# Reverse engineering learned optimizers reveals known and novel mechanisms

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Learned optimizers are algorithms that are themselves trained to solve optimization problems. In contrast to baseline optimizers (such as momentum or Adam) that use simple update rules derived from theoretical principles, learned optimizers use flexible, high-dimensional, nonlinear parameterizations. Although this can lead to better performance in certain settings, their inner workings remain a mystery. How is a learned optimizer able to outperform a well tuned baseline? Has it learned a sophisticated combination of existing optimization techniques, or is it implementing completely new behavior? In this work, we address these questions by careful analysis and visualization of learned optimizers. We study learned optimizers trained from scratch on three disparate tasks, and discover that they have learned interpretable mechanisms, including: momentum, gradient clipping, learning rate schedules, and a new form of learning rate adaptation. Moreover, we show how the dynamics of learned optimizers enables these behaviors. Our results help elucidate the previously murky understanding of how learned optimizers work, and establish tools for interpreting future learned optimizers.

## 1 Introduction

Optimization algorithms underlie nearly all of modern machine learning. A recent thread of research is focused on learning optimization algorithms, by directly parameterizing and training an optimizer on a distribution of tasks. These so-called *learned optimizers* have been shown to outperform baseline optimizers in restricted settings (Andrychowicz et al., 2016; Wichrowska et al., 2017; Lv et al., 2017; Bello et al., 2017; Li & Malik, 2016; Metz et al., 2019, 2020). Despite improvements in the design, training, and performance of learned optimizers, fundamental questions remain about their behavior. We understand remarkably little about *how* these systems work.

Contrast this with existing "hand-designed" optimizers such as momentum (Polyak, 1964), AdaGrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012), or Adam (Kingma & Ba, 2014). These algorithms are motivated and analyzed via intuitive mechanisms. This understanding allows future studies to build on these techniques by highlighting flaws in their operation (Loshchilov & Hutter, 2018), studying convergence (Reddi et al., 2019), and developing deeper knowledge about why key mechanisms work (Zhang et al., 2020). Without analogous understanding of the inner workings of a learned optimizers, it is incredibly difficult to analyze or synthesize their behavior.

In this work, we develop analysis tools for isolating and elucidating mechanisms in nonlinear, high-dimensional learned optimization algorithms (§3). We then use these methods to study learned optimizers trained on three disparate tasks, showing how learned optimizers utilize both known and novel techniques, including: momentum (§4.1), gradient clipping (§4.2), learning rate schedules (Appendix §A), and a new type of learning rate adaptation (§4.3). Taken together, our work can be seen as part of a new approach to scientifically interpret and understand learned algorithms.

Submitted to the 4th Workshop on Meta-Learning at NeurIPS 2020, Vancouver, Canada. Do not distribute.

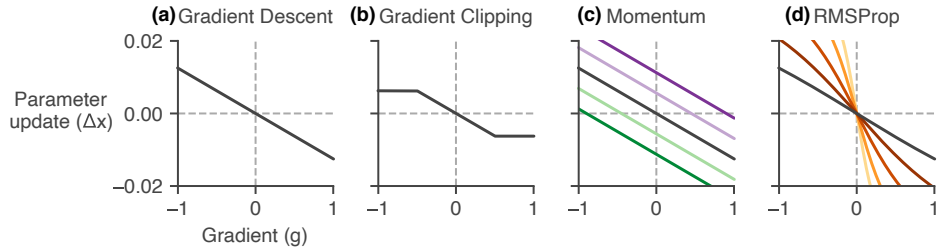


Figure 1: Visualizing optimizer behavior with update functions (see §3.1 for details) for different commonly used optimization techniques. **(a)** Gradient descent is a (stateless) linear function, whose slope is the learning rate. **(b)** Gradient clipping saturates the update, beyond a threshold. **(c)** Momentum introduces a vertical offset depending on the accumulated velocity (colors indicate different values of the accumulated momentum). **(d)** RMSProp changes the slope (effective learning rate) of the update (colors denote changes in the state variable, the accumulated squared gradient).

## 2 Background

We are interested in optimization problems that minimize a loss function ($f$) over parameters ($\boldsymbol{x}$). We focus on first-order optimizers, which at iteration $k$ have access to the gradient $g_i^k \equiv \nabla f(x_i^k)$ and produce an update $\Delta x_i^k$. These are *component-wise* optimizers that are applied to each parameter or component ($x_i$) of the problem in parallel. Standard optimizers used in machine learning (e.g. momentum, Adam) are in this category[1]. Going forward, we use $x$ for the parameter to optimize, $g$ for its gradient, $k$ for the current iteration, and drop the parameter index ($i$) to reduce excess notation.

