Appendix for

Reverse engineering learned optimizers reveals known and novel mechanisms

A Comparing results across all mechanisms and tasks

Below, we present and discuss results comparing mechanisms in learned optimizers across the different tasks studied (linear regression, Rosenbrock, Two Moons, and MNIST classification; see §3). This comparative anatomy across networks lets us see both how general particular mechanisms are, and hints at possible ways in which learned optimizers become tuned to a particular task distribution, at the expense of generalizing to other (untrained) tasks.

A.1 Momentum

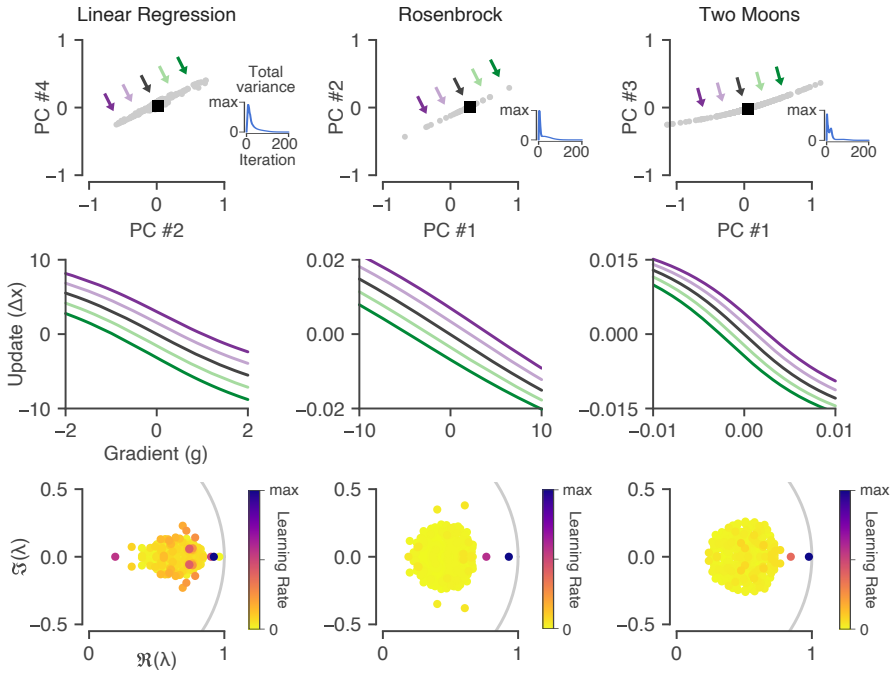

Figure 8: Momentum in learned optimizers. Each column shows the same phenomena, but for optimizers trained on different tasks. **Top row**: Projection of the optimizer state around a convergence point (black square). *Inset:* the total variance of the optimizer states over test problems goes to zero as the trajectories converge. **Middle row**: visualization of the update functions (§4.1) along the slow mode of the dynamics (colored lines correspond to arrows in (a)). Along this dimension, the effect on the system is to induce an offset in the update, just as in classical momentum (cf. Fig. 2c). **Bottom row**: Eigenvalues of the linearized optimizer dynamics at the convergence fixed point (black square in top row) plotted in the complex plane. The eigenvalue magnitudes are momentum timescales, and the color indicates the corresponding learning rate.

We found highly consistent momentum mechanisms across the first three tasks studied (linear regression, Rosenbrock, and Two Moons classification). Results for these are shown in Figure 8. Across these three tasks, we find that optimizer state trajectories converge to a single (global) fixed point (top row). This can be seen as the total variance across hidden states (inset in top row) goes to zero as training progresses. The dynamics around these final convergence points are organized around an approximate 1D line (indicated by gray dots in Fig. 8). When moving along this line, the effect on the update function is to induce a vertical offset (Fig. 8, middle row). Finally, the dynamics around these fixed points can be approximated using a linear approximation. This is done by computing the Jacobian of the system at the fixed point, and looking at its eigenvalues and eigenvectors. The bottom row of Figure 8 shows these eigenvalues in the complex plane. Across tasks, we find one or

two modes that seem responsible for the dynamics. The eigenvalue of these modes can be interpreted as momentum timescales.

