REVERSE ENGINEERING LEARNED OPTIMIZERS REVEALS KNOWN AND NOVEL MECHANISMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Learned optimizers are algorithms that can themselves be 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. Are learned optimizers simply learning a clever combination of known techniques? Or do they learn fundamentally new behaviors that have not yet been proposed in the optimization literature? If they did learn a new optimization technique, how would we know?

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 and theoretical principles (such as accumulating update velocity in momentum, or rescaling updates based on gradient magnitudes in RMSProp or Adam). This understanding of underlying mechanisms 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. In particular, we demonstrate that learned optimizers learn momentum (§4.1), gradient clipping (§4.2), learning rate schedules (§4.3), and a new type of learning rate adaptation (§4.4). Taken together, our work can be seen as part of a new approach to scientifically interpret and understand learned algorithms. We provide code for training and analyzing learned optimizers, as well as the trained weights for the learned optimizers studied here, at `redacted URL`.

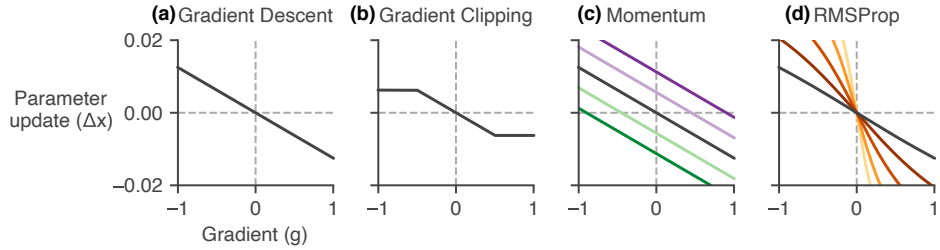

Figure 1: Visualizing optimizer behavior with update functions (see §3.1 for details) for different commonly used optimization techniques. **(a)** Gradient descent is a (stateless) linear function, whose slope is the learning rate. **(b)** Gradient clipping saturates the update, beyond a threshold. **(c)** 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 indicate different values of the state variable, the accumulated squared gradient).

2 Background and related work

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 Δx_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:

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

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

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 C.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.

Our work is heavily inspired by recent work using neural networks to parameterize optimizers. Andrychowicz et al. (2016) originally showed promising results on this front, with additional studies improving robustness (Wichrowska et al., 2017; Lv et al., 2017), meta-training (Metz et al., 2019), and generalization (Metz et al., 2020) of learned optimizers.

We also build on recent work on reverse engineering dynamical systems. Sussillo & Barak (2013) showed how linear approximations to nonlinear dynamical systems can yield insight into the algorithms used by these networks. More recently, these techniques have been applied to understand trained RNNs in a variety of domains, from natural langauge processing (Maheswaranathan et al., 2019a; Maheswaranathan & Sussillo, 2020) to neuroscience (Schaeffer et al., 2020).

Finally, our work draws on a useful vantage point for understanding optimizers, which is to treat them as dynamical systems. This perspective has yielded a number of intuitive and theoretical insights (Su et al., 2014; Wilson et al., 2016; Shi et al., 2019).

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

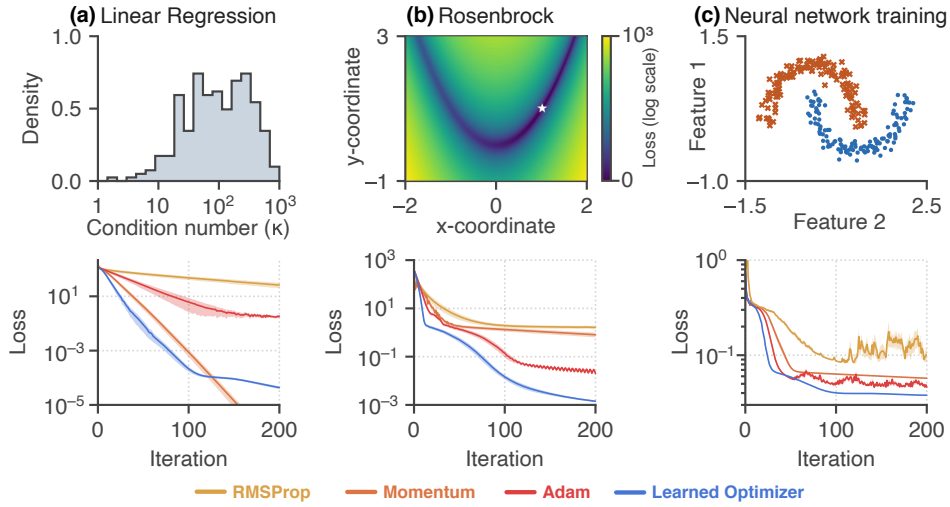

Figure 2: Learned optimizers outperform well tuned baselines on three different tasks: **(a)** linear regression, **(b)** the Rosenbrock function, and **(c)** training a neural network on the two moons dataset. *Upper row*: task schematics (described in §4). *Bottom row*: training curves for different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

