neural network $\mathcal{N}$ that maps the 256 values of the observed signal $u(t)$ to the unknown value t_0. We chose a standard architecture inspired by image classification networks (Simonyan and Zisserman, 2014) and train it according to Eq. 2. The network consists of five convolutional layers with ReLU activation functions, batch normalization, and max-pooling layers, followed by two fully-connected layers. During the optimization, the estimate of t_0 repeatedly moves from minimum to minimum until settling after around 500 iterations. Like BFGS, most examples do not converge to the global optimum and stop at a local minimum instead. However, the cross-talk between different examples, induced by the shared parameters θ and the summation of the individual loss functions, regularizes the movement in t_0 space, preventing solutions from moving far away from the global optimum. Meanwhile, the feedback from the analytic gradients of F ensures that each example finds a locally optimal solution. Overall, this results in around 80% of examples finding a better solution than BFGS.

For supervised training of $\mathcal{N}$, we use the same training data set as for the neural adjoint method. This approach's much smoother loss landscape lets all solution estimates progress close to the ground truth. However, lacking the gradient feedback from the forward process $\mathcal{F}$, the inferred solutions are slightly off from the actual solution and, since the highest loss values are close to the global optimum, this raises the overall loss during training even though the solutions are approaching the global optima. This phenomenon gets resolved with solution refinement using BFGS.

Fig. 2d shows the results for different numbers of inverse problems and training set sizes n. Since BFGS optimizes each example independently, the data set size has no influence on its performance. Variances in the mean final loss indicate that the specific selection of inverse problems may be slightly easier or harder to solve than the average. The neural adjoint method and reparameterized optimization both perform better than BFGS with the reparameterized optimization producing lower loss values. However, both do not scale with n in this example. This feature can only be observed with supervised training whose solution quality noticeably increases with n. This is due to the corresponding increase in training set size, which allows the model to improve generalization and does not depend on the number of tested inverse problems. For $n \geq 32$, supervised training in combination with the above-mentioned solution refinement consistently outperforms all other methods.

A detailed description of the network architecture along with additional learning curves, parameter evolution plots as well as the performance on further data set sizes n can be found in Appendix B.1.

4.2 BILLIARDS

Next, we consider a rigid-body setup inspired by differentiable billiards simulations of previous work (Hu et al., 2020). The task consists of finding the optimal initial velocity $\vec{v}_0$ of a cue ball so it hits another ball, imparting momentum in a non-elastic collision to make the second ball come to rest at a fixed target location. This setup is portrayed in Fig. 3a and the corresponding loss landscape for a fixed x velocity in Fig. 3b. A collision only occurs if $\vec{v}_0$ is large enough and pointed towards the other ball. Otherwise, the second ball stays motionless, resulting in a constant loss value and $\frac{\partial \mathcal{L}}{\partial \vec{v}_0} = 0$.

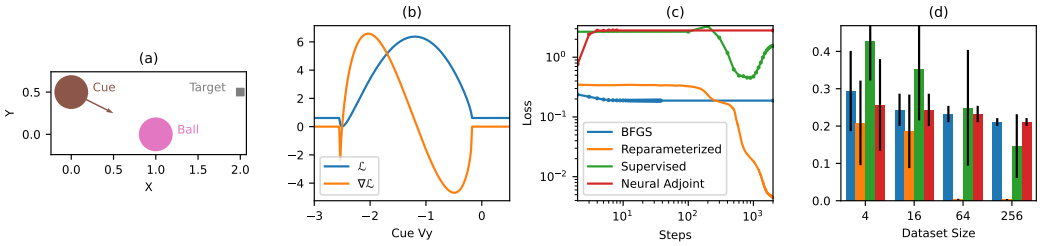


Figure 3: Billiards experiment. **(a)** Task: the cue ball must hit the other ball so that it comes to rest at the target, **(b)** corresponding loss and gradient landscape for v_y, **(c)** optimization curves without refinement, **(d)** refined loss L/n by number of examples n, mean and standard deviation over multiple network initializations and data sets.

This property prevents classical optimizers from converging if they hit such a region in the solution space. The optimization curves are shown in Fig. 3c. BFGS only converges for those examples where the cue ball already hits the correct side of the other ball.

For reparameterization, we employ a fully-connected neural network $\mathcal{N}$ with three hidden layers using Sigmoid activation functions and positional encoding. The joint optimization with $\mathcal{N}$ drastically improves the solutions. While for $n \leq 32$ only small differences to BFGS can be observed, access to more inverse problems lets gradients from some problems steer the optimization of others that get no useful feedback. This results in almost all problems converging to the solution for $n \geq 64$ (see Fig. 3d).

In this experiment, the distribution of the solutions $P(\vec{v}_0)$ is not available as hitting the target precisely requires a specific velocity $\vec{v}_0$ that is unknown a-priori. We can, however, generate training data with varying $\vec{v}_0$ and observe the final positions of the balls, then train a supervised $\mathcal{N}$ as well as a surrogate network for the neural adjoint method on this data set. However, this is less efficient as most of the examples in the data set do not result in an optimal collision.

The neural adjoint method fails to approach the true solutions and instead gets stuck on the training data boundary in solution space. Likewise, the supervised model cannot accurately extrapolate the true solution distribution from the sub-par training set.

4.3 KURAMOTO–SIVASHINSKY EQUATION

The Kuramoto–Sivashinsky (KS) equation, originally developed to model the unstable behavior of flame fronts (Kuramoto, 1978), models a chaotic one-dimensional system, $\dot{u}(t) = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} - u \cdot \nabla u$. We consider a two-parameter inverse problem involving the forced KS equation with altered advection strength,

$$\dot{u}(t) = \alpha \cdot G(x) - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} - \beta \cdot u \cdot \nabla u,$$

where $G(x)$ is a fixed time-independent forcing term and $\alpha, \beta \in \mathbb{R}$ denote the unknown parameters governing the evolution. Each inverse problem starts from a randomly generated initial state $u(t = 0)$ and is simulated until $t = 25$, by which point the system becomes chaotic but is still smooth enough to allow for gradient-based optimization. We constrain $\alpha \in [-1, 1]$, $\beta \in [\frac{1}{2}, \frac{3}{2}]$ to keep the system numerically stable. Fig. 4a shows example trajectories of this setup and the corresponding gradient landscape of $\frac{\partial \mathcal{L}}{\partial \beta}\|_{\alpha=\alpha^*}$ for the true value of α is shown in Fig. 4b.

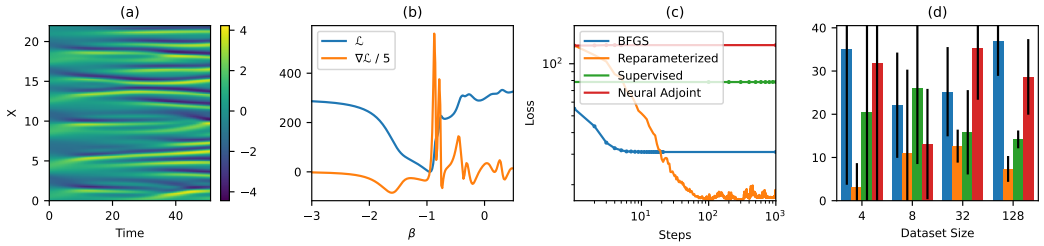


Figure 4: Kuramoto–Sivashinsky experiment. **(a)** Example trajectory, **(b)** corresponding loss and gradient landscape for β, **(c)** optimization curves without refinement, **(d)** refined loss L/n by number of examples n, mean and standard deviation over multiple network initializations and data sets.

Fig. 4c shows the optimization curves for finding α, β. Despite the complex nature of the loss landscape, BFGS manages to find the correct solution in about 60% of cases. The reparameterized optimization, based on a similar network architecture as for the wavepacket experiment but utilizing 2D convolutions, finds the correct solutions in over 80% of cases but, without refinement, the accuracy stagnates far from machine precision. Refining these solutions with BFGS, as described above, sees the accuracy of these cases decrease to machine precision in 4 to 17 iterations, less than the 12 to 22 that BFGS requires when initialized from the distribution mean $\mathbb{E}[P(\xi)]$.

Supervised training with refinement produces better solutions in 58% of examples, averaged over the shown n. The unrefined solutions benefit from larger n on this example because of the large number of possible observed outputs that the KS equation can produce for varying α, β. At $n = 2$, all unrefined solutions are worse than BFGS while for $n \geq 64$ around 20% of problems find better solutions. With refinement, these number jump to 50% and 62%.

This property also makes it hard for a surrogate network, required by the neural adjoint method, to accurately approximate the KS equation, causing the following adjoint optimization to yield inaccurate results that fail to match BFGS even after refinement.