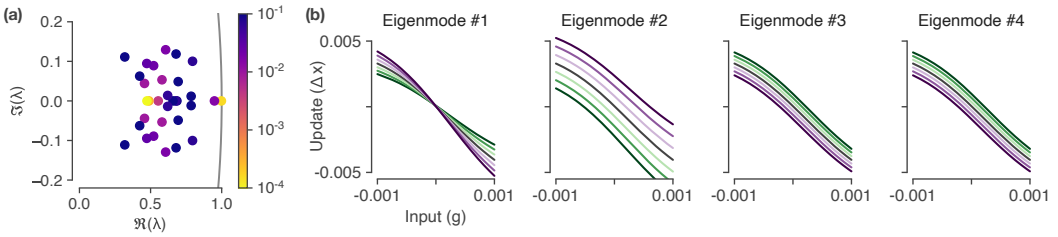

Figure 9: Momentum in a learned optimizer trained on MNIST. **(a)** Eigenvalues of the Jacobian of the optimizer dynamics, computed at an approximate convergence fixed point. There are multiple modes with large learning rates that seem to all contribute to the dynamics. **(b)** Update functions corresponding to moving along different eigenvectors of the Jacobian. For the first of these eigenmodes, the effect seems to be a change in the learning rate. For the rest, we see behavior that resembles momentum (a vertical shift in the update function).

In a learned optimizer trained on MNIST, we found that trajectories did not converge to just one fixed point, but instead converged to any number of fixed points along a 1D manifold. The purpose of this manifold appears to be used for a type of learning rate adaptation, discussed more below in section A.4. What this means for the momentum mechanism, is twofold. First, the total variance of the trajectories no longer goes to zero, instead, it remains elevated as the trajectories for different parameters. Second, we find that there is not a single dominant eigenmode that seems responsible for momentum, the effect is spread out across multiple eigenmodes.

This is shown in Figure 9. Fig. 9a shows the eigenvalues of the Jacobian computed at one of the convergence fixed points along the manifold (others are similar). Instead of a single dominant mode, as observed for the other tasks, there are a number of modes with large learning rates. Fig. 9b shows the effect of moving along the eigenvector associated with these top eigenvalues in state space. The first eigenmode corresponds to moving along the fixed point manifold[7] and is discussed more below in the section on learning rate adaptation (A.4). However, eigenmodes 2-4 (and beyond) all have dynamics that look like momentum; our interpretation of this is that the network is using multiple momentum timescales as opposed to just one in this case.

A.2 Gradient clipping

The gradient clipping effect (discussed in §5.2) can be tuned by learned optimizers by making the update function more linear, or more saturating, over the range of expected gradient magnitudes. Indeed, we find this is the case for learned optimizers trained across the four tasks (Figure 10). For some tasks, such as linear regression, the update functions look quite linear. For others, such as the Rosenbrock task, the network uses much more of the saturating part of the curve. Perhaps this is due to the large changes in curvature in the (non-convex) loss surface; clipping allows the network to have large learning rates in the valley of the Rosenbrock function, while not diverging if a step happens to take the network into the high curvature region outside the curved valley.

For the MNIST task (far right column of Fig. 10), we find that the update function does not quite saturate, but instead slopes backwards, giving a non-monotonic update function. We found this consistently across multiple learned optimizers trained using different random seeds; however, the functional benefit of this non-monotonic behavior (if any) remains a mystery.

A.3 Learning rate schedules

Learned optimizers can learn an effective learning rate schedule through the use of autonomous (as opposed to input-driven) dynamics. The input to the learned optimizer is the gradient with respect to

[7] A oen dimensional fixed point manifold is an approximate line attractor; for a discrete time linear dynamical system this moving along the manifold corresponds to moving along an eigenvector whose corresponding eigenvalue is on the unit circle. Indeed, we see that the first eigenmode corresponds to the eigenvalue on the unit circle in Fig. 9a.

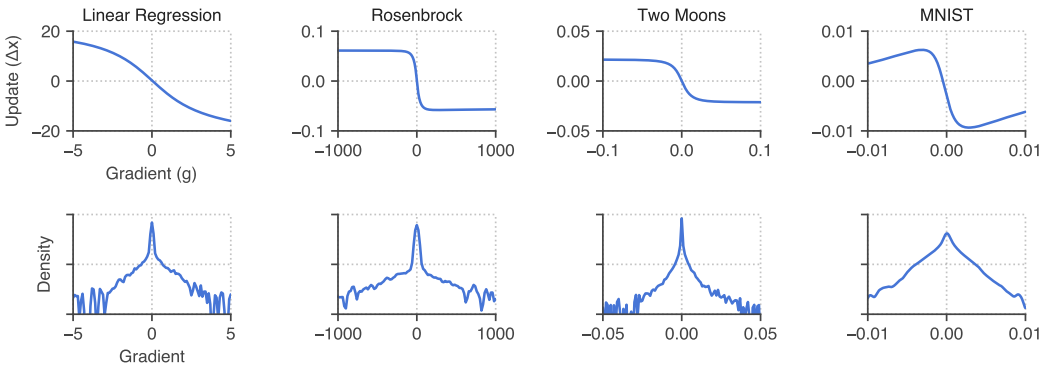

Figure 10: Gradient clipping in learned optimizers, trained on four different tasks. **Top row**: The update function computed at the initial state saturates for large gradient magnitudes. This effect is similar to that of gradient clipping (cf. Fig. 2b). **Bottom row**: the empirical density of encountered 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 linear regression, left column) or can potentially use more of the saturating region (seen on the Rosenbrock task, middle left).