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 (Δx). 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, α (Fig. 1a). Gradient clipping is also stateless, but is a saturating function of the gradient (Fig. 1b). For momentum, the update is $\Delta x = -\alpha(v + \beta g)$, where v denotes the momentum state (velocity) and β 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).

Now, what about learned optimizers, or optimizers with much more complicated or high-dimensional state variables? One advantage of update functions is that, as scalar functions, they can be easily visualized and compared to the known methods in Figure 1.

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. Fixed points are points in the state space of the optimizer, where — as long as input gradients do not perturb it — the system does not move. For an optimizer, (stable) fixed points correspond to convergence points[3].

[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.

[3]Technically, in order for a stable fixed point to be a convergence point, the readout evaluated at that state must also be zero, in order to not continuously update the parameters after the state has converged. Empirically we find that this is true, the stable fixed point in our learned optimizers always has near zero readout.

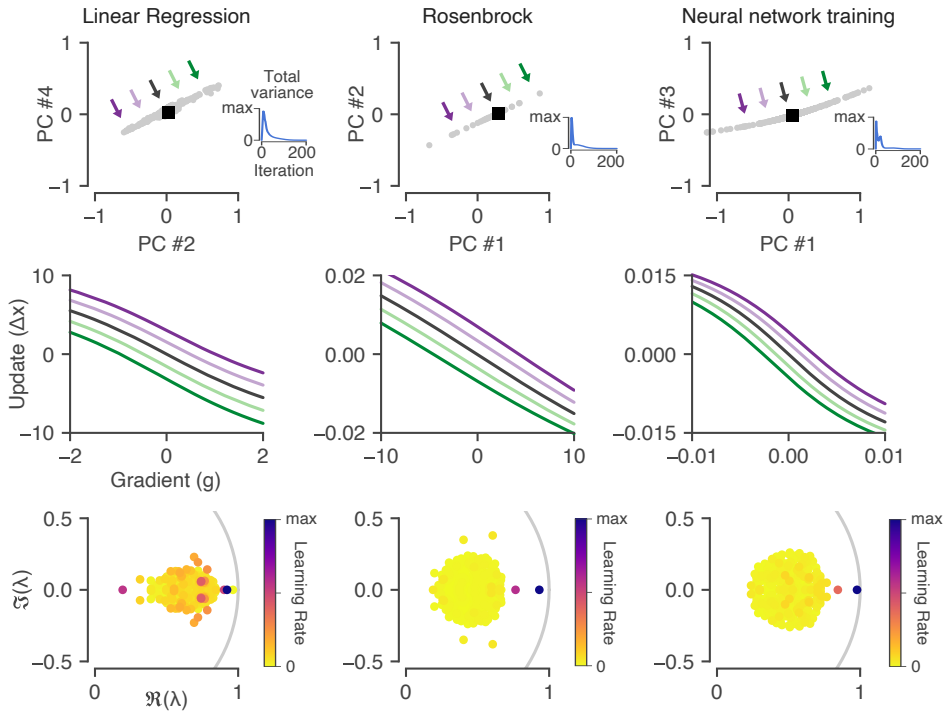

Figure 3: Momentum in learned optimizers. Each column shows the same phenomena, but for optimizers trained on different tasks. **Top row**: Projection of the optimizer state around a convergence point (black square). *Inset:* the total variance of the optimizer states over test problems goes to zero 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.

We can numerically find approximate fixed points (Sussillo & Barak, 2013; Maheswaranathan et al., 2019b). 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 C.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 C.1 for more details about the three tasks.