4.4 INCOMPRESSIBLE NAVIER-STOKES

Incompressible Newtonian fluids are described by the Navier-Stokes equations,

$$\dot{u}(\vec{x}, t) = \nu \nabla^2 u - u \cdot \nabla u - \nabla p \quad \text{s.t.} \quad \nabla^2 p = \nabla \cdot v$$

with $\nu \geq 0$. As they can result in highly complex dynamics (Batchelor and Batchelor, 1967), they represent a particularly challenging test case, which is relevant for a variety of real-world

problems (Pope, 2000). We consider a setup similar to particle imaging velocimetry (Grant, 1997) in which the velocity in the upper half of a two-dimensional domain with obstacles can be observed. The velocity is randomly initialized in the whole domain and a localized force is applied near the bottom of the domain at $t = 0$. The task is to reconstruct the position x_0 and initial velocity $\vec{v}_0$ of this force region by observing the initial and final velocity field only in the top half of the domain. The initial velocity in the bottom half is unknown and cannot be recovered, making a perfect fit impossible. Fig. 5a,b show an example initial and final state of the system. The final velocity field is measured at $t = 56$ by which time fast eddies have dissipated significantly.

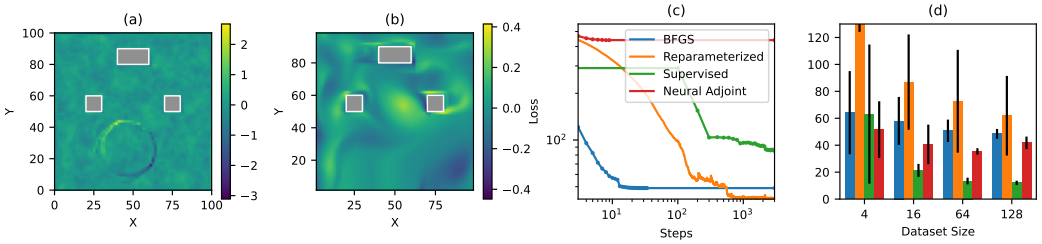


Figure 5: Fluid experiment. **(a,b)** Example initial and final velocity fields, obstacles in gray. Only the upper half, $y \geq 50$, is observed. **(c)** Optimization curves without refinement, **(d)** refined loss L/n by the number of examples n, mean and standard deviation over multiple network initializations and data sets.

Fig. 5c shows the optimization curves. On this problem, BFGS converges to some optimum in all cases, usually within 10 iterations, sometimes requiring up to 40 iterations. However, many examples get stuck in local optima.

For joint optimization, we reparameterize the solution space using a network architecture similar to the previous experiment, featuring four 2D convolutional layers and two fully-connected layers. For all tested n, the reparameterized optimization produces larger mean loss values than BFGS, especially for small n. This results from about 10% of examples seeing higher than average loss values. Nonetheless, 66.7% of the inverse problems are solved more accurately than BFGS on average for $n > 4$.

The neural adjoint method nearly always converges to solutions within the training set parameter space, not relying on the boundary loss. With solution refinement, this results in a mean loss that seems largely independent of n and is slightly lower than the results from direct BFGS optimization. However, most of this improvement comes from the secondary refinement stage which runs BFGS on the true F. Without solutions refinement, the neural adjoint method yields inaccurate results, losing to BFGS in 98.2% of cases.

Supervised training does not suffer from examples getting stuck in a local minimum early on. The highest-loss solutions, which contribute the most to L, are about an order of magnitude better than the worst BFGS solutions, leading to a much smaller total loss for $n \geq 16$. With solution refinement, 64%, 73% and 72% of examples yield a better solution than BFGS for $n = 16, 64, 128$, respectively.

5 Discussion

In our experiments, we have focused on relatively small data sets of between 2 and 256 examples to quantify the worst-case for machine learning methods and observe trends. Using off-the-shelf neural network architectures and optimizers with no tuning to the specific problem, joint optimization finds better solutions than BFGS in an average of 69% of tested problems. However, to achieve the best accuracy, the solution estimates must be passed to a classical optimizer for refinement as training the network to this level of accuracy would take an inordinate amount of time and large data sets. Tuning the architectures to the specific examples could lead to further improvements in performance but would make the approach domain-dependent.

When training data including ground truth solutions are available or can be generated, supervised learning can sidestep many difficulties that complex loss landscapes pose, such as local minima,

Table 2: Fraction of inverse problems for which neural-network-based methods with refinement find better or equal solutions than BFGS. Mean over multiple seeds and all n shown in subfigures (d).

Experiment	Reparameterized		Supervised		Neural Adjoint	
	Better	Equal	Better	Equal	Better	Equal
Wave packet fit	**86.0%**	1.8%	65.1%	14.4%	40.2%	47.4%
Billiards	**61.7%**	9.0%	27.0%	27.2%	1.6%	98.4%
Kuramoto–Sivashinsky	**62.3%**	0.0%	57.7%	0.0%	23.9%	62.2%
Incompr. Navier-Stokes	64.1%	0.0%	**66.2%**	0.1%	56.9%	0.1%

alternating gradient directions, or zero-gradient areas. This makes supervised learning another promising alternative to direct optimization, albeit a more involved one.

The neural adjoint method, on the other hand, yields only very minor improvements over BFGS optimization in our experiments, despite the surrogate network successfully learning to reproduce the training data. This is not surprising as the neural adjoint method tries to approximate the original loss landscape which is often difficult to optimize. Improvements over BFGS must therefore come from regularization effects and exposure to a larger part of the solution space. The fact that the neural adjoint method with solution refinement produces similar results almost independent of the number of data points n shows that the joint optimization has little benefit here. Instead, the refinement stage, which treats all examples independently, dominates the final solution quality. Note that the neural adjoint method is purely data-driven and does not require an explicit form for the forward process F, making it more widely applicable than the setting considered here.

Tab. 2 summarizes the improvements over classical optimizations for all methods. A corresponding table without solution refinement can be found in Appendix B. Considering that reparameterized optimization is the only network-based method that does not require domain-specific information and nevertheless shows the biggest improvement overall, we believe it is the most attractive variant among the three learned versions. Inverse problems for which reparameterized training does not find good solutions are easy to identify by their outlier loss values. In these cases, one could simply compare the solution to a reference solution obtained via direct optimization, and choose the best result.

Limitations We have only considered unconstrained optimization problems in this work, enforcing hard constraints by running bounded parameters through a scaled $\tanh$ function which naturally clamps out-of-bounds values in a differentiable manner.

The improved solutions found by joint optimization come with an increased computational cost compared to direct optimization. The time it took to train the reparameterization networks was 3x to 6x longer for the first three experiments and 22x for the fluids experiment (see appendix B).

6 Conclusions and outlook

We have investigated the effects of joint optimization of multiple inverse problems by reparameterizing the solution space using a neural network, showing that joint optimization can often find better solutions than classical optimization techniques. Since our reparameterization approach does not require any more information than classical optimizers, it can be used as a drop-in replacement. This could be achieved by adding a function or option to existing optimization libraries that internally sets up a standard neural network with the required number of inputs and outputs and runs the optimization, hiding details of the training process, network architecture, and hyperparameters from the user while making the gains in optimization accuracy conveniently accessible. To facilitate this, we will make the full source code publicly available.

From accelerating matrix multiplications (Fawzi et al., 2022) to solving systems of linear equations (Calì et al., 2023; Sappl et al., 2019), it is becoming increasingly clear that machine learning methods can be applied to purely numerical problems outside of typical big data settings, and our results show that this also extends to solving nonlinear inverse problems.

REFERENCES

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.

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.

Kelsey R Allen, Tatiana Lopez-Guevara, Kimberly Stachenfeld, Alvaro Sanchez-Gonzalez, Peter Battaglia, Jessica Hamrick, and Tobias Pfaff. Physical design using differentiable learned simulators. *arXiv preprint arXiv:2202.00728*, 2022.

Randall Balestriero, Jerome Pesenti, and Yann LeCun. Learning in high dimension always amounts to extrapolation. *arXiv preprint arXiv:2110.09485*, 2021.

Cx K Batchelor and George Keith Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 1967.

Mikhail Belkin, Daniel J Hsu, and Partha Mitra. Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate. *Advances in neural information processing systems*, 31, 2018.

Salvatore Calì, Daniel C Hackett, Yin Lin, Phiala E Shanahan, and Brian Xiao. Neural-network preconditioners for solving the dirac equation in lattice gauge theory. *Physical Review D*, 107(3): 034508, 2023.

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.

Hanjun Dai, Yuan Xue, Zia Syed, Dale Schuurmans, and Bo Dai. Neural stochastic dual dynamic programming. *arXiv preprint arXiv:2112.00874*, 2021.

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.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

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.

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.

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

Ian Grant. Particle image velocimetry: a review. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 211(1):55–76, 1997.