An optimizer has two parts: the *optimizer state* that stores information about the current problem, and *readout weights* ($\boldsymbol{w}$) that update parameters given the current state. The optimization algorithm is specified by the initial state, the state transition dynamics, and readout, defined as follows:

$$\boldsymbol{h}^{k+1} = F(\boldsymbol{h}^k, g^k) \quad (1)$$

$$x^{k+1} = x^k + \boldsymbol{w}^T \boldsymbol{h}^{k+1}, \quad (2)$$

where $\boldsymbol{h}$ is the optimizer state, $F$ governs the optimizer state dynamics, and $\boldsymbol{w}$ are the readout weights. *Learned optimizers* are constructed by parameterizing the function $F$, and then learning those parameters along with the readout weights through meta-optimization (detailed in Appendix D.2). *Hand-designed* optimization algorithms, by distinction, specify these functions at the outset.

For example, in momentum, the state is a scalar (known as the velocity) that accumulates a weighted average of recent gradients. For momentum and other hand-designed optimizers, the state variables are low-dimensional, and their dynamics are straightforward. In contrast, learned optimizers have high-dimensional state variables, and the potential for rich, nonlinear dynamics.

## 3 Tools for analyzing optimizer behavior

### 3.1 Parameter update function visualizations

Any optimizer, at a particular state, can be viewed as a scalar function that takes in a gradient ($g$) and returns a change in the parameter ($\Delta x$). We refer to this as the optimizer *update function*. Mathematically, the update function is computed as the the state update projected onto the readout, $\Delta x = \boldsymbol{w}^T F(\boldsymbol{h}, g)$, following equations (1) and (2). In addition, the slope of this function can be thought of as the *effective learning rate* at a particular state[2].

It is instructive to visualize these update functions for commonly used optimizers (Figure 1). For gradient descent, the update ($\Delta x = -\alpha g$) is stateless and is always a fixed linear function whose slope is the learning rate, $\alpha$ (Fig. 1a). Gradient clipping is also stateless, but is a saturating function

[1]Notable exceptions include quasi-Newton methods such as L-BFGS (Nocedal & Wright, 2006) or K-FAC (Martens & Grosse, 2015).

[2]We compute this slope at $g = 0$. We find that the update function is always linear in the middle with saturation at the extremes, thus the slope at zero is a good summary of the effective learning rate.

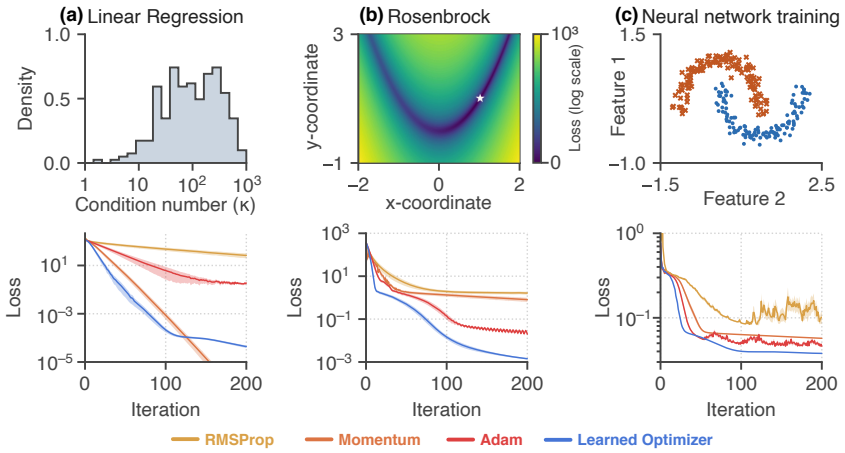


Figure 2: Learned optimizers outperform well tuned baselines on three different tasks: **(a)** linear regression, **(b)** the Rosenbrock function, and **(c)** training a neural network on the two moons dataset. *Upper row*: task schematics (described in §4). *Bottom row*: training curves for different optimizers: momentum (orange), RMSProp (yellow), Adam (red) and a learned optimizer (blue).

of the gradient (Fig. 1b). For momentum, the update is $\Delta x = -\alpha(v + \beta g)$, where $v$ denotes the momentum state (velocity) and $\beta$ is the momentum hyperparameter. The velocity adds an offset to the update function (Fig. 1c). For adaptive optimizers such as RMSProp, the state variable changes the slope, or effective learning rate, within the linear region of the update function (Fig. 1d).

### 3.2 A dynamical systems perspective

In order to understand the dynamics of a learned optimizer, we approximate the nonlinear dynamical system via linearized approximations (Strogatz, 2018). These linear approximations hold near *fixed points* of the dynamics. We can numerically find approximate fixed points (Sussillo & Barak, 2013; Maheswaranathan et al., 2019). One can think of the structure of the fixed points as shaping a dynamical skeleton that governs the optimizer behavior. As we will see, for a well trained optimizer, the dynamics around fixed points enable interesting and useful computations.

## 4 Results

We parametrize the learned optimizer with a recurrent neural network (RNN), similar to Andrychowicz et al. (2016). Specifically, we use a gated recurrent unit (GRU) (Cho et al., 2014) with 256 units. The only input to the optimizer is the gradient. The RNN is is trained by minimizing a *meta-objective*, which we define as the average training loss when optimizing a target problem. See Appendix D.2 for details about the optimizer architecture and meta-training procedures.