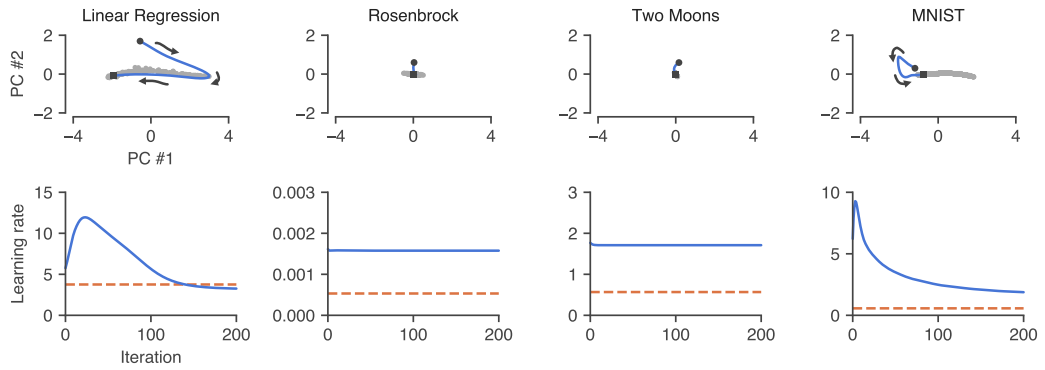

Figure 11: Learning rate schedules mediated by autonomous dynamics. **Top row**: Low-dimensional projection of the dynamics of the learned optimizer in response to zero gradients (no input). These autonomous dynamics allow the system to learn a learning rate schedule (see §5.3). Gray circles are approximate fixed points of the dynamics. **Bottom row**: Effective learning rate (measured as the slope of the update function) as a function of iteration during the autonomous trajectories in the top row. We only observe a clear learning rate schedule in the linear regression task (far left) and the MNIST task (far right), both of which include a warm-up and decay. For context, dashed lines indicate the best (tuned) learning rate for momentum.

a particular parameter. When that derivative is zero, the network should not update the parameter. However, this does not mean that the hidden state needs to remain constant, instead, the hidden state is free to evolve along a subspace orthogonal to the readout used to update the parameter.

The functional benefit of this is that through the use of autonomous dynamics, the network can effectively change its behavior as a function of time (i.e. the current optimization step). One type of behavioral change that we observe is a change in learning rate. This is prominent in learned optimizers trained on linear regression (far left column of Fig. 10) and on MNIST (far right column of Fig. 10).

A.4 Learning rate adaptation

The final phenomenon we found across tasks is learning rate adaptation. The overall effect of this phenomenon is to reduce (increase) the learning rate for parameters as their corresponding gradient