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

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.

Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

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

Philipp Holl, Vladlen Koltun, and Nils Thuerey. Scale-invariant learning by physics inversion. *arXiv preprint arXiv:2109.15048*, 2021.

Stephan Hoyer, Jascha Sohl-Dickstein, and Sam Greydanus. Neural reparameterization improves structural optimization. *arXiv preprint arXiv:1909.04240*, 2019.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *International Conference on Learning Representations (ICLR)*, 2020.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep fluids: A generative network for parameterized fluid simulations. *Computer Graphics Forum*, 2019. ISSN 1467-8659. doi: 10.1111/cgf.13619.

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

Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. *arXiv:2102.01010 [physics]*, 2021. URL http://arxiv.org/abs/2102.01010.

Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, and Michael W Mahoney. Characterizing possible failure modes in physics-informed neural networks. *Advances in Neural Information Processing Systems*, 34:26548–26560, 2021.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Yoshiki Kuramoto. Diffusion-induced chaos in reaction systems. *Progress of Theoretical Physics Supplement*, 64:346–367, 1978.

Housen Li, Johannes Schwab, Stephan Antholzer, and Markus Haltmeier. Nett: Solving inverse problems with deep neural networks. *Inverse Problems*, 36(6):065005, 2020.

Junbang Liang, Ming Lin, and Vladlen Koltun. Differentiable cloth simulation for inverse problems. In *Advances in Neural Information Processing Systems*, pages 771–780, 2019.

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

Lu Lu, Raphael Pestourie, Wenjie Yao, Zhicheng Wang, Francesc Verdugo, and Steven G Johnson. Physics-informed neural networks with hard constraints for inverse design. *SIAM Journal on Scientific Computing*, 43(6):B1105–B1132, 2021.

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.

Kohta Murase, Markus Ahlers, and Brian C Lacki. Testing the hadronuclear origin of pev neutrinos observed with icecube. *Physical Review D*, 88(12):121301, 2013.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Stephen Pope. *Turbulent Flows*. Cambridge University Press, 2000. ISBN 978-0-511-84053-1. doi: 10.1017/CBO9780511840531.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3 edition, 2007. ISBN 9780521880688.

Patrick Putzky and Max Welling. Recurrent inference machines for solving inverse problems. *arXiv preprint arXiv:1706.04008*, 2017.

Maziar Raissi, Paris Perdikaris, and George Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

Stephan Rasp and Nils Thuerey. Data-driven medium-range weather prediction with a resnet pretrained on climate simulations: A new model for weatherbench. *Journal of Advances in Modeling Earth Systems*, 13(2):e2020MS002405, 2021.

Simiao Ren, Willie Padilla, and Jordan Malof. Benchmarking deep inverse models over time, and the neural-adjoint method. *Advances in Neural Information Processing Systems*, 33:38–48, 2020.

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.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International conference on machine learning*, pages 8459–8468. PMLR, 2020.

Johannes Sappl, Laurent Seiler, Matthias Harders, and Wolfgang Rauch. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems. *arXiv preprint arXiv:1906.06925*, 2019.

Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, pages 317–335, 2018.

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

Samuel S Schoenholz and Ekin D Cubuk. Jax, md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv:1912.04232*, 2019.

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. ISSN 0885-7474, 1573-7691. doi: 10.1007/s10915-007-9166-4.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Kimberly Stachenfeld, Drummond B Fielding, Dmitrii Kochkov, Miles Cranmer, Tobias Pfaff, Jonathan Godwin, Can Cui, Shirley Ho, Peter Battaglia, and Alvaro Sanchez-Gonzalez. Learned coarse models for efficient turbulence simulation. *arXiv preprint arXiv:2112.15275*, 2021.

Tetsuya Takahashi, Junbang Liang, Yi-Ling Qiao, and Ming C Lin. Differentiable fluids with solid coupling for learning and control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35(7), pages 6138–6146, 2021.

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

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of Machine Learning Research*, pages 3424–3433, 2017.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9446–9454, 2018.

Kiwon Um, Robert Brand, Yun Raymond Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial intelligence review*, 18:77–95, 2002.

Pauli Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

XIA Yang, Suhaib Zafar, J-X Wang, and Heng Xiao. Predictive large-eddy-simulation wall modeling via physics-informed neural networks. *Physical Review Fluids*, 4(3):034602, 2019.

Ya-xiang Yuan. A review of trust region algorithms for optimization. In *Iciam*, volume 99(1), pages 271–282, 2000.

A BASELINES AND NETWORK ARCHITECTURES

This section details the neural network architectures we employ in our experiments, as well as implementation details of the baselines to which we compare the reparameterized optimization.

Network architectures. We deliberately use generic off-the-shelf neural network architectures. In all performed experiments, the solution space consists of a finite number of scalars while the initial and final simulation states often involve spatial data, i.e. grids. For grid data, we use convolutional layers on multiple resolution levels while fully-connected layers process non-grid data. Applying this approach to our problems results in the following scenarios:

- **Grid to scalars (G2S).** We use a standard architecture for classification that largely follows by VGG (Simonyan and Zisserman, 2014). It consists of multiple convolutional blocks followed by fully-connected layers. Each convolutional block consists of one or multiple convolutional layers with kernel size 3^d where d denotes the number of grid dimensions. A batch normalization, activation, and max pooling layer follows each convolutional layer, reducing the resolution by half at the end of each block. The result is passed to a multilayer perceptron (MLP) which alternates linear, activation, and batch normalization layers before the result is outputted by a final linear layer.
- **Scalars to scalars (S2S).** When no grid data is involved, we simply use MLPs (Goodfellow et al., 2016) to map inputs to outputs, optionally with positional encoding at the inputs. The MLP consists of multiple linear layers and activation layers but we do not use batch normalization layers since our networks are relatively shallow with no more than three hidden layers.
- **Grid to grid (G2G).** This case is only needed for the surrogate network required by the neural adjoint method. Here, we use the popular U-Net (Ronneberger et al., 2015) architecture with residual connections. Multiple convolutional blocks progressively decrease the spatial resolution, followed by upsampling convolutional blocks. The downsampling blocks match the ones described above. The upsampling blocks linearly interpolate the result to double the resolution before concatenating the corresponding processed input of the same resolution for the residual connections.

The specific hyperparameter values used for these generic architectures are given in the corresponding experimental details sections.

BFGS. We use the BFGS Implementation from SciPy (Virtanen et al., 2020) which runs the optimizer-internal computations on the CPU. All loss and gradient evaluations are bundled and dispatched to the GPU to be processed in parallel.

Neural adjoint. The neural adjoint method (Ren et al., 2020) employs a neural network $\tilde{\mathcal{N}}$ to act as a surrogate for F. $\tilde{\mathcal{N}}$ is then used in place of F in an iterative optimization. Since $\tilde{\mathcal{N}}$ cannot be expected to produce accurate results outside of the region covered by its training data, a boundary loss term is added to the optimization to prevent the optimizer from leaving that region (Ren et al., 2020). We formulate this boundary loss in a differentiable manner to make it compatible with higher-order optimizers. First, we determine the minimum $\xi^j_{\min}$ and the maximum $\xi^j_{\max}$ value in the training set for each parameter j. Then, we formulate the boundary loss as

$$B(\xi) = \sum_j \mathrm{SoftPlus}_\gamma \left(\frac{\max(\xi^j - \xi^j_{\max}, \xi^j_{\min} - \xi^j)}{\xi^j_{\max} - \xi^j_{\min}} \right),$$

where $\mathrm{SoftPlus}_\gamma(x) \equiv \frac{1}{\gamma} \log(1 + e^{\gamma x})$.

B EXPERIMENTAL DETAILS

This section lists additional details about the experiments showcased in the main paper. An overview of the symbols introduced with the experiments is given in Tab. 3.

Table 3: Physical quantities corresponding to the abstract symbols used in Eqs. 1 and 2 for each experiment.

Experiment	ξ	x	y
Wave packet fit	t_0	$\epsilon(t)$	$u(t)$
Billiards	$\vec{v}_0$	Initial ball positions	Final ball positions
Kuramoto–Sivashinsky	α, β	$u(x)\|_{t=0}$	$u(x)\|_{t=25}$
Incompr. Navier-Stokes	$x_0, \vec{v}_0$	$u(x,y)\|_{t=0}$	$u(x,y)\|_{t=56}$

Software and hardware. We used PyTorch (Paszke et al., 2019) and Φ_{Flow} (Holl et al., 2020) to run our experiments. The full source code is part of the supplemental material. The first three experiments were run on a GeForce RTX 3090. Due to memory requirements, the fluid experiment was run partly on a Quadro RTX 8000 which allowed 128 simulations to be held in GPU memory.