We trained these learned optimizers on each of three tasks: randomly generated linear regression problems (which have a distribution of condition numbers shown in Fig. 2a), the Rosenbrock function (Rosenbrock, 1960) (Fig. 2b), and training a small neural network to classify the two moons dataset (Fig. 2c). These tasks were selected because they are fast to train (particularly important for meta-optimization) and covered a range of loss surfaces (convex and non-convex, low- and high-dimensional). See Appendix D.1 for more details about the three tasks.

On each task, we additionally tuned three baseline optimizers (momentum, RMSProp, and Adam). We selected the hyperparameters for each problem out of 2500 samples randomly drawn from a grid. Details about the exact grid ranges used for each task are in Appendix D.3.

Figure 2 compares the performance of the learned optimizer (blue) to baseline optimizers (red, yellow, and orange), on each of the three tasks described above. Across all three tasks, the learned optimizer outperforms the baseline optimizers on the meta-objective[3] (Appendix Fig. 9). We discovered four

[3]As the meta-objective is the average training loss during an optimization run, it naturally penalizes the training curve earlier in training (when loss values are large) compared to later in training. This explains the

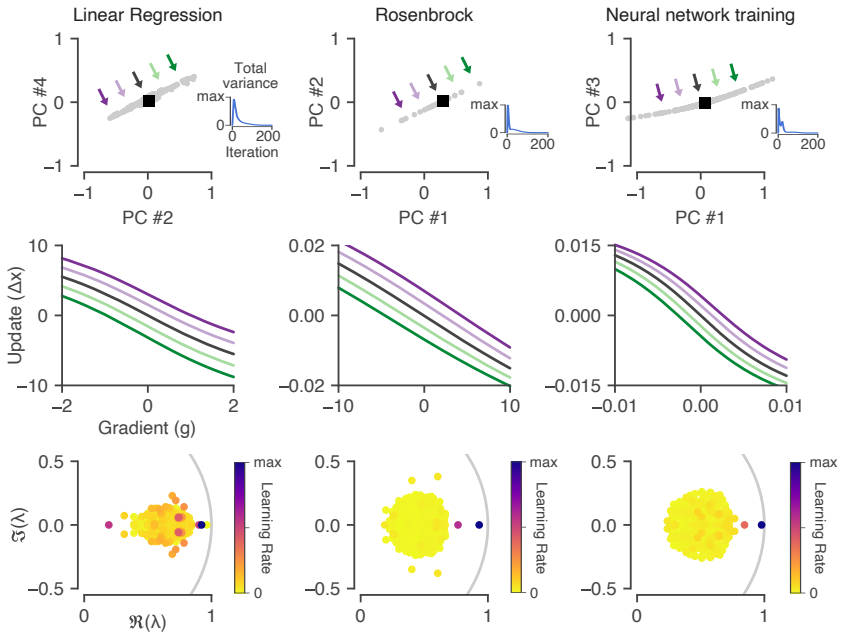


Figure 3: Momentum in learned optimizers. Each column shows the same phenomena, but for optimizers trained on different tasks. **Top row**: Projection of the optimizer state around a convergence point (black square). *Inset:* the total variance of the optimizer states over test problems goes to zero as the trajectories converge. **Middle row**: visualization of the update functions (§3.1) 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. 1c). **Bottom row**: Eigenvalues of the linearized optimizer dynamics at the convergence fixed point (black square in (a)) plotted in the complex plane. The eigenvalue magnitudes are momentum timescales, and the color indicates the corresponding learning rate. See §4.1 for details.

mechanisms in the learned optimizers that seem responsible for their superior performance. In the following sections, we go through each in detail, showing how it is implemented in the learned optimizer. For the behaviors that are task dependent, we highlight how they vary across tasks.

### 4.1 Momentum

We discovered that learned optimizers implement momentum using approximate linear dynamics (Figure 3). First, we found that each optimizer converges to a single global fixed point of the dynamics. We can see this as the total variance of hidden states across test problems goes to zero as the optimizer is run (inset in Fig. 3). The top row of Fig. 3 is a projection[4] of the hidden state space, showing the convergence fixed point (black square). Around this fixed point, the dynamics are organized along a line (gray circles). Shifting the hidden state along this line (indicated by colored arrows) induces a corresponding shift in the update function (middle row of Fig. 3), similar to what is observed in classical momentum (cf. Fig. 1c).

At a fixed point, we can linearly approximate the nonlinear dynamics of the optimizer using the Jacobian of the state update. This Jacobian is a matrix with $N$ eigenvalues and eigenvectors. Writing the update in these coordinates allows us to rewrite the learned optimizer as a momentum algorithm (see Appendix C), albeit with $N$ timescales instead of just one. The magnitude of the eigenvalues

discrepancy in the training curves for linear regression (Fig. 2a, bottom) where momentum continues to decrease the loss for small loss values. Despite this, the learned optimizer has an overall smaller meta-objective due to having lower loss at earlier iterations.