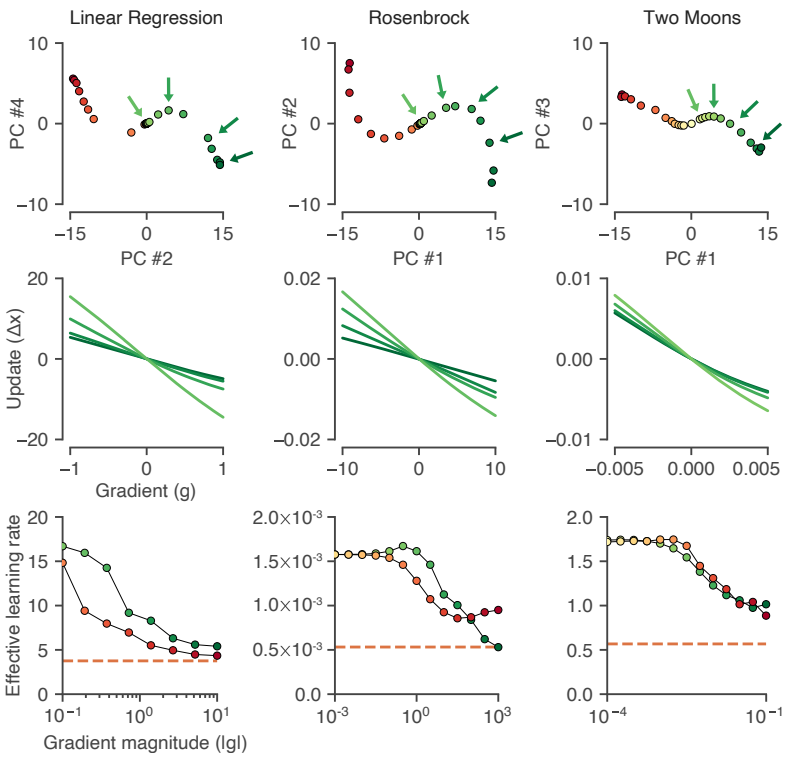

Figure 12: Learning rate adaptation in learned optimizers. **Top row**: Approximate fixed points (colored circles) of the dynamics computed for different gradients reveal an S-curve structure. **Middle row**: Update functions (§4.1) computed at different points along the S-curve (corresponding to arrows from the top row). The effect of moving towards the edge of the S-curve is to make the update function more shallow (thus have a smaller effective learning rate, cf. Fig. 2d). The effect is similar along both arms; only one arm is shown for clarity. **Bottom row**: Summary plot showing the effective learning rate along each arm of the S-curve, for negative (red) and positive (green) gradients. The overall effect is to reduce learning rates when the gradient magnitude is large.

magnitudes get large (small). While this effect is similar to that of RMSProp or Adam, we found the underlying mechanism in learned optimizers to be different.

For learned optimizers trained on the first three tasks, this mechanism is mediated by *input-dependent* fixed points. These points are found by running the numerical optimization routine discussed in §4.2, but while holding the input fixed at various non-zero levels (rather than zero). The significance of this is that in the presence of large gradients, the network hidden state is away from the convergence point, and instead attracted towards different points (the input-dependent fixed points). The speed at which the hidden state approaches these fixed points depends on their dynamical properties, presumably this is something that can be tuned by the network as well. The functional benefit of this is that it allows the network to adapt its behavior *after encountering large gradients for a number of consecutive iterations*.

Figure 12 breaks down the mechanism found across the first three tasks. The top row of Fig. 12 shows the set of input-dependent fixed points, which across these tasks forms an S-shaped curve. Large positive gradients (shown in green) pull the state along one arm of the S-curve, and large negative gradients (shown in red) pull the state along the opposite arm; the dynamics are symmetric across positive and negative gradients.

Along this curve, the network changes its behavior through a changing learning rate. We can see this as the slope of the update function changes (middle row of Fig. 12), these green lines correspond to the locations given by the arrows in the top row. Only changes along one arm of the S-curve are shown for clarity (for positive gradients), the effects are similar for negative gradients. The effect is shown for both arms in the bottom row of Fig. 12, summarized as the slope of the update function

(the effective learning rate). Again, we see that the effect is to reduce the learning rate for large gradients. Interestingly, the most conservative learning rates for each task seem to match the best tuned learning rates used by momentum (shown as the dashed line). We hypothesize that momentum (whose learning rate does not adapt) must constrain its learning rate so that it does not diverge even in the off chance that a large gradient is encountered. Learned optimizers, on the other hand, are able to increase the learning rate as the gradient magnitudes decrease, which presumably partially explains their improved performance.

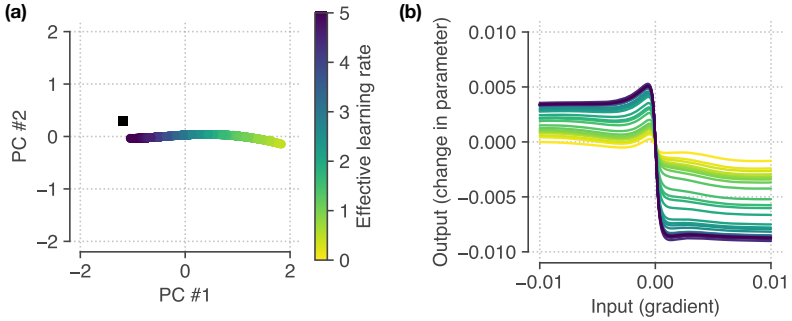

Figure 13: Variation in learning rate in a learned optimizer trained to optimize a network on MNIST classification. **(a)** Initial state (black square) and approximate fixed points (colored circles) found in a network trained on MNIST. Instead of a single fixed point, we see an approximately one dimensional manifold of fixed points. **(b)** Update functions corresponding to the points in the manifold in (a). Along this manifold, the overarching effect is a change in the effective learning rate.