The corresponding wall-clock run times are shown in Tab. 4. There, *parallel BFGS* denotes a BFGS implementation that runs the forward process and backpropagation on the GPU. This is much more efficient than looping over the individual examples, the way most classical BFGS optimizations are implemented. For the KS experiment, sequential BFGS solves take 8x longer than the batched solve for $n = 16$ and 76x longer for $n = 256$. Running this on the CPU increases the runtime by an additional 50-60%. Compared to the sequential CPU approach at $n = 256$, our method is 18x faster than BFGS.

Hyperparameter selection. For each network, we select one of the three generic architectures listed above. Our main objective in choosing the values of the hyperparameters, such as the number of layers and layer width, is keeping the total number of parameters large enough to fit the problem easily but low enough to train the network quickly. The only hyperparameter which we tune is the Adam (Kingma and Ba, 2015) learning rate η. We start with $\eta = 0.01$ and progressively reduce it by a factor of 10 until the loss decreases during the optimization. The exact hyperparameter values are given in the corresponding experiment section.

Refinement. We apply BFGS refinement as a second stage to reparameterized optimization, supervised training, and the neural adjoint method. In all cases, we run a standard BFGS optimization on the actual $\mathcal{L}$, so the gradients are backpropagated through F. For reparameterized optimization and the neural adjoint method, we use the parameter estimate with the lowest recorded loss value. As supervised training makes use of pre-trained network, we use the final parameter estimate ξ_i as an initial guess. The full evolution of the parameter estimates over the course of training is shown for all experiments below.

Tab. 5 lists the fraction of the total loss improvement performed by the network fit. In all experiments, the network fit stage is responsible for the bulk of the improvement and the refinement stage improves the loss much less overall. We also observe that, for larger data set sizes, the network fit contributes even more to the overall improvement while for small data set sizes, the refinement stage is more important.

Table 4: Training and optimization times for the largest tested data set size n in seconds.

Experiment	Parallel BFGS	Network fit	Refinement	Supervised fit	Sup. Refinement	Surrogate fit	Neural Adjoint	N.A. Refinement
Wave packet fit	15.1 ± 1.0	46.9 ± 0.2	15.0 ± 0.2	24.0 ± 0.2	13.8 ± 0.7	46.5 ± 0.2	13.1 ± 0.5	13.8 ± 1.7
Billiards	21.5 ± 1.0	115.2 ± 0.6	25.3 ± 0.2	8.8 ± 0.1	20.9 ± 1.7	12.9 ± 0.1	16.5 ± 2.5	21.8 ± 1.1
Kuramoto–Sivashinsky	152.8 ± 11.8	638.8 ± 3.7	109.3 ± 7.2	11.4 ± 0.9	122 ± 64	16.4 ± 0.8	13.7 ± 2.3	147 ± 12
Incompr. Navier-Stokes	1858 ± 95	29510 ± 637	1270 ± 205	212 ± 7	1390 ± 333	195.6 ± 0.1	8.7 ± 1.2	1451 ± 63

Table 5: Fraction of the total loss decrease achieved by the network fit. The remaining improvement is made by the refinement stage using BFGS. The given fractions are computed per example and then averaged.

Experiment	$n = 4$	$n = 8$	$n = 32$	$n = 128$
Wave packet fit	$78.5\% \pm 17.8\%$	$89.1\% \pm 8.8\%$	$92.4\% \pm 3.7\%$	$91.7\% \pm 4.5\%$
Billiards	$88.9\% \pm 13.0\%$	$86.8\% \pm 14.0\%$	$92.9\% \pm 11.2\%$	$98.1\% \pm 2.0\%$
Kuramoto–Sivashinsky	$93.4\% \pm 9.8\%$	$96.2\% \pm 5.9\%$	$96.0\% \pm 2.5\%$	$95.9\% \pm 1.1\%$
Incompr. Navier-Stokes	$100.0\% \pm 0.0\%$	$99.4\% \pm 0.5\%$	$96.6\% \pm 3.4\%$	$96.8\% \pm 2.5\%$

Results. Complementary to Tab. 2 of the main paper, Tab. 6 gives an overview of how many examples were improved by the various neural-network-based approaches *without* refinement. Learning curves, loss and improvement statistics, as well as example parameter trajectories are shown in the following subsections.

Table 6: Fraction of inverse problems for which neural-network-based methods *without* refinement find better or equal solutions than BFGS. Mean over multiple seeds and all n shown in subfigures (d) of the main paper.

Experiment	Reparameterized		Supervised		Neural Adjoint	
	Better	Equal	Better	Equal	Better	Equal
Wave packet fit	**80.5%**	0.9%	55.9%	2.0%	28.2%	0.8%
Billiards	**44.3%**	9.0%	14.6%	19.9%	1.6%	29.3%
Kuramoto–Sivashinsky	**42.8%**	0.0%	14.4%	0.0%	6.4%	0.0%
Incompr. Navier-Stokes	**62.5%**	0.0%	23.5%	0.0%	1.1%	0.0%

B.1 WAVE PACKET LOCALIZATION

For the wave packet experiment, we first determine the true position t_0 of the wave packet by sampling random values from a uniform distribution between $t_0 \in [26, 230]$. Noise $\epsilon(t)$ is sampled from a normal distribution with standard deviation $\sigma = 0.1$ for every $t = 1, ..., 256$ and superimposed on the signal, as described in the main text. The noise pattern is only used to generate the reference data and is not available to the optimizers. We run this experiment five times with varying initialization seeds for both networks and data sets.

Networks. The surrogate network, required by the neural adjoint method, takes t_0 and $\epsilon(t)$ as input and outputs an approximation of $u(t)$. Since $\epsilon(t)$ and $u(t)$ are one-dimensional grids, we employ the G2G architecture with two input feature maps of size 256 and one output feature map, totaling 13,073 parameters. The reparameterization network maps the grid $u(t)$ to the estimated scalar t_0. Consequently, we use the G2S architecture described in section A. We use five blocks with one convolutional layer with 16 feature maps each, reducing the resolution from 256 to 8. The MLP part consists of two hidden fully-connected layers with sizes 64 and 32. In total, this network contains 13,925 parameters. All networks are trained using Adam with a learning rate of $\eta = 0.001$. Fig. 9 shows the absolute the change in the reparameterization network weights that results from training.

Additional results. Fig. 6 shows the resulting loss and improvement over BFGS, both before and after refinement. The learning curves for four data set sizes n are shown in Fig. 7, and the parameter evolution of four examples during optimization are shown in Fig. 8.

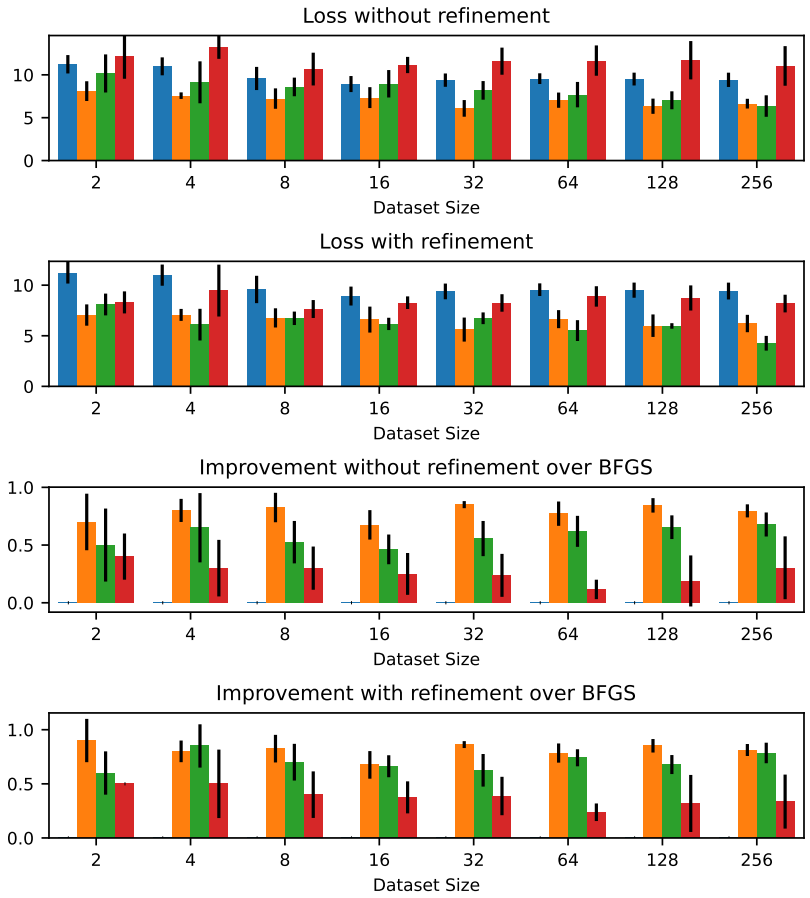


Figure 6: Loss and improvement over BFGS before and after refinement for the wave packet experiment. Colors match figures from the main paper (blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint).