[4]We use principal components analysis (PCA) to project the high-dimensional hidden state into 2D. Depending on the task, we found that different mechanisms would correspond to different principal components (hence the different numbers on the x- and y- axes of the top row of Fig. 3).

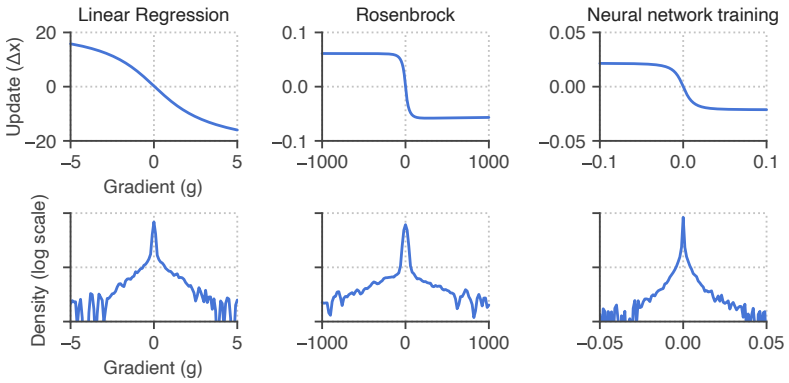


Figure 4: Gradient clipping in a learned optimizer. **Top row**: The update function computed at the initial state saturates for large gradient magnitudes. The effect of this is similar to that of gradient clipping (cf. Fig. 1b). **Bottom row**: the empirical density of encountered gradients for each 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).

are exactly momentum timescales, each with a corresponding learning rate. Note that this type of optimizer has been previously proposed as *aggregated momentum* by Lucas et al. (2018).

We find that learned optimizers use a single mode to implement momentum. The bottom row of Fig. 3 shows the eigenvalues (computed at the convergence fixed point) in the complex plane, colored by that mode's learning rate (see Appendix C for how these quantities are computed). This reveals a single dominant eigenmode (colored in purple), whose eigenvector corresponds to the momentum direction and whose eigenvalue is the corresponding momentum timescale.

While we analyze the best performing learned optimizers in the main text, we did find a learned optimizer on the linear regression task that had slightly worse performance but strongly resembled classical momentum; in fact, this optimizer recovered the optimal momentum parameters for this task. We analyze this optimizer in Appendix B as it is instructive for understanding.

### 4.2 Gradient clipping

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. 1b).

We find that learned optimizers also use saturating update functions as the gradient magnitude increases, thus learning a soft form of gradient clipping (Figure 4). Although we show the saturation for a particular optimize state (the initial state, top row of Fig. 4), 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. 4). For some problems, such as linear regression, the learned optimizer largely stays within the linear region of the update function (Fig. 4, left column). For others, such as the Rosenbrock problem (Fig. 4, right column), the optimizer utilizes more of the saturating part of the update function.

### 4.3 Learning rate adaptation

The final mechanism we discovered is a type of learning rate adaptation. The effect of this mechanism is to decrease the learning rate of the optimizer when large gradients are encountered.

This works by changing the fixed points of the system depending on the current gradient. These *input-dependent* fixed points attract the hidden state away from the final convergence point into different regions of state space; these regions can be used to modify the optimizer.

