A Gradient clipping

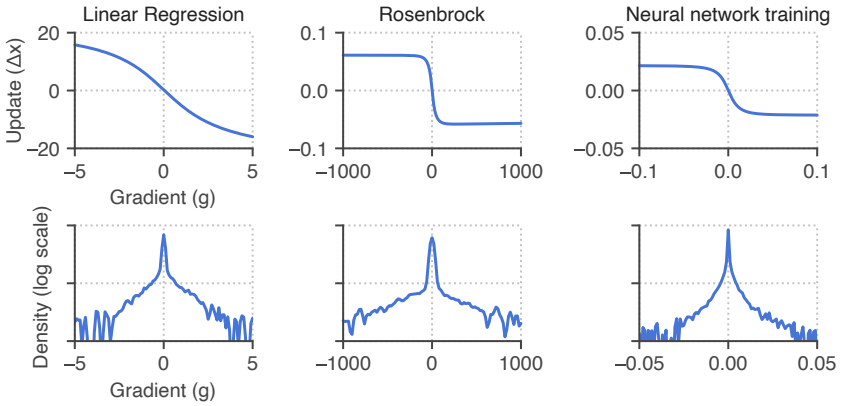

Figure 6: Gradient clipping in a learned optimizer. **Top row**: The update function computed at the initial state saturates for large gradient magnitudes. The effect of this is similar to that of gradient clipping (cf. Fig. 2b). **Bottom row**: the empirical density of encountered gradients for each task (note the different ranges along the x-axes). Depending on the problem, the learned optimizer can tune its update function so that most gradients are in the linear portion of the function, and thus not use gradient clipping (seen in the linear regression task, left column) or can potentially use more of the saturating region (seen in the Rosenbrock task, middle column).

In standard gradient descent, the parameter update is a linear function of the gradient. Gradient clipping (Pascanu et al., 2013) instead modifies the update to be a saturating function (Fig. 2b).

We find that learned optimizers also use saturating update functions as the gradient magnitude increases, thus learning a soft form of gradient clipping (Figure 6). Although we show the saturation for a particular optimize state (the initial state, top row of Fig. 6), we find that these saturating thresholds are consistent throughout the state space.

The strength of the clipping effect depends on the training task. We can see this by comparing the update function to the distribution of gradients encountered for a given task (bottom row of Fig. 6). For some problems, such as linear regression, the learned optimizer largely stays within the linear region of the update function (Fig. 6, left column). For others, such as the Rosenbrock problem (Fig. 6, right column), the optimizer utilizes more of the saturating part of the update function.

B A learned optimizer that recovers momentum

When training learned optimizers on the linear regression tasks, we noticed that we could train a learned optimizer that seemed to strongly mimic momentum, both in terms of behavior and performance. With additional training, the learned optimizer would eventually start to outperform momentum (Figure 1a). We highlight this latter, better performing optimizer in the main text. However, it is still instructive to go through the analysis for the learned optimizer that mimics momentum. This example in particular clearly demonstrates the connections between eigenvalues, momentum, and dynamics.

The learned optimizer that performs as well as momentum learns to mimic linear dynamics (we also used a GRU for this optimizer). That is, the dynamics of the nonlinear optimizer could be very well approximated using a linearization computed at the convergence point. This linearization is shown in Figure 7. We find a single mode pops out of the bulk of eigenvalues (Fig. 7a). Additionally, if we plot these eigenvalue magnitudes, which are the momentum time scales, against the corresponding extracted learning rate of each mode, as discussed below in Appendix C), we see that this mode also has a large learning rate compared to the bulk (top right blue circle in Fig. 7b). Moreover, the extracted momentum timescale and learning rate for this mode essentially exactly match the best tuned hyperparameters (gold star in Fig. 7b) from tuning the momentum algorithm directly, which can also be derived from theory.