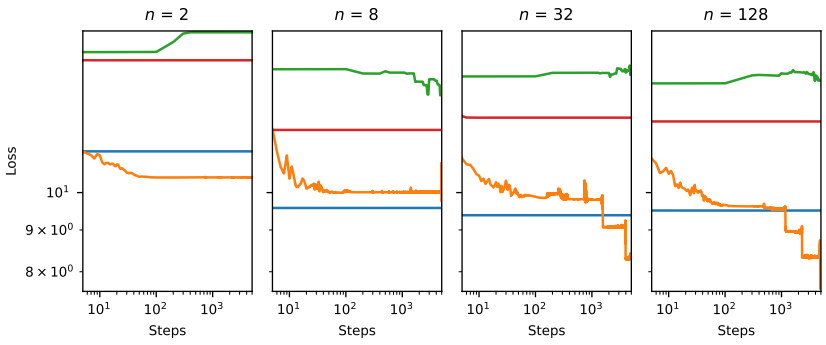


Figure 7: Optimization curves for different data set sizes of the wave packet experiment before refinement. Curves show the mean over 5 network and data set initialization seeds. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint.

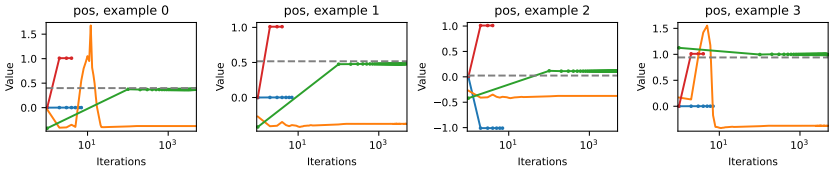


Figure 8: Example parameter evolution during optimization of the wave packet experiment with $n = 128$. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. The dashed gray lines indicate the reference solution from which the example was generated. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

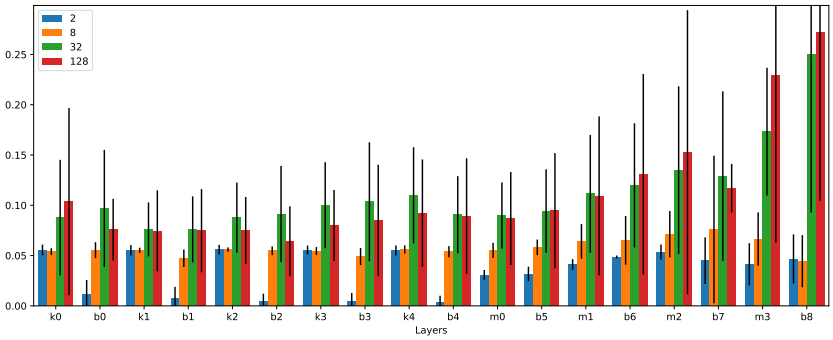


Figure 9: Reparameterization network change in the wave packet experiment for different data set sizes n, measured as the mean absolute difference in weight values before and after fitting the network. Error bars represent the standard deviation across multiple network initializations. The change is given per layer where k denotes convolution kernels, m the matrices of fully-connected layers and b biases.

B.2 BILLIARDS

For the billiards experiment, we set up a rigid body simulation of spherical balls with radius $r = 0.2$ moving in the x-y-plane. In each step, the simulator analytically integrates the evolution until the time of the subsequent collision, allowing us to simulate the dynamics at little computational cost. Collisions use a fixed elasticity of 0.8 and preserve momentum. Friction is assumed to be proportional to the speed of the balls. The simulation stops once no more collisions take place and integrates up to $t = \infty$ to let all balls come to rest. we sample initial states by randomly placing the second ball between $(1, 0)$ and $(1, 1)$ while keeping the target fixed at $(2, 0.5)$. The cue ball, located at $x = 0$ is given a starting initial velocity of $\vec{v}_0^{\text{start}} = (1, 0)$ so it will collide with the second ball in many cases by default. Starting the optimization with $\vec{v}_0^{\text{start}} = 0$ would yield $\nabla L_i = 0 \, \forall i$ and prevent any optimization using F. However, in none of these examples does the ball exactly reach the target. As the distribution of actual solutions $\vec{v}_0$ is unknown, the generated training sets for supervised and surrogate network training must rely on this broader data set, making learning more difficult.

Networks. The surrogate neural network is given a value for the initial velocity $\vec{v}_0$ and the balls' positions as input, and it outputs the predicted final position of the ball. We use positional encoding for the input using sine, cosine functions with four equidistant frequencies. The surrogate network follows the S2S architecture from section A. It is an MLP with three hidden layers containing 128 neurons, each, and comprises 37,506 parameters in total. The reparameterization network predicts $\vec{v}_0$ based on the initial and final ball positions, and we use the same network architecture as for the surrogate network. All networks are trained using Adam. For the reparameterized optimization, we use a learning rate of $\eta = 10^{-4}$ while all other methods use $\eta = 0.001$. Fig. 13 shows the absolute the change in the reparameterization network weights that results from training.

Additional results. Fig. 10 shows the resulting loss and improvement over BFGS, both before and after refinement. The learning curves for four data set sizes n are shown in Fig. 11, and the parameter evolution of four examples during optimization are shown in Fig. 12.

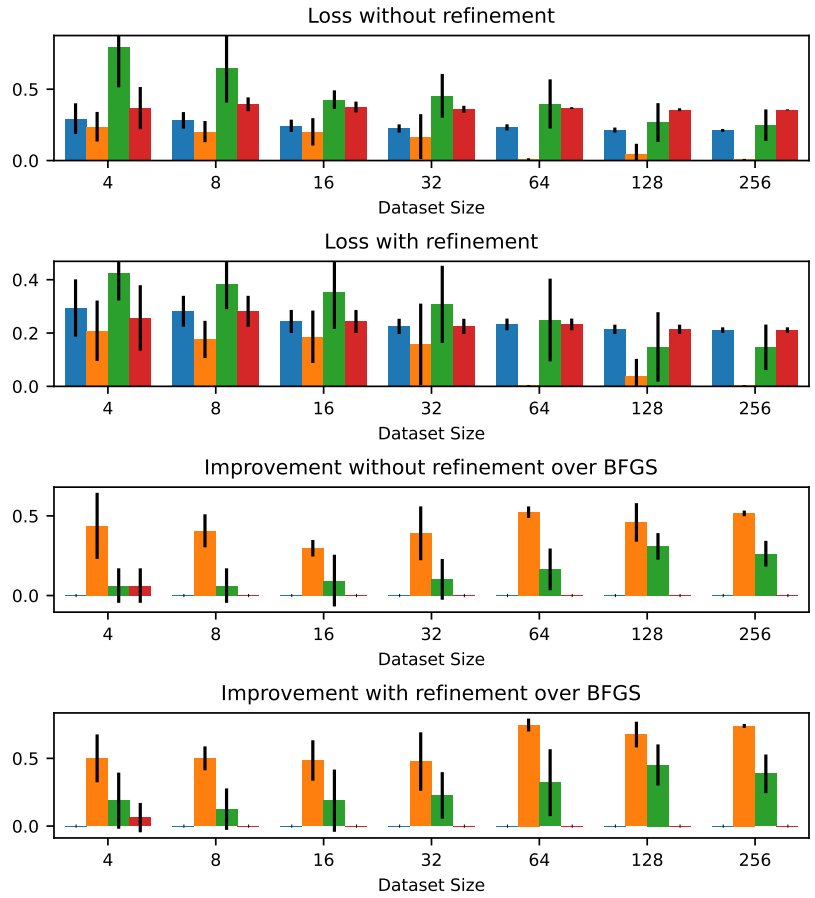


Figure 10: Loss and improvement over BFGS before and after refinement for the billiards experiment. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint.