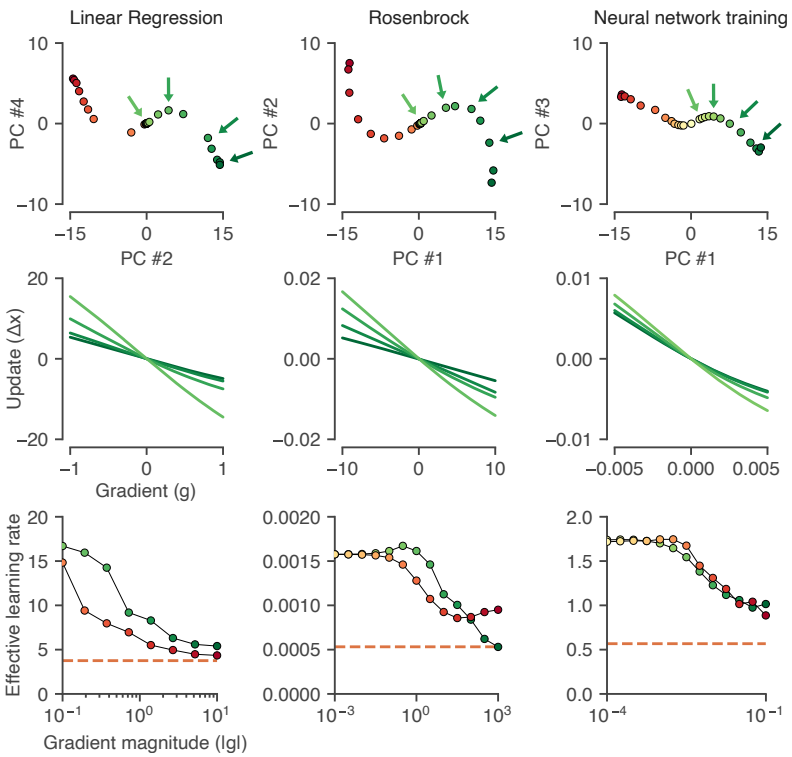


Figure 5: Learning rate adaptation in learned optimizers. **Top row**: Approximate fixed points (colored circles) of the dynamics computed for different gradients reveal an S-curve structure. **Middle row**: Update functions (§3.1) computed at different points along the S-curve (corresponding to arrows 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. 1d). 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.

A picture of these different input driven fixed points is shown as the colored circles in the top row of Figure 5. Across all tasks, we see that these points form an S-curve: one arm of this curve corresponds to negative gradients (red), while the other corresponds to positive gradients (green). The tails of the S-curve correspond to the largest magnitude gradients encountered by the optimizer, and the central spine of the S-curve contains the final convergence point[5].

We see that as we move out to one of the tails of the S-curve (corresponding to large gradients) the slope of the update function becomes more shallow (middle row of Fig. 5), similar to the changes observed as the RMSProp state varies (Fig. 1d).

The changing learning rate along both arms of the S-curve are shown in the bottom row of Fig. 5, for positive (green) and negative (red) gradients, plotted against the magnitude of the gradient on a log scale. This allows the system to increase its learning rate for smaller gradient magnitudes. For context, the best tuned learning rate for classical momentum for each task is shown as a dashed line.

## 5 Discussion

In this work, we trained learned optimizers on three different optimization tasks, and then studied their behavior. We discovered that learned optimizers learn a plethora of intuitive mechanisms: momentum, gradient clipping, schedules, and learning rate adaptation. While the coarse behaviors are qualitatively similar across different tasks, the mechanisms are tuned for particular tasks. Our work shows how to extract insight from the high-dimensional nonlinear dynamics of learned optimizers.

[5]The top row of Figure 5 uses the same projection as the top row of Figure 3, just zoomed out.

# References


Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In Advances in neural information processing systems, pp. 3981–3989, 2016.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. arXiv preprint arXiv:1709.07417, 2017.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.

Dami Choi, Christopher J Shallue, Zachary Nado, Jaehoon Lee, Chris J Maddison, and George E Dahl. On empirical comparisons of optimizers for deep learning. arXiv preprint arXiv:1910.05446, 2019.

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

Rong Ge, Sham M Kakade, Rahul Kidambi, and Praneeth Netrapalli. The step decay schedule: A near optimal, geometrically decaying learning rate procedure for least squares. In Advances in Neural Information Processing Systems, pp. 14977–14988, 2019.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Ke Li and Jitendra Malik. Learning to optimize. arXiv preprint arXiv:1606.01885, 2016.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. arXiv preprint arXiv:1910.07454, 2019.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. arXiv preprint arXiv:1608.03983, 2016.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. 2018.

James Lucas, Shengyang Sun, Richard Zemel, and Roger Grosse. Aggregated momentum: Stability through passive damping. arXiv preprint arXiv:1804.00325, 2018.

Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. arXiv preprint arXiv:1703.03633, 2017.

Niru Maheswaranathan, Alex Williams, Matthew Golub, Surya Ganguli, and David Sussillo. Universality and individuality in neural dynamics across large populations of recurrent networks. In Advances in neural information processing systems, pp. 15629–15641, 2019.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In International conference on machine learning, pp. 2408–2417, 2015.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In International Conference on Machine Learning, pp. 4556–4565, 2019.

Luke Metz, Niru Maheswaranathan, C. Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves, 2020.

Jorge Nocedal and Stephen Wright. Numerical optimization. Springer Science & Business Media, 2006.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In International conference on machine learning, pp. 1310–1318, 2013.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics, 4(5):1–17, 1964.

Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237, 2019.

Herbert Robbins and Sutton Monro. A stochastic approximation method. The annals of mathematical statistics, pp. 400–407, 1951.

HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. The Computer Journal, 3(3):175–184, 1960.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In International Conference on Machine Learning, pp. 343–351, 2013.

Leslie N Smith. Cyclical learning rates for training neural networks. In 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), pp. 464–472. IEEE, 2017.

Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. arXiv preprint arXiv:1711.00489, 2017.

Steven H Strogatz. Nonlinear dynamics and chaos with student solutions manual: With applications to physics, biology, chemistry, and engineering. CRC press, 2018.

David Sussillo and Omri Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. Neural computation, 25(3):626–649, 2013.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2):26–31, 2012.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. arXiv preprint arXiv:1703.04813, 2017.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. arXiv preprint arXiv:1803.02021, 2018.

Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. In International Conference on Learning Representations, 2020. URL https://openreview.net/forum?id=BJgnXpVYwS.