Finally, this was another mechanism where we see a different mechanism for the learned optimizer trained on MNIST classification. Here, we find that the network learns a manifold of fixed points (not just one) even when the input is held at zero. That is, while learned optimizers trained on the other tasks all converged to a single global fixed point, the MNIST networks converged to a 1D manifold of approximate fixed points. This manifold is shown in Figure 13a. The functional benefit of this seems to be to induce a type of learning rate adaptation. Along this manifold, the update functions change dramatically (Fig. 13b). One consequence of this is that the effective learning rate changes. This denoted by the variation in color in Fig. 13, going from small learning rates (yellow) to large ones (purple). We suspect these fixed points are responsible for the dynamics that separates trajectories based on the underlying parameter type and layer (discussed in §5.5).

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. 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.

B.1 Recovering momentum using linear dynamics

The learned optimizer (parameterized by a GRU) that performs as well as momentum learns to mimic linear dynamics. That is, the dynamics of the nonlinear optimizer can be very well approximated using a linear approximation computed at the convergence point. The dynamics of this optimizer (projected onto the top principal components of the optimizer state space) are shown in Figure 14. A single principal component explains nearly all of the variance in hidden states (right inset of Fig. 14a). The optimizer state trajectories for an example problem are shown in Fig. 14a (recall that this problem is a five dimensional quadratic, so there are five trajectories). All trajectories converge to a single global fixed point, indicated by a black circle.

The update functions for this optimizer along the first principal component are shown in Fig. 14b (c.f. Fig. 2 and Fig. 3). The effect of moving along the first principal component looks exactly

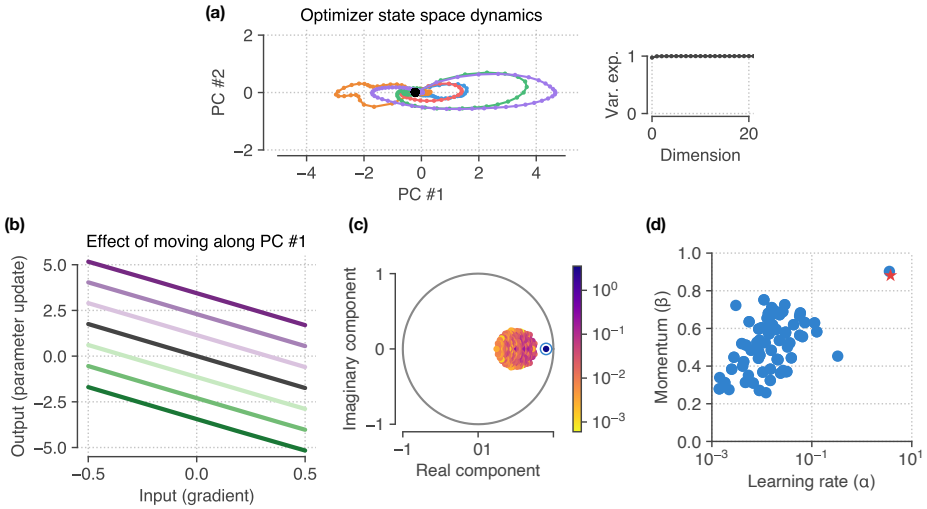

Figure 14: A learned optimizer that recovers momentum on the linear regression task. **(a)** Optimizer state dynamics after training, projected onto the top principal components of the optimizer state space. Most of the action happens along a single dimension (PC 1). The dynamics converge to a single global fixed point (black circle). **(b)** Update functions of the optimizer along the first principal component. Moving along this component induces a vertical offset in the update function. **(c)** 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. The color of each point corresponds to the effective learning rate for that eigenmode. **(d)** Another way of visualizing eigenvalues is by translating them into optimization parameters (learning rates and momentum timescales), as described in section 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 (red star).

like what happens as you change the state variable in momentum. We can additionally analyze the dynamics of the optimizer hidden state by linearizing the nonlinear RNN dynamics around the fixed point. The eigenvalues of the Jacobian of the dynamics at the fixed point are shown in Fig. 14c. We find a single mode that pops out of the bulk of eigenvalues, indicated with a blue circle. The color in Fig. 14c corresponds to the effective learning rate of each eigenmode (as discussed below in section C).

Additionally, we can plot these eigenvalue magnitudes (which are the momentum time scales), against the extracted learning rate of each mode (Fig. 14d). The single mode that dominates the dynamics is in the upper right of the plot. Moreover, the extracted momentum timescale and learning rate for this mode match the best tuned hyperparameters (red star in Fig. 14d) from tuning the momentum algorithm directly, which can also be derived analytically.

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. 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. One benefit of our analysis is that we can now separate the overall mechanism (linear dynamics along eigenmodes) from the particular hyperparameters of that mechanism (the specific learning rate and momentum timescale).