Finally, if we extract and run just the dynamics along this particular mode, we see that it matches the behavior of the full, nonlinear optimizer almost exactly (Fig. 7c). This suggests that in this scenario, the learned optimizer has simply learned the single mechanism of momentum. Moreover, the learned optimizer has encoded the best hyperparameters for this particular task distribution in its dynamics. Our analysis shows how to separate the overall mechanism (linear dynamics along eigenmodes) from the particular hyperparameters of that mechanism (the specific learning rate and momentum timescale).

C Linearized optimizers and aggregated momentum

In this section, we elaborate on the connections between linearized optimizers and momentum with multiple timescales. We begin with our definition of an optimizer, equations (1) and (2) in the main text:

$$\begin{aligned} \boldsymbol{h}^{k+1} &= F(\boldsymbol{h}^k, g^k) \\ x^{k+1} &= x^k + \boldsymbol{w}^T \boldsymbol{h}^{k+1}, \end{aligned}$$

where $\boldsymbol{h}$ is the optimizer state, g is the gradient, x is the parameter being optimized, and k is the current iteration. Note that since this is a component-wise optimizer, it is applied to each parameter (x_i) of the target problem in parallel; therefore we drop the index (i) to reduce notation.

Near a fixed point of the dynamics, we approximate the recurrent dynamics with a linear approximation. The *linearized* state update can be expressed as:

$$F(\boldsymbol{h}^k, g^k) \approx \boldsymbol{h}^* + \frac{\partial F}{\partial \boldsymbol{h}} \left(\boldsymbol{h}^k - \boldsymbol{h}^*\right) + \frac{\partial F}{\partial g} g^k, \tag{3}$$

where $\boldsymbol{h}^*$ is a fixed point of the dynamics, $\frac{\partial F}{\partial \boldsymbol{h}}$ is a square matrix known as the Jacobian, and $\frac{\partial F}{\partial g}$ is a vector that controls how the scalar gradient enters the system. Both of these latter two quantities are evaluated at the fixed point, $\boldsymbol{h}^*$, and $g^* = 0$.

For a linear dynamical system, as we have now, the dynamics decouple along eigenmodes of the system. We can see this by rewriting the state in terms of the left eigenvectors of the Jacobian matrix. Let $\boldsymbol{v} = \boldsymbol{U}^T \boldsymbol{h}$ denote the transformed coordinates, in the left eigenvector basis $\boldsymbol{U}$ (the columns of $\boldsymbol{U}$ are left eigenvectors of the matrix $\frac{\partial F}{\partial \boldsymbol{h}}$). In terms of these coordinates, we have:

$$\boldsymbol{v}^{k+1} = \boldsymbol{v}^* + \mathbf{B}\left(\boldsymbol{v}^k + \boldsymbol{v}^*\right) + \boldsymbol{a} g^k, \tag{4}$$

where $\mathbf{B}$ is a diagonal matrix containing the eigenvalues of the Jacobian, and $\boldsymbol{a}$ is a vector obtained by projecting the vector that multiplies the input $\left(\frac{\partial F}{\partial g}\right)$ from eqn. (3) onto the left eigenvector basis.