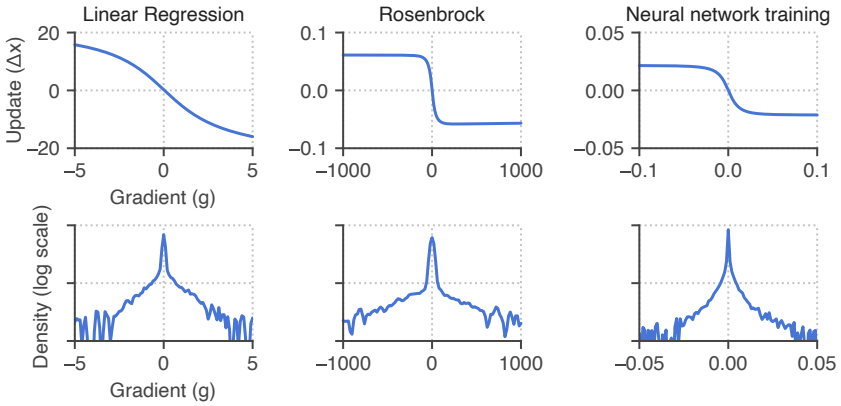

Figure 4: Gradient clipping in a learned optimizer. **Top row**: The update function computed at the initial state saturates for large gradient magnitudes. The effect of this is similar to that of gradient clipping (cf. Fig. 1b). **Bottom row**: the empirical density of encountered gradients for each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

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 C.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[4] (Appendix Fig. 9). We discovered four 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[5] 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 B), albeit with N timescales instead of just one. The magnitude of the eigenvalues 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).

[4]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 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.

[5]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).

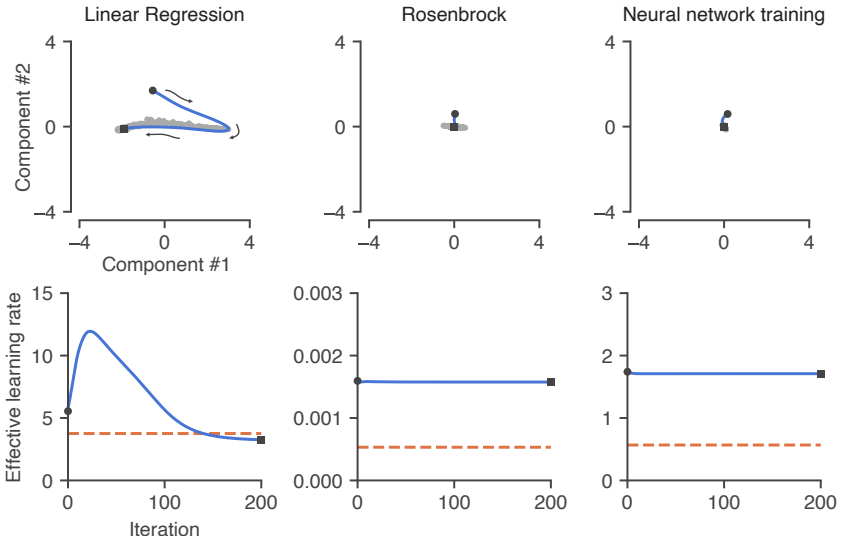

Figure 5: Learning rate schedules mediated by autonomous dynamics. **Top row**: Low-dimensional projection of the dynamics of the learned optimizer in response to zero gradients (no input). These autonomous dynamics allow the system to learn a learning rate schedule (see §4.3). **Bottom row**: Effective 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.

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 B 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 the particular task distribution. We analyze this optimizer in Appendix A as it is instructive for understanding the momentum mechanism.