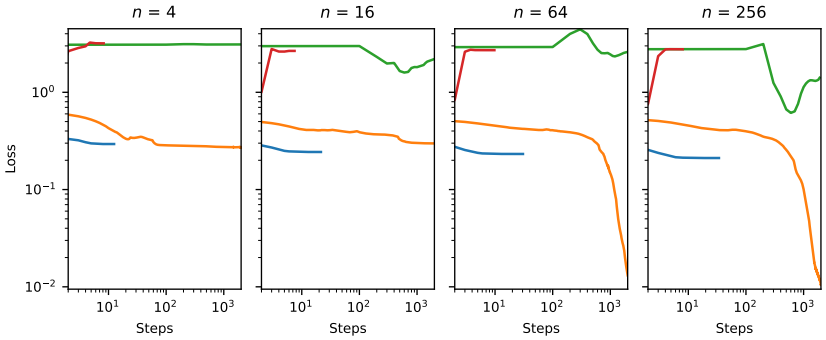


Figure 11: Optimization curves for different data set sizes of the billiards experiment before refinement. Envelopes show the standard deviation over 4 network and data set initialization seeds. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

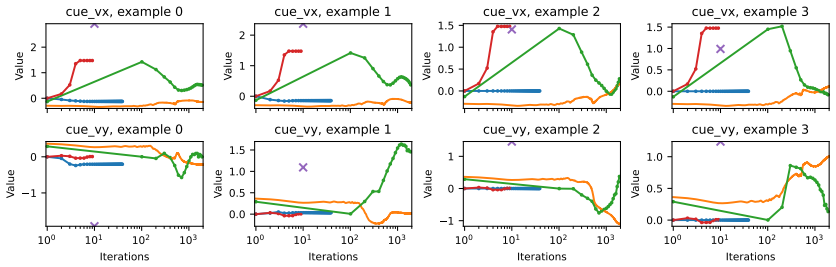


Figure 12: Example parameter evolution during optimization of the billiards experiment with $n = 128$. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. The purple crosses indicate the reference from which the example was generated and is not a valid solution in this experiment. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

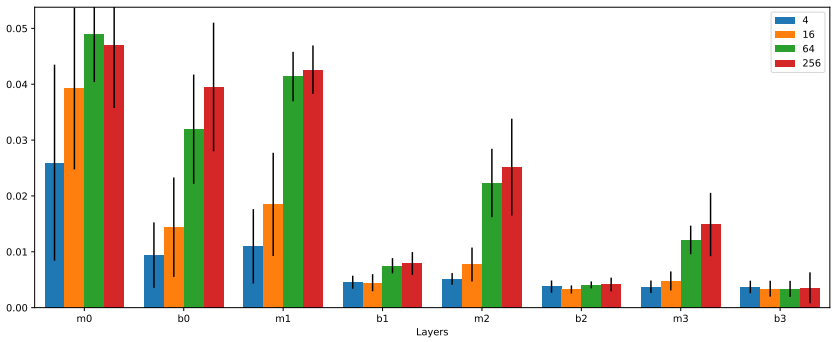


Figure 13: Reparameterization network change in the Billiards experiment for different data set sizes n, measured as the mean absolute difference in weight values before and after fitting the network. Error bars represent the standard deviation across multiple network initializations. The change is given per layer where k denotes convolution kernels, m the matrices of fully-connected layers and b biases.

B.3 KURAMOTO–SIVASHINSKY EQUATION

For this experiment, we set up a differentiable simulation of the Kuramoto–Sivashinsky (KS) equation in one dimension with a resolution of 128. We simulate the linear terms of KS equation in frequency space and use a Runge-Kutta-2 (Press et al., 2007) scheme for the non-linear term. The initial state is sampled from random noise in frequency space with smoothing applied to suppress high frequencies. In each simulation step, we add a forcing of the form $G(x) = 0.1\cos(x) - 0.01\cos(x/16) \cdot (1 - 2\sin(x/16)$ which is controlled by the parameter α as described in the main text.

Networks. The surrogate network maps the initial state $u(x, t = 0)$ and parameters α, β to the final state $u(x, t = 25)$. Since u is sampled on a grid, we use the G2G architecture from section A with three input and one output feature map, operating on four resolution levels. The reparameterization network maps $u(x, t = 0)$ and $u(x, t = 25)$ to α, β and we employ the G2S architecture with four convolutional layers of widths 32, 32, 64, 64, followed by two hidden fully-connected layers with 64 neurons each. We train both networks using Adam with a learning rate of $\eta = 0.001$ for 1000 iterations. Fig. 17 shows the absolute the change in the reparameterization network weights that results from training.

Additional results. Fig. 14 shows the resulting loss and improvement over BFGS, both before and after refinement. The learning curves for four data set sizes n are shown in Fig. 15, and the parameter evolution of four examples during optimization are shown in Fig. 16.

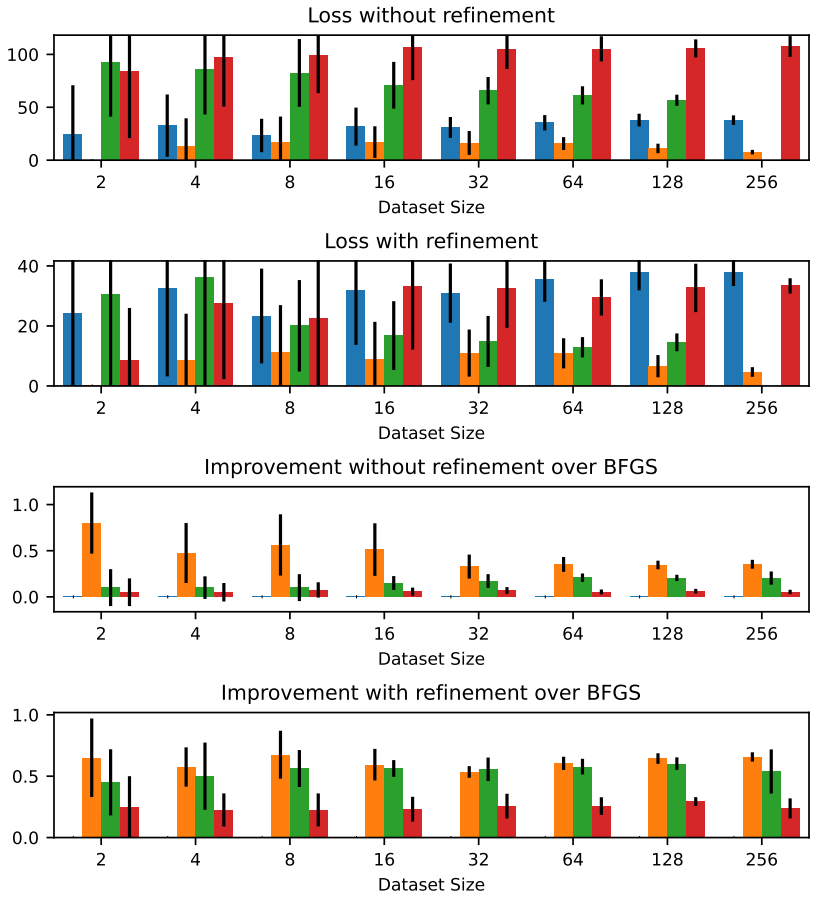


Figure 14: Loss and improvement over BFGS before and after refinement for the Kuramoto–Sivashinsky experiment. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint.

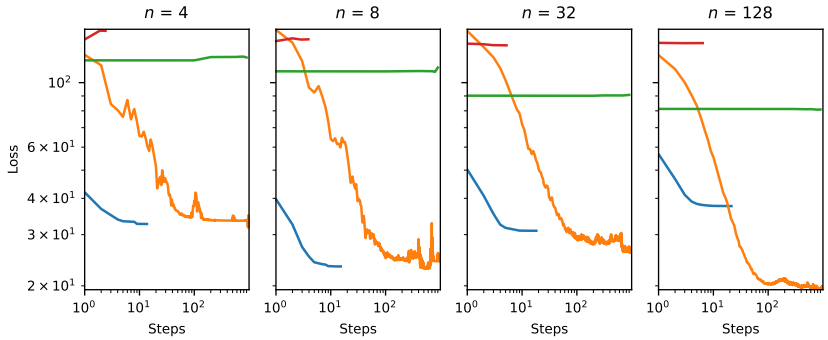


Figure 15: Optimization curves for different data set sizes of the Kuramoto–Sivashinsky experiment before refinement. Envelopes show the standard deviation over 10 network and data set initialization seeds. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

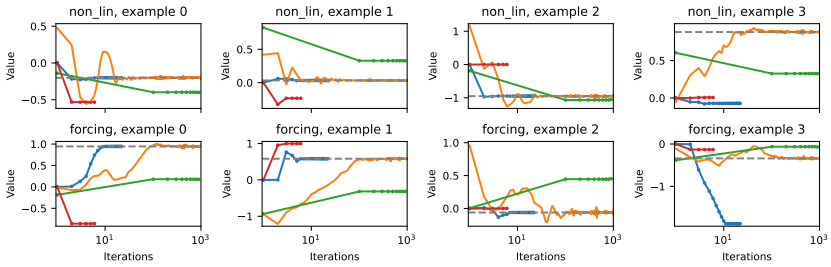


Figure 16: Example parameter evolution during optimization of the Kuramoto–Sivashinsky experiment with $n = 128$. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. The dashed gray lines indicate the reference solution from which the example was generated. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

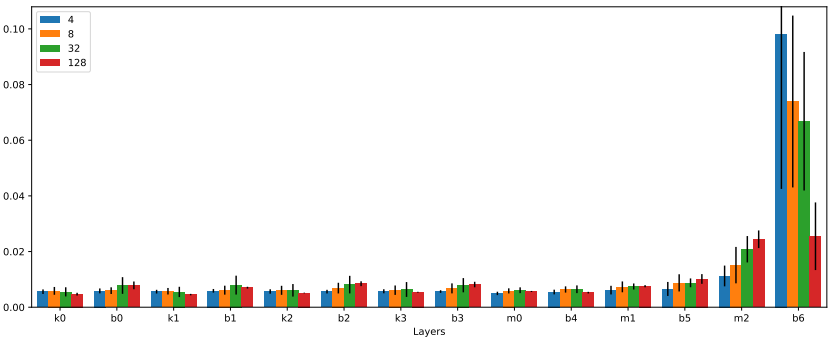


Figure 17: Reparameterization network change in the Kuramoto–Sivashinsky experiment for different data set sizes n, measured as the mean absolute difference in weight values before and after fitting the network. Error bars represent the standard deviation across multiple network initializations. The change is given per layer where k denotes convolution kernels, m the matrices of fully-connected layers and b biases.