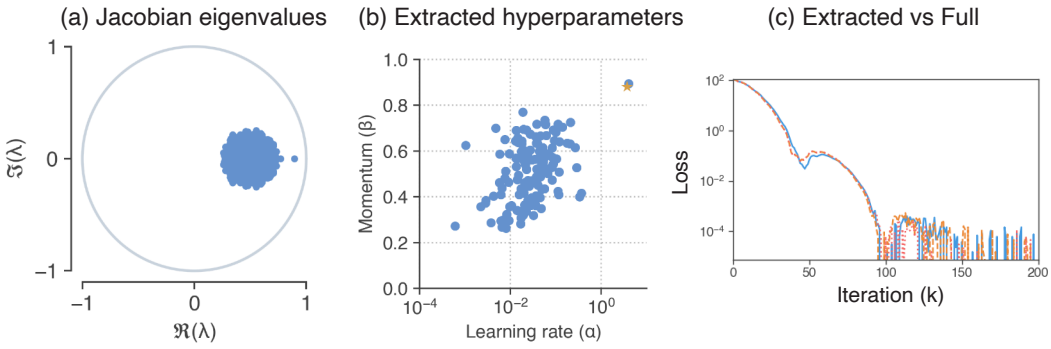

Figure 7: A learned optimizer that recovers momentum on the linear regression task. **(a)** Eigenvalues of the Jacobian of the optimizer dynamics evaluated at the convergence fixed point. There is a single eigenmode that has separated from the bulk. **(b)** Another way of visualizing eigenvalues is by translating them into optimization parameters (learning rates and momentum timescales), as described in Appendix C. When we do this for this particular optimizer, we see that the slow eigenvalue (momentum timescale closest to one) also has a large learning rate. These specific hyperparameters match the best tuned momentum hyperparametrs for this task distribution (gold star). **(c)** When we extract and run just the dynamics along this single mode (orange dashed line), we see that this reduced optimizer matches the full, nonlinear optimizer (solid line) almost exactly.

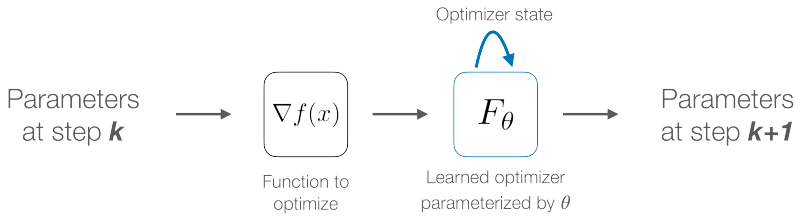

Figure 8: Schematic of a learned optimizer.

If we have an N-dimensional state vector $\boldsymbol{h}$, then eqn. (4) defines N independent (decoupled) scalar equations that govern the evolution of the dynamics along each eigenvector: $v_j^{k+1} = v_j^* + \beta_j \left(v_j^k + v_j^*\right) + \alpha_j g^k$, where we use β_j to denote the j^{th} eigenvalue and α_j is the j^{th} component of $\boldsymbol{a}$ in eqn. (4). Collecting constants yields the following simplified update:

$$v_j^{k+1} = \beta_j v_j^k + \alpha_j g + \text{const.}, \tag{5}$$

which is exactly equal to the momentum update ($v^{k+1} = \beta v^k + \alpha g^k$), up to a (fixed) additive constant. The main difference between momentum and the linearized momentum in eqn. (5) is that we now have N different momentum timescales. Again these timescales are exactly the eigenvalues of the Jacobian matrix from above. Moreover, we also have a way of extracting the corresponding learning rate associated with eigenmode j, as α_j. This particular optimizer (momentum with multiple timescales) has been proposed under the name *aggregated momentum* by Lucas et al. (2018).

Taking a step back, we have drawn connections between a linearized approximation of a nonlinear optimizer, and a form of momentum with multiple timescales. What this now allows us to do is interpret the behavior of learned optimizers near fixed points through this new lens. In particular, we have a way of translating the parameters of a dynamical system (Jacobians, eigenvalues and eigenvectors) into more intuitive optimization parameters (learning rates and momentum timescales).

D Supplemental methods

D.1 Tasks for training learned optimizers

An optimization problem is specified by both the loss function to minimize and the initial parameters. When training a learned optimizer (or tuning baseline optimizers), we sample this loss function and initial condition from a distribution that defines a task. Then, when evaluating an optimizer, we sample new optimization problems from this distribution to form a test set.

The idea is that the learned optimizer will discover useful strategies for optimizing the particular task it was trained on. By studying the properties of optimizers trained across different tasks, we gain insight into how different types of tasks influence the learned algorithms that underlie the operation of the optimizer. This sheds insight on the inductive bias of learned optimizers; i.e. we want to know what properties of tasks affect the resulting learned optimizer and whether those strategies are useful across problem domains.

We train and analyzed learned optimizers on three distinct tasks. In order to train a learned optimizer, for each task, we must repeatedly initialize and run the corresponding optimization problem (resulting in thousands of optimization runs). Therefore we focused on simple tasks that could be optimized within a couple hundred iterations, but still covered different types of loss surfaces: convex and non-convex functions, over low- and high-dimensional parameter spaces. We also focused on deterministic functions (whose gradients are not stochastic), to reduce variability when training and analyzing optimizers.