B.2 Meta-training dynamics

For this example, we also examined how the properties of the learned dynamics changed during meta-training. Specifically, we looked at how the effective learning rate and momentum of the optimizer (extracted from the Jacobian of the dynamics at the fixed point) varied over the course of meta-training (Figure 15). Fig. 15a shows the evolution of both the learning rate and momentum parameters for each eigenmode of the Jacobian. Panels (b) and (c) show the meta-training dynamics of just the top eigenmode (the only one that is used).

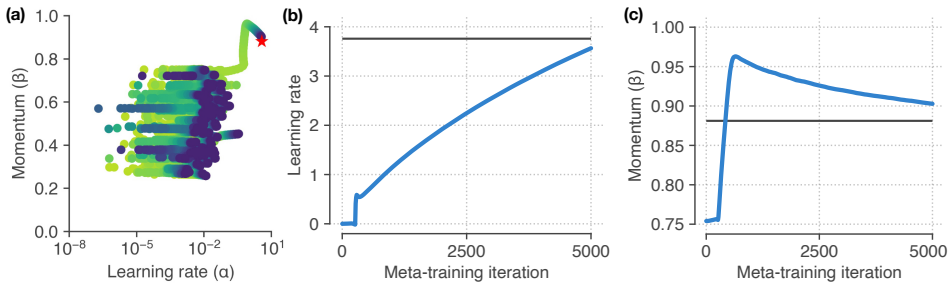

Figure 15: Meta-training dynamics. **(a)** Meta-training dynamics of the learning rate and momentum parameters, extracted from the Jacobian of the convergence fixed point. Color indicates the meta-training iteration, from initialization (yellow-green) to when the optimizer is fully trained (purple). The red star indicates the optimal momentum parameters, derived analytically. **(b)** and **(c)** How the learning rate and momentum parameters for the top eigenmode evolve over the course of meta-training. Horizontal black line indicates the optimal value, derived analytically.

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:

$$F(\boldsymbol{h}^k, g^k) \approx \boldsymbol{h}^* + \frac{\partial F}{\partial \boldsymbol{h}} \left(\boldsymbol{h}^k - \boldsymbol{h}^*\right) + \frac{\partial F}{\partial g} g^k, \tag{4}$$

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:

$$\boldsymbol{v}^{k+1} = \boldsymbol{v}^* + \mathbf{B}\left(\boldsymbol{v}^k + \boldsymbol{v}^*\right) + \boldsymbol{a} g^k, \tag{5}$$

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. (4) onto the left eigenvector basis.

If we have an N-dimensional state vector $\boldsymbol{h}$, then eqn. (5) 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. (5). Collecting constants yields the following simplified update:

$$v_j^{k+1} = \beta_j v_j^k + \alpha_j g + \text{const.}, \tag{6}$$

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. (6) is that we now have N different momentum timescales. Again these timescales are exactly the eigenvalues

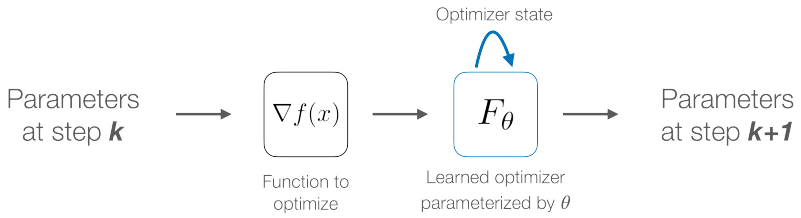

Figure 16: Schematic of a learned optimizer.

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. [31].

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) [45] 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 of the optimizer state, as defined 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 16. 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

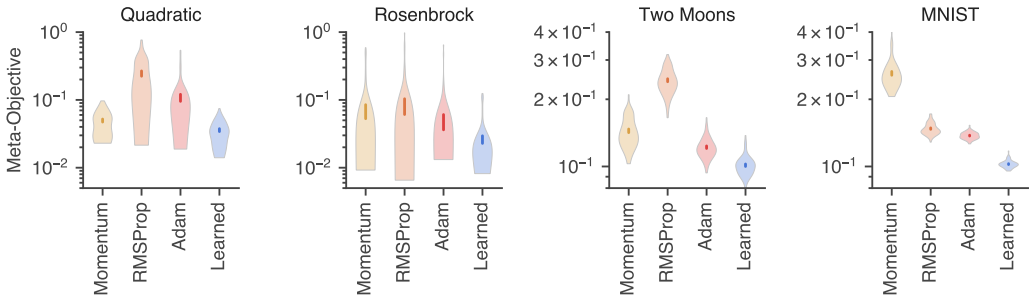

Figure 17: Performance summary. Each panel shows the distribution of the meta-objective over 64 random test problems for baseline and learned optimizers. Dark bars inside of each violin plot indicate the mean and standard error across the 64 random seeds. The learned optimizer has the lowest (best) meta-objective, on average, for each task.

