Learned optimizers that outperform SGD on wall-clock and test loss

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, Jascha Sohl-dickstein
Google Brain
{lmetz,nirum,jeremynixon,cdfreeman,jaschasd}@google.com

Abstract

Deep learning has shown that learned functions can dramatically outperform hand-designed functions on perceptual tasks. Analogously, this suggests that learned optimizers may similarly outperform current hand-designed optimizers, especially for specific problems. However, learned optimizers are notoriously difficult to train and have yet to demonstrate wall-clock speedups over hand-designed optimizers, and thus are rarely used in practice. Typically, learned optimizers are trained by truncated backpropagation through an unrolled optimization process. The resulting gradients are either strongly biased (for short truncations) or have exploding norm (for long truncations). In this work we propose a training scheme which overcomes both of these difficulties, by dynamically weighting two unbiased gradient estimators for a variational loss on optimizer performance. This allows us to train neural networks to perform optimization faster than well tuned first-order methods. Moreover, by training the optimizer against validation loss (as opposed to training loss), we are able to learn optimizers that train networks to better generalization than first order methods. We demonstrate these results on problems where our learned optimizer trains convolutional networks in a fifth of the wall-clock time compared to tuned first-order methods, and with an improvement in validation loss.

1 Introduction

Gradient based optimization is a cornerstone of modern machine learning. Improvements in optimization have been critical to recent successes on a wide variety of problems. In practice, this typically involves analysis and development of hand-designed optimization algorithms [1, 2, 3, 4]. These algorithms generally work well on a wide variety of tasks, and are tuned to specific problems via hyperparameter search. On the other hand, a complementary approach is to *learn* the optimization algorithm [5, 6, 7, 8, 9, 10, 11]. That is, to learn a function to perform optimization, targeted to particular problems of interest. In this way, the algorithm may learn task specific structure, enabling dramatic performance improvements over more general optimizers.

However, training learned optimizers is notoriously difficult. Existing work in this vein can be classified into two broad categories. On one hand are black-box methods such as evolution [12, 13], random search [14], reinforcement learning [11, 15, 10], or Bayesian optimization [16]. However, these methods scale poorly with the number of optimizer parameters. The other approach is to use first-order methods, by computing the gradient of some measure of optimizer effectiveness with respect to the optimizer parameters. Computing these gradients is costly as we need to iteratively apply the learned update rule, and then backpropagate through these applications, a technique commonly referred to as "unrolled optimization" [17, 18]. To address the problem of backpropagation through many optimization steps (analogous to many timesteps in recurrent neural networks), many works make use of truncated backpropagation though time (TBPTT) to partition the long unrolled computational graph into separate pieces [19, 20]. This yields computational savings, but at the

cost of increased bias [20] and/or exploding gradients due to many iterated update steps [21, 22]. Existing methods address the bias at the cost of increased variance or computational complexity [23, 24, 20]. Previous techniques for training RNNs via TBPTT have thus far not been effective for training optimizers.

In Appendix A we analytically and experimentally explore the debilitating role of bias and exploding gradients on training optimizers. We then show how these pathologies can be remedied by optimizing the parameters of a distribution over the optimizer parameters, known as variational optimization [25] (§2). We define two unbiased gradient estimators for this objective: a reparameterization based gradient, and evolutionary strategies [26, 27]. By dynamically reweighting the contribution of these two gradient estimators, we are able to avoid strongly biased or exploding gradients, and thus stably and efficiently train learned optimizers.

We demonstrate the utility of this approach by training a learned optimizer to target optimization of convolutional networks on image classification (§3). On the targeted task distribution, this learned optimizer achieves better test loss, and is five times faster in *wall-clock time*, compared to well tuned hand-designed optimizers such as SGD+Momentum, RMSProp, and ADAM (Figure 1). While not the explicit focus of this work, we also find that the learned optimizer demonstrates