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 schedules

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

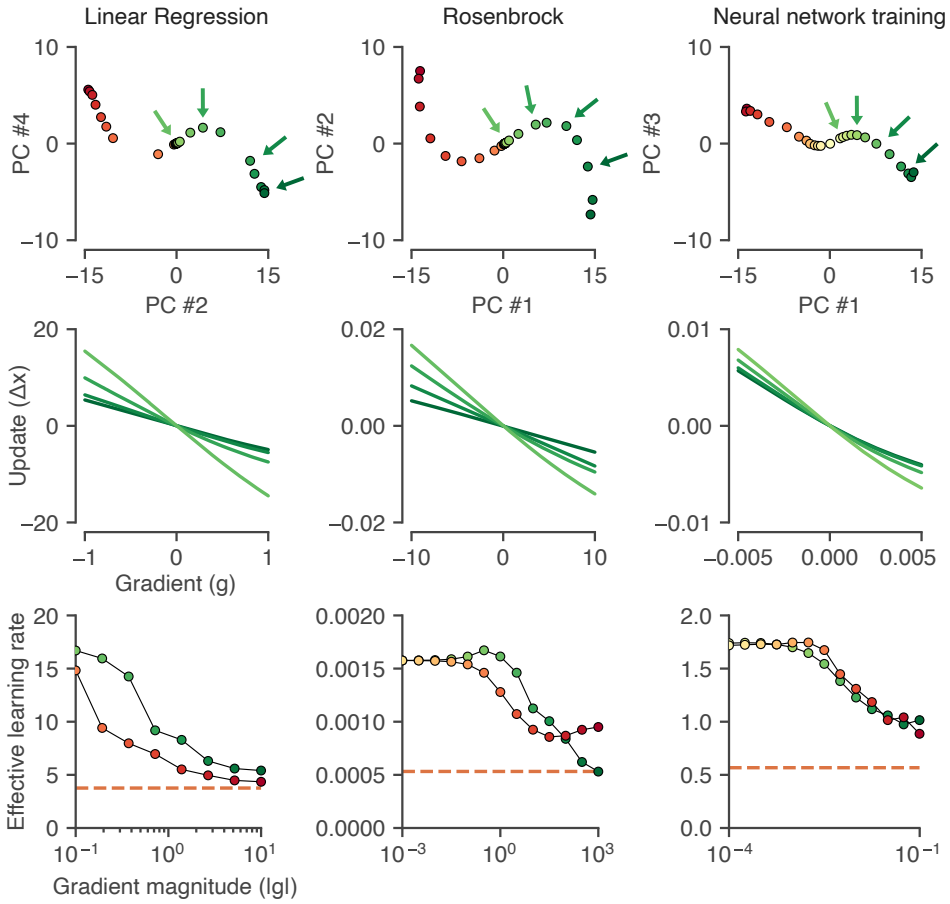

Figure 6: Learning rate adaptation in learned optimizers. **Top row**: Approximate fixed points (colored circles) of the dynamics computed for different gradients reveal an S-curve structure. **Middle row**: Update functions (§3.1) computed at different points along the S-curve (corresponding to arrows 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.

(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.

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

For the linear regression task, we found a 2D subspace[7] where the autonomous dynamics occur (Figure 5), 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. 5 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. 5), the system has learned to initially increase the learning 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.

4.4 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.

A picture of these different input driven fixed points is shown as the colored circles in the top row of Figure 6. 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[8].

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. 6), 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. 6, 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.

While we have isolated individual mechanisms, we still lack a holistic picture of how a learned optimizer stitches these mechanisms together. One may be able to extract or distill a compressed optimizer from these mechanisms, perhaps using data-driven techniques (Brunton et al., 2016; Champion et al., 2019) or symbolic regression (Cranmer et al., 2020).

Finally, we are left wondering *why* the learned optimizers we have studied learned these particular mechanisms. Presumably, they have to do with properties of the tasks used to train the optimizer. What are these quantitative properties (e.g. curvature, convexity, or something else)? Understanding these relationships would allow us to take learned optimizers trained in one setting, and know when and how to apply them to new problems.

Previously, not much was known about how learned optimizers worked. The analysis presented here demonstrates that learned optimizers are capable of learning a number of interesting optimization phenomena. The methods we have developed (update functions and visualization of state dynamics) should be part of a growing toolbox we can use to extract insight from the high-dimensional nonlinear dynamics of learned optimizers, and meta-learned algorithms more generally.

[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 6.

[8] The top row of Figure 6 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.

Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. Proceedings of the national academy of sciences, 113(15):3932–3937, 2016.

Kathleen Champion, Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of coordinates and governing equations. Proceedings of the National Academy of Sciences, 116 (45):22445–22451, 2019.

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.

Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. arXiv preprint arXiv:2006.11287, 2020.

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 and David Sussillo. How recurrent networks implement contextual processing in sentiment analysis. arXiv preprint arXiv:2004.08013, 2020.

Niru Maheswaranathan, Alex Williams, Matthew Golub, Surya Ganguli, and David Sussillo. Reverse engineering recurrent networks for sentiment classification reveals line attractor dynamics. In Advances in Neural Information Processing Systems, pp. 15696–15705, 2019a.

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, 2019b.

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.

Rylan Schaeffer, Mikail Khona, Leenoy Meshulam, Ila Rani Fiete, et al. Reverse-engineering recurrent neural network solutions to a hierarchical inference task for mice. bioRxiv, 2020.

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

Bin Shi, Simon S Du, Weijie Su, and Michael I Jordan. Acceleration via symplectic discretization of high-resolution differential equations. In Advances in Neural Information Processing Systems, pp. 5744–5752, 2019.

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.

Weijie Su, Stephen Boyd, and Emmanuel Candes. A differential equation for modeling nesterov's accelerated gradient method: Theory and insights. In Advances in neural information processing systems, pp. 2510–2518, 2014.

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.

Ashia C Wilson, Benjamin Recht, and Michael I Jordan. A lyapunov analysis of momentum methods in optimization. arXiv preprint arXiv:1611.02635, 2016.

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.