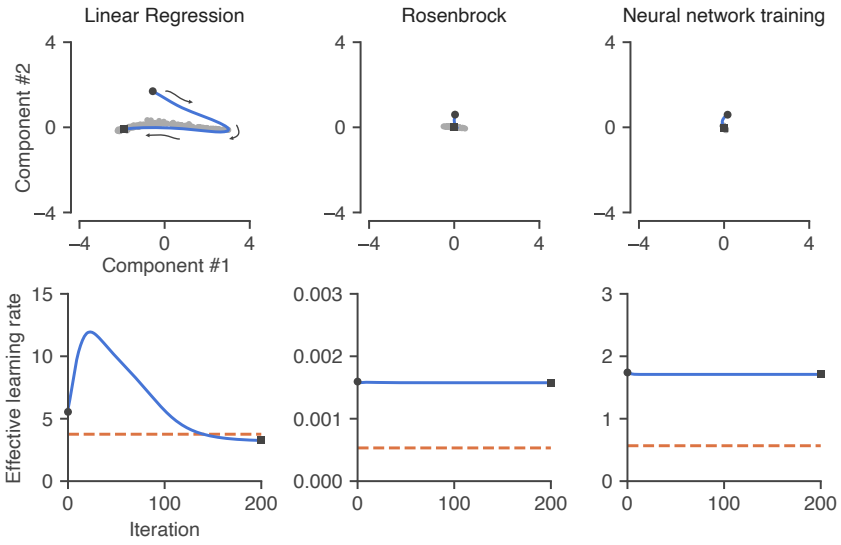


Figure 6: Learning rate schedules mediated by autonomous dynamics. **Top row**: Low-dimensional projection of the dynamics of the learned optimizer in response to zero gradients (no input). These autonomous dynamics allow the system to learn a learning rate schedule (see §A). **Bottom row**: Effective 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 (left column), which includes both a warm-up and decay. For context, dashed lines indicate the best (tuned) learning rate for momentum.

# A Learning rate schedules in learned optimizers

Practitioners often tune a learning rate schedule, that is, a learning rate that varies per iteration. Originally motivated for use with stochastic gradients to guarantee convergence to a fixed point (Robbins & Monro, 1951), schedules are now used more broadly (Schaul et al., 2013; Smith et al., 2017; Ge et al., 2019; Choi et al., 2019). These schedules are often a decaying function of the iteration — meaning the learning rate goes down as optimization progresses — although Goyal et al. (2017) use an additional (increasing) warm-up period, and even more exotic schedules have also been proposed (Loshchilov & Hutter, 2016; Smith, 2017; Li & Arora, 2019).

We discovered that learned optimizers can implement a schedule using *autonomous* — that is, not input driven — dynamics. By moving the initial state (which are trainable parameters) away from the convergence fixed point, then even in the absence of input, autonomous dynamics will encode a particular trajectory as a function of the iteration as the system relaxes to the fixed point.

This autonomous trajectory must additionally avoid updating parameters being optimized[6]. This requirement is satisfied by the autonomous dynamics evolving only in a subspace orthogonal to the readout weights used to update the parameters. Finally, along this autonomous trajectory, the system can then modify its update function to implement a schedule.

For the linear regression task, we found a 2D subspace[7] where the autonomous dynamics occur (Figure 6), driving the system from the initial state (black circle) to the final convergence point (black square). The shaded gray points in the top row of Fig. 6 are slow points of the dynamics (Sussillo & Barak, 2013), which shape the trajectory.

By computing the effective learning rate (slope of the update function) of the system along the autonomous trajectory, we can study the effect of these dynamics. We find that for the linear regression task (left column of Fig. 6), the system has learned to initially increase the learning

[6]Otherwise, the optimizer would modify parameters even if their gradient was zero.

[7]We found this subspace by looking for dimensions that maximized the variation of the autonomous trajectory; this subspace is different from the low-dimensional projection used in Figures 3 and 5.

rate over the course of 25 iterations, followed by a roughly linear decay. We find that the learned optimizer trained on the other tasks does not learn to use a learning rate schedule.

## 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 2a). 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

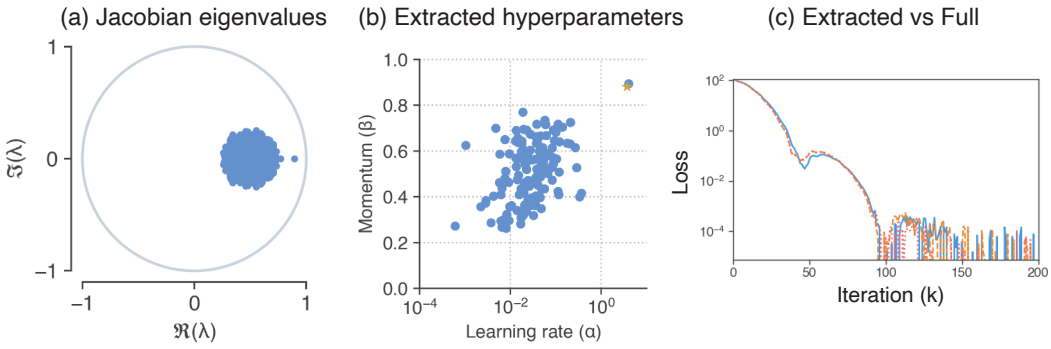


Figure 7: A learned optimizer that recovers momentum on the linear regression task. **(a)** Eigenvalues of the Jacobian of the optimizer dynamics evaluated at the convergence fixed point. There is a single eigenmode that has separated from the bulk. **(b)** Another way of visualizing eigenvalues is by 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.