D.2 Training a learned optimizer

We train learned optimizers that are parameterized by recurrent neural networks (RNNs). In all of the learned optimizers presented here, we use gated recurrent unit (GRU) (Cho et al., 2014) to parameterize the optimizer. This means that the function F in eqn. (1) is the state update function

of a GRU, and the optimizer state is the GRU state. In addition, for all of our experiments, we set the readout function r in eqn. (2) to be linear. The parameters of the learned optimizer are now the GRU parameters, and the weights of the linear readout. We meta-learn these parameters through a meta-optimization procedure, described below.

In order to apply a learned optimizer, we sample an optimization problem from our task distribution, and iteratively feed in the current gradient and update the problem parameters, schematized in Figure 8. This iterative application of an optimizer builds an unrolled computational graph, where the number of nodes in the graph is proportional to the number of iterations of optimization (known as the length of the unroll). This is sometimes called the *inner* optimization loop, to contrast it with the *outer* loop that is used to update the optimizer parameters.

In order to train a learned optimizer, we first need to specify a target objective to minimize. In this work, we use the average loss over the unrolled (inner) loop as this meta-objective. In order to minimize the meta-objective, we compute the gradient of the meta-objective with respect to the optimizer weights. We do this by first running an unrolled computational graph, and then using backpropagation through the unrolled graph in order to compute the meta-gradient.

This unrolled procedure is computationally expensive. In order to get a single meta-gradient, we need to initialize, optimize, and then backpropagate back through an entire optimization problem. This is why we focus on small optimization problems, that are fast to train.

Another known difficulty with this kind of meta-optimization arises from the unrolled inner loop. In order to train optimizers on longer unrolled problems, previous studies have *truncated* this inner computational graph, effectively only using pieces of it in order to compute meta-gradients. While this saves computation, it is known that this induces bias in the resulting meta-gradients (Wu et al., 2018; Metz et al., 2019).

To avoid this, we compute and backpropagate through fully unrolled inner computational graphs. This places a limit on the number of steps that we can then run the inner optimization for, in this work, we set this unroll length to 200 for all three tasks. Backpropagation through a single unrolled optimization run gives us a single (stochastic) meta-gradient, when meta-training, we average these over a batch size of 32.

Now that we have a procedure for computing meta-gradients, we can use these to iteratively update parameters of the learned optimizer (the outer loop, also known as meta-optimization). We do this using Adam as the meta-optimizer, with the default hyperparameters (except for the initial learning rate, which was tuned via random search). In addition, we use gradient clipping (with a clip value of five applied to each parameter independently and decay the learning rate exponentially (by a factor of 0.8 every 500 steps) during meta-training. We added a small ℓ_2-regularization penalty to the parameters of the learned optimizer, with a penalty strength of 10^{-5}. We trained each learned optimizer for a total of 5000 steps.

For each task, we ended up with a single (best performing) learned optimizer architecture. These are the optimizers that we then analyzed, and form the basis of the results in the main text. The final meta-objective for each learned optimizer and best tuned baselines are compared below in Figure 9.

D.3 Hyperparameter selection for baseline optimizers

We tuned the hyperparameters of each baseline optimizer, separately for each task. For each combination of optimizer and task, we randomly sampled 2500 hyperparameter combinations from a grid, and selected the best one using the same meta-objective that was used for training the learned optimizer. We ensured that the best parameters did not occur along the edge of any grid.

For momentum, we tuned the learning rate (α) and momentum timescale (β). For RMSProp, we tuned the learning rate (α) and learning rate adaptation parameter (γ). For Adam, we tuned the learning rate (α), momentum (β_1), and learning rate adaptation (β_2) parameters. The result of these hyperparameter runs are shown in Figures 10 (linear regression), 11 (Rosenbrock), and 12 (two moons classification). In each of these figures, the color scale is the same — purple denotes the optimal hyperparameters.