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 [46, 6].

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.

We meta-train our optimizers on a single TPUv2 core. Each model meta-trains in a few hours. Our code is built on top of the scientific python stack, including: NumPy [47], SciPy [48], Matplotlib [49], and JAX [50].

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 17.

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

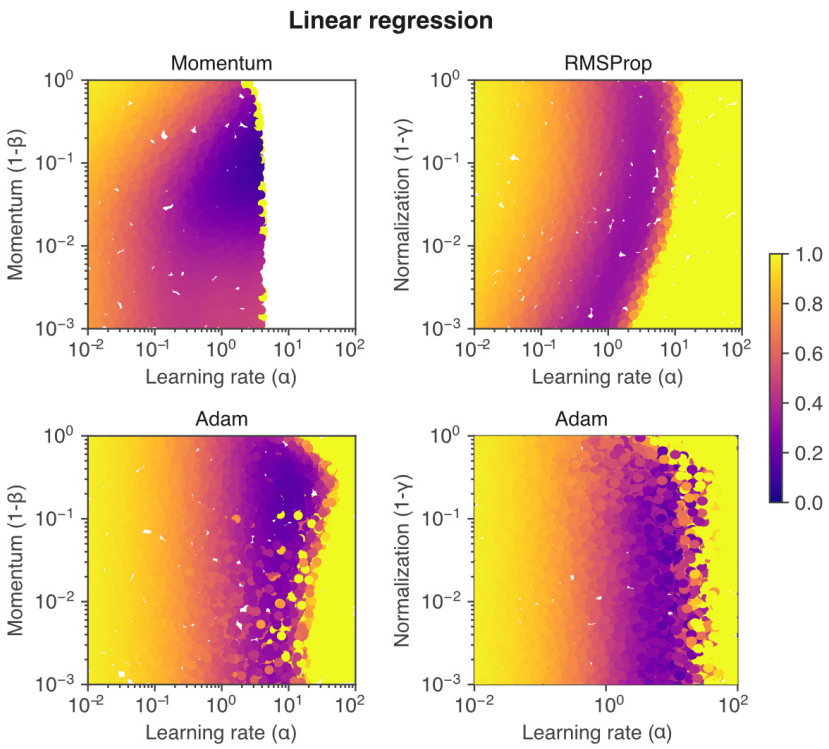

Figure 18: Hyperparameter selection for linear regression.

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 18 (linear regression), 19 (Rosenbrock), 20 (two moons classification), and 21 (MNIST classification). In each of these figures, the color scale is the same — purple denotes the optimal hyperparameters.

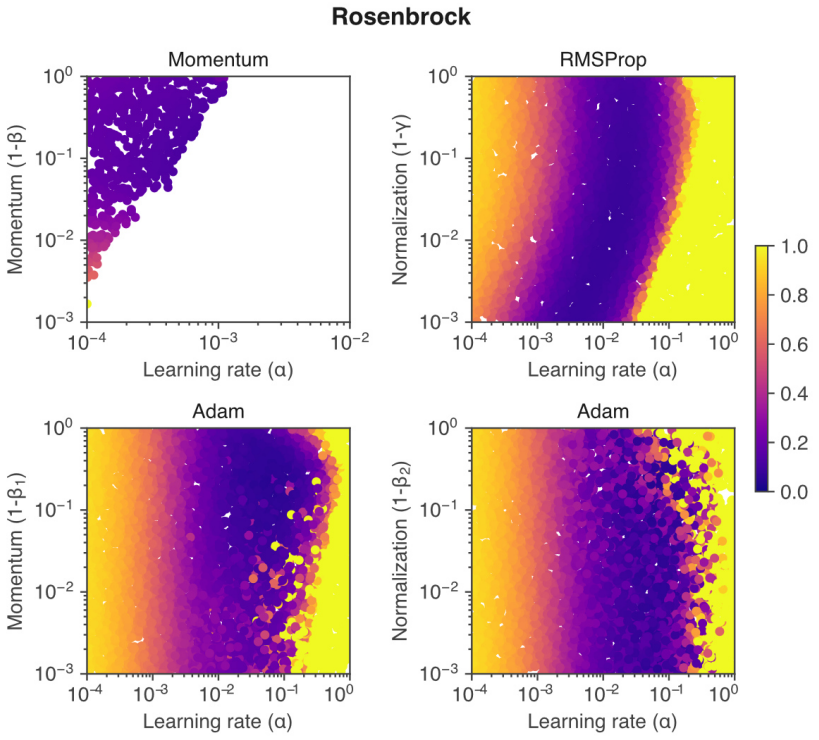

Figure 19: Hyperparameter selection for Rosenbrock.