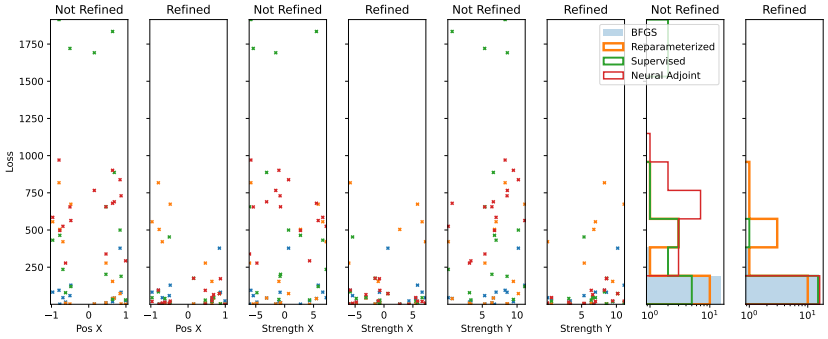


Figure 18: Distribution of loss values for $n = 4$ in the Navier-Stokes experiment, with and without refinement. The first six plots show the loss distribution along the ground truth value of one of the parameters to be optimized. The right plots show the margin distribution of loss values. The results of 4 network and data set initialization seeds are accumulated.

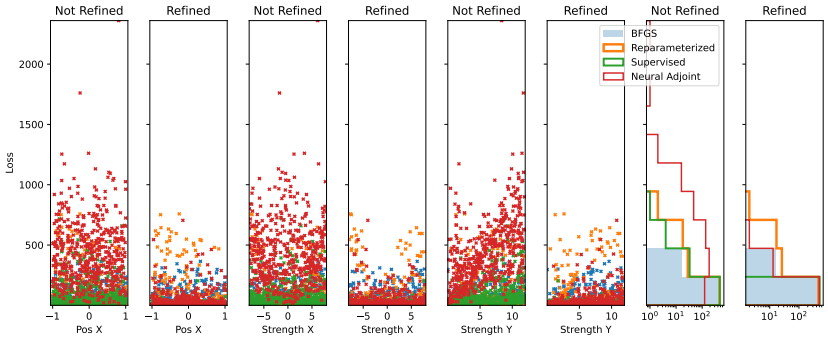


Figure 19: Distribution of loss values for $n = 128$ in the Navier-Stokes experiment, with and without refinement. The first six plots show the loss distribution along the ground truth value of one of the parameters to be optimized. The right plots show the margin distribution of loss values. The results of 4 network and data set initialization seeds are accumulated.

B.4 Incompressible Navier-Stokes

We simulate an incompressible two-dimensional fluid in a 100 by 100 box with a resolution of 64 by 64, employing a direct numerical solver for incompressible fluids from Φ_{Flow} (Holl et al., 2020). Specifically, we use the marker-in-cell (MAC) method (Harlow and Welch, 1965; Harlow, 1972) 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 (Selle et al., 2008) to transport the marker and the velocity vectors. We do not simulate explicit diffusion as the numerical diffusion introduced by the advection scheme on this resolution is sufficient for our purposes. Incompressibility is achieved via Helmholtz decomposition of the velocity field using a conjugate gradient solve.

We initialize the whole domain with a velocity field sampled from random noise in frequency space, resulting in eddies of various sizes. The initial velocity values have a mean of zero and a standard deviation of 0.5. Then, ground truth values for x_0 and $\vec{v}_0$ are sampled from uniform distributions with $\vec{v}_0^y \geq 0$ never pointing downward. These values are used to initialize a spherical force or wind blast near the bottom of the domain that moves upwards during the simulation and induces flow around all obstacles from the pressure computation. The velocity is only observable in the domain's upper half, and all optimizers assume a zero-initialization in the unobservable bottom half.

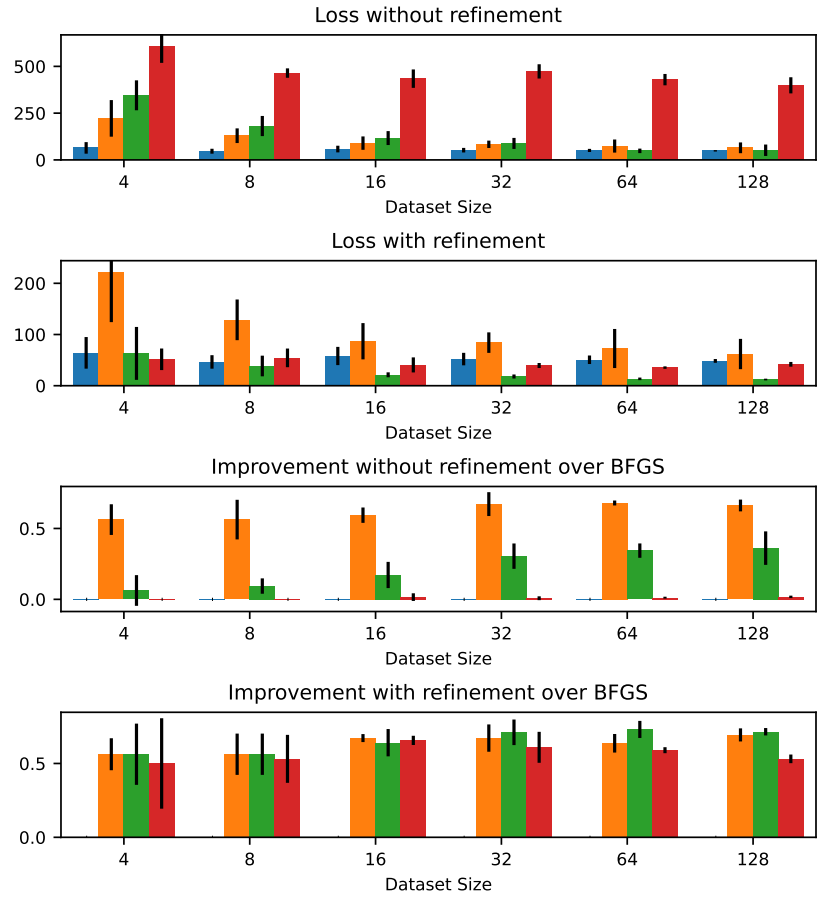


Figure 20: Loss and improvement over BFGS before and after refinement for the Navier-Stokes experiment. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint.

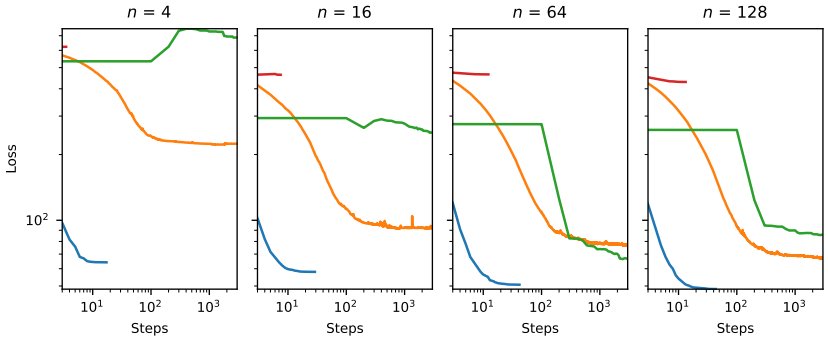


Figure 21: Optimization curves for different data set sizes of the Navier-Stokes experiment before refinement. Envelopes show the standard deviation over 4 network and data set initialization seeds. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

Networks. The surrogate network approximates the final state $u(x, y \geq 50, t = 56)$ in the upper half of the domain from the initial state $u(x, y, t = 0)$ and the parameters $x_0, \vec{v}_0$. As before, we implement this as G2G (section A) with five input and two output feature maps, totaling 38.290 parameters. The G2S reparameterization network comprises four convolutional layers with 16, 32,