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.

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 $\beta_j$ to denote the $j^{\text{th}}$ eigenvalue and $\alpha_j$ is the $j^{\text{th}}$ 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 $\alpha_j$. 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

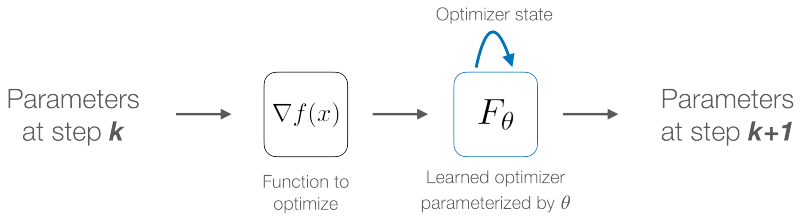


Figure 8: Schematic of a learned optimizer.

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.

**Convex, quadratic:** The first task consists of random linear regression problems. These are generated by sampling random matrices $\boldsymbol{A}$ and $\boldsymbol{b}$, and then building the quadratic loss function: $f(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$. Much of our theoretical understanding of the behavior of optimization algorithms can be derived using quadratic functions, in part because they have a constant Hessian ($\boldsymbol{A}^T\boldsymbol{A}$) over the entire parameter space. The choice of how to sample the problem data $\boldsymbol{A}$ and $\boldsymbol{b}$ will generate a particular distribution of Hessians and condition numbers. The distribution of condition numbers for our task distribution is shown in Figure 2a.

**Non-convex, low-dimensional:** The second task is minimizing the Rosenbrock function (Rosenbrock, 1960), a commonly used test function for optimization. It is a non-convex function which contains a curved valley and a single global minimum. The function is defined over two parameters ($x$ and $y$) as $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$. The distribution of problems for this task consists of the same loss function with different initializations sampled uniformly over a grid. The rosenbrock loss surface is shown in Figure 2b, on a log scale to highlight the curved valley. The grid used to sample initializations is the same as the grid shown in the figure; the x-coordinate is sampled uniformly from (-2, 2) and the y-coordinate is sampled uniformly from (-1, 3).

**Non-convex, high-dimensional:** The third task involves training a neural network to classify a toy dataset, the two moons dataset. This is a 2D dataset, which is advantageous for training learned optimizers when the entire dataset must be kept in memory. The raw dataset is shown in Figure 2c. As the data are not linearly separable, a nonlinear classifier is required to solve the task. The optimization problem is to train the weights of a three hidden layer fully connected neural network, with 64 units per layer and tanh nonlinearities, by minimizing the logistic loss. The distribution of problems involves sampling the initial weights of the network.

### 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 6. 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

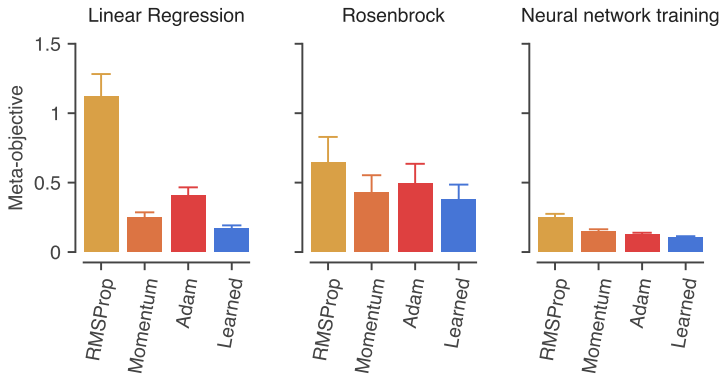


Figure 9: Performance summary. Each panel shows the meta-objective (average training loss) over 64 random test problems for baseline and learned optimizers. Error bars show standard error. The learned optimizer has the lowest (best) meta-objective on each task.