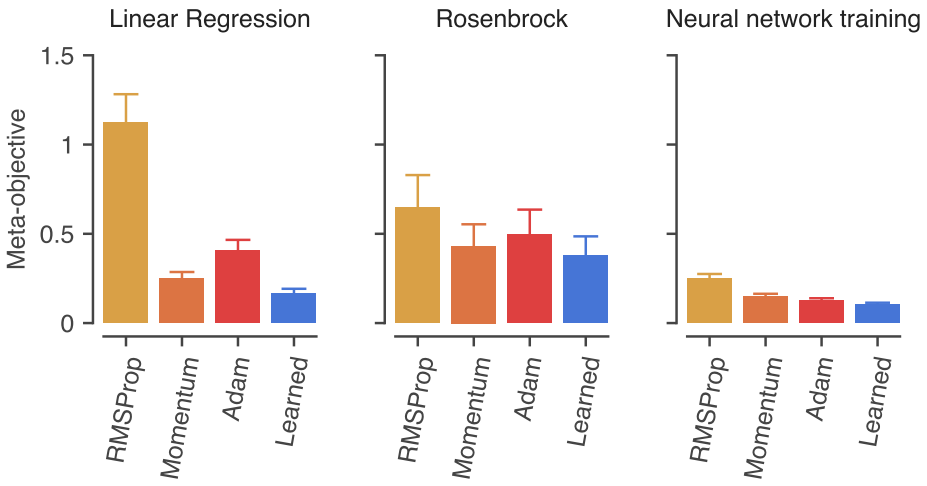

Figure 9: Performance summary. Each panel shows the meta-objective (average training loss) over 64 random test problems for baseline and learned optimizers. Error bars show standard error. The learned optimizer has the lowest (best) meta-objective on each task.

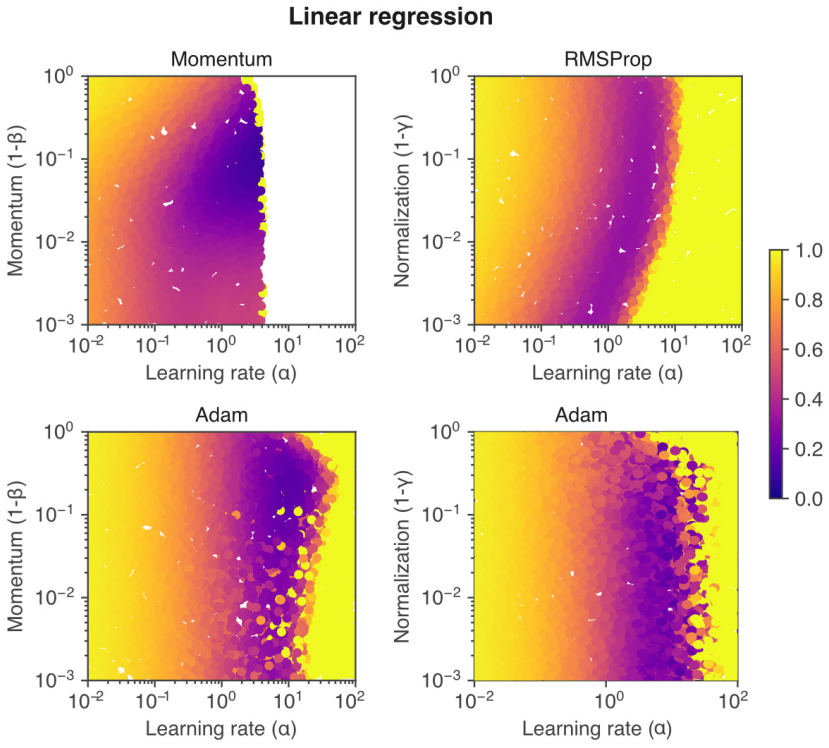

Figure 10: Hyperparameter selection for linear regression.