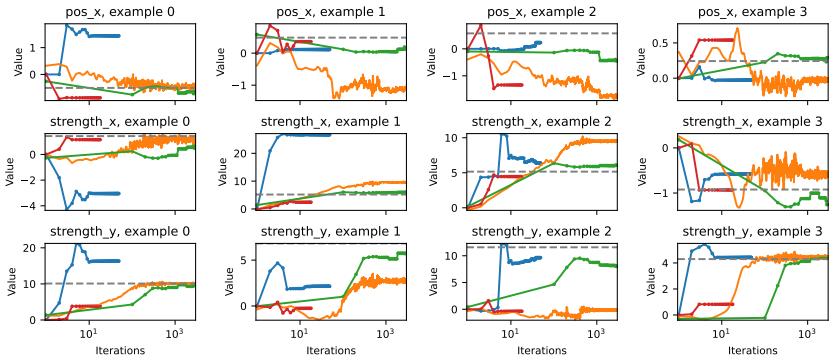


Figure 22: Example parameter evolution during optimization of the Navier-Stokes experiment with $n = 128$. Blue: BFGS, orange: reparameterized, green: supervised, red: neural adjoint. The dashed gray lines indicate the reference solution from which the example was generated. BFGS-based optimization curves stop when all examples have fully converged to an optimum.

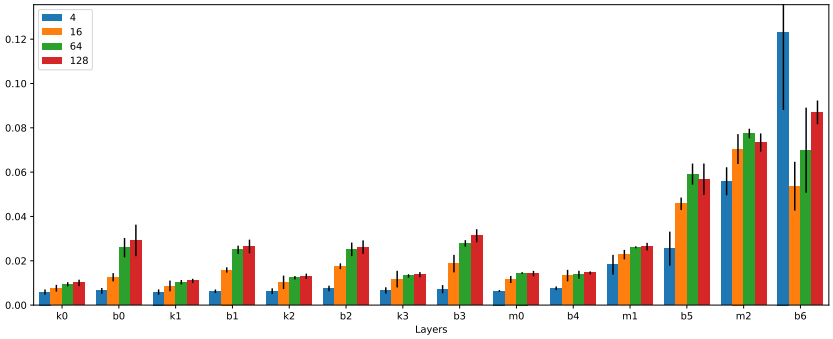
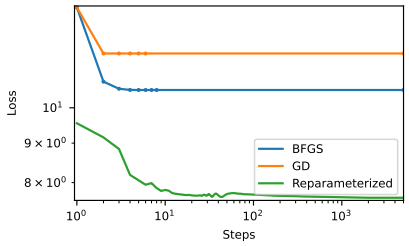


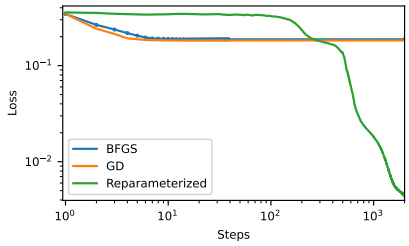
Figure 23: Reparameterization network change in the incompressible fluid experiment for different data set sizes n, measured as the mean absolute difference in weight values before and after fitting the network. Error bars represent the standard deviation across multiple network initializations. The change is given per layer where k denotes convolution kernels, m the matrices of fully-connected layers and b biases.

32, and 32 feature maps, respectively, followed by two fully-connected layers with 64 neurons each, resulting in 44.723 total parameters. Both networks are trained using Adam with a learning rate of $\eta = 0.001$. Fig. 23 shows the absolute the change in the reparameterization network weights that results from training.

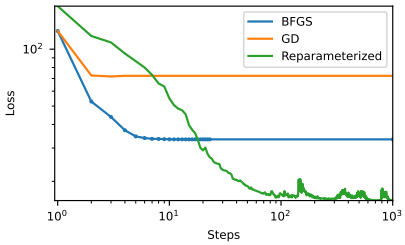
Additional results. As noted in the main text, the high loss value of the reparameterized optimization is largely due to a fraction of examples with considerably higher loss than the average. A summary of the individual loss values for $n = 4, 128$ is given in Figs. 18 and 19, respectively. While the neural adjoint method produces the highest loss values before refinement, these all get mapped to relatively small values during the refinement stage. Meanwhile, the reparameterized training finds better solutions without refinement since it uses feedback from F. However, that also means that the secondary BFGS optimization cannot improve the estimates by nearly as much since many are already close to a (local) minimum. This leaves a fraction of examples stranded on sub-optimal solutions that contribute significantly to the total loss, despite most problems finding better solutions than BFGS. Fig. 20 shows the resulting loss and improvement over BFGS, both before and after refinement. The learning curves for four data set sizes n are shown in Fig. 21, and the parameter evolution of four examples during optimization are shown in Fig. 22.



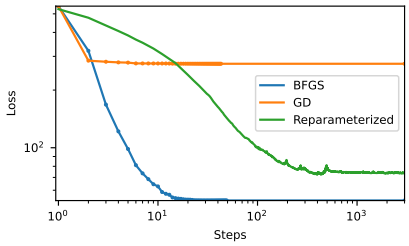
(a) Wave packet experiment



(b) Billiards experiment



(c) Kuramoto–Sivashinsky experiment



(d) Incompressible fluids experiment

Figure 24: Optimization curves for gradient descent (GD), BFGS and reparameterization similar to subfigures c in Figs. 2-5.

Table 7: Fraction of examples for which gradient descent (GD) outperforms or is equal to BFGS.

Experiment	GD better	Equal to BFGS
Wave packet fit	18.4%	24.6%
Billiards	0.8%	30.9%
Kuramoto–Sivashinsky	7.0%	0.4%
Incompr. Navier-Stokes	3.9%	0.0%

B.5 Gradient descent as a baseline

In the main text, we use BFGS as a baseline as it works well for most optimization problems and is very stable. However, in the field of machine learning, first-order methods, such as gradient descent (GD) are more popular since they do not require the Hessian matrix, whose memory requirement scales quadratically with the number of parameters.

However, to avoid accidental bias in our experiments, we run GD with adaptive step size as an additional baseline. Example learning curves are shown in Fig. 24 for the four experiments described in the main text. GD generally performs worse than BFGS in our experiments. Table 7 shows the fractions of examples in which GD performs better than or equal to BFGS.

Consequently, our method shows a bigger improvement over GD than BFGS, albeit slightly. In the incompressible fluids experiment with $n = 128$, for example, the reparameterized and refined optimization outperforms GD in 88.3% of cases vs 86.0% when compared against BFGS.

B.6 ADDITIONAL EXPERIMENTS

To compare our method with previous work, we replicate the robotic arm experiment from the neural adjoint paper (Ren et al., 2020). However, the original paper did not use the functional form of the forward process, i.e. the true simulation. When using it, the problems presented there become much easier to solve. The results are shown in Fig. 25. BFGS and gradient descent (GD) manage to reach machine precision accuracy within a couple of iterations while the network approaches take longer to fit the data (3c). The refinement stage then optimizes all examples to machine precision accuracy (3d). While reparameterized fitting successfully solves these experiments, there is no need to use it since they can be solved perfectly with classical optimizers. This stands in contrast to the inverse problems shown in the main text which exhibit non-trivial features, such as local optima, zero-gradient regions, or chaotic behavior. However, this experiment shows, that reparameterized fitting can be applied to convex optimization problems in the same way.

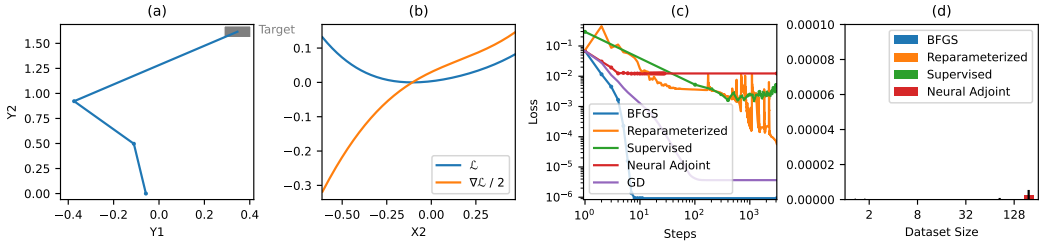


Figure 25: Robotic arm experiment from [RPM20]. **(a)** Task: the arm must be positioned and the joints rotated to reach the target, **(b)** corresponding loss and gradient landscape for the first joint angle (x_2), **(c)** optimization curves without refinement, **(d)** refined loss L/n by number of examples n, mean and standard deviation over multiple network initializations and data sets. All solutions converge to the global optimum with zero loss.