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 $\ell_2$-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

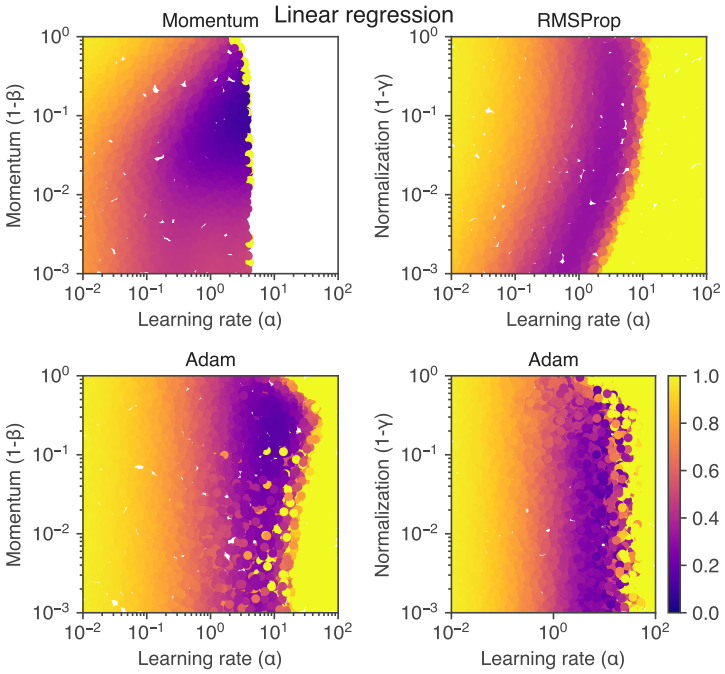


Figure 10: Hyperparameter selection for linear regression.

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 ($\alpha$) and momentum timescale ($\beta$). For RMSProp, we tuned the learning rate ($\alpha$) and learning rate adaptation parameter ($\gamma$). For Adam, we tuned the learning rate ($\alpha$), momentum ($\beta_1$), and learning rate adaptation ($\beta_2$) 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.

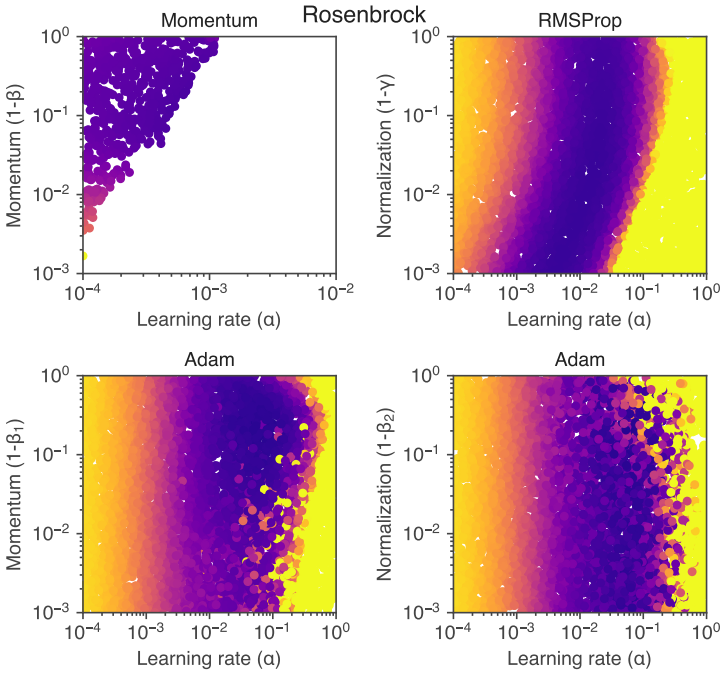


Figure 11: Hyperparameter selection for Rosenbrock.

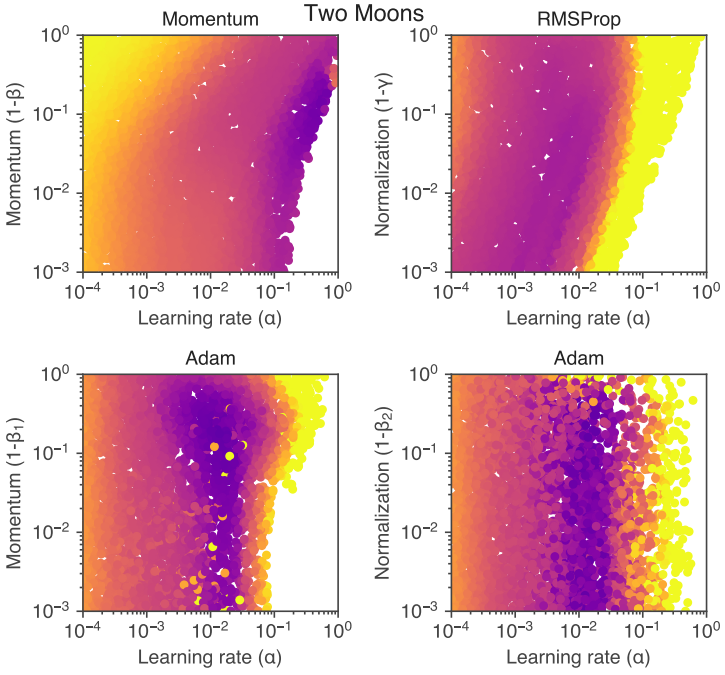


Figure 12: Hyperparameter selection for training a neural network on two moons data.