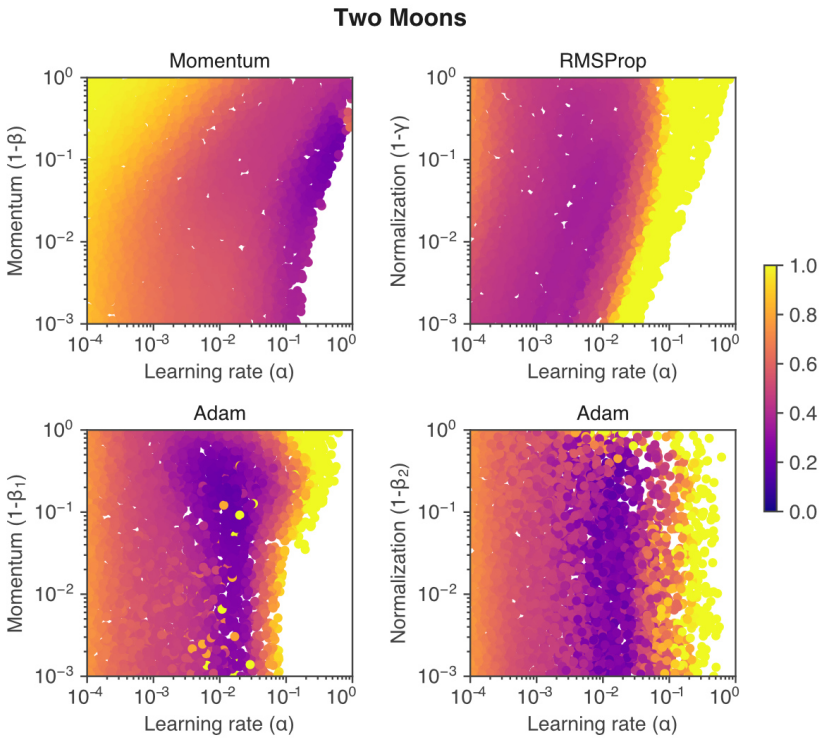

Figure 20: Hyperparameter selection for training a neural network on two moons data.

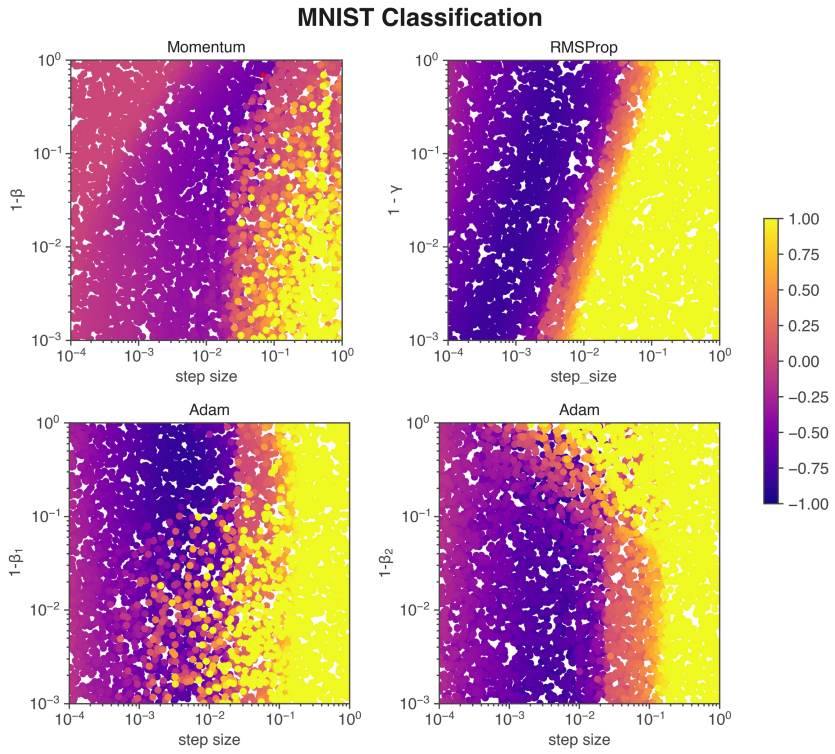

Figure 21: Hyperparameter selection for training a neural network on MNIST classification.