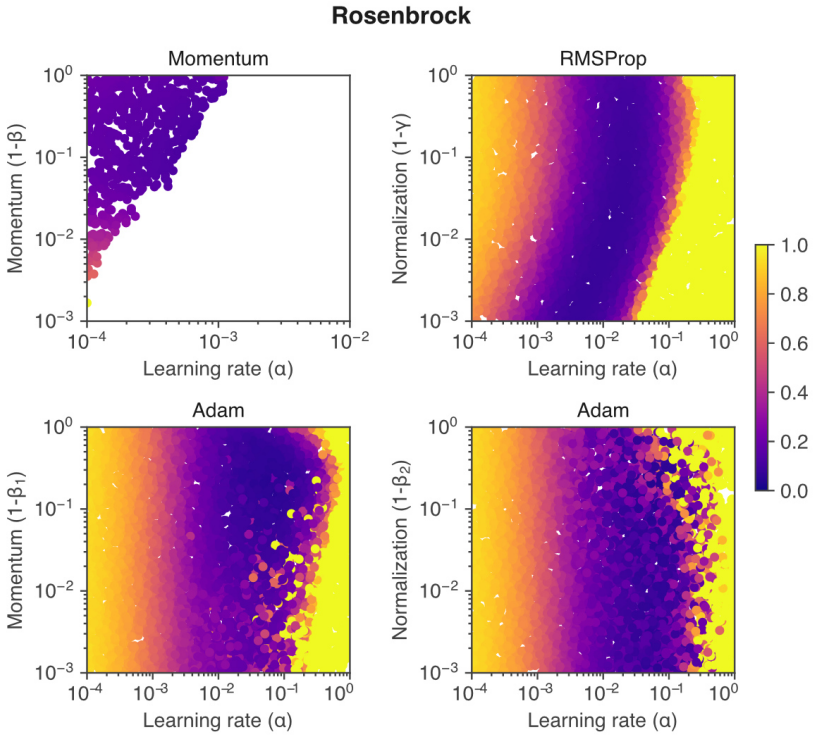

Figure 11: Hyperparameter selection for Rosenbrock.

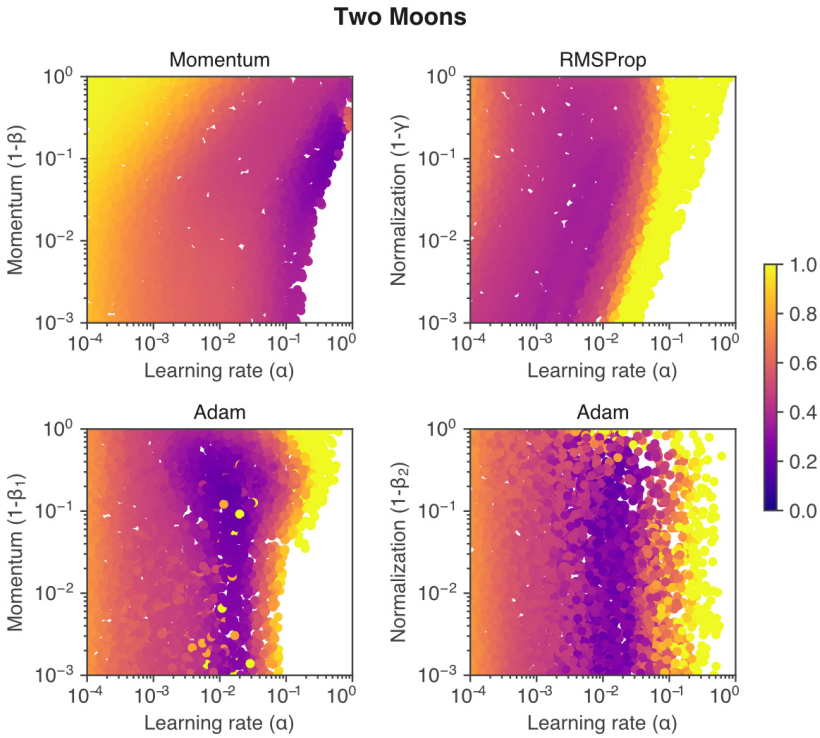

Figure 12: Hyperparameter selection for training a neural network on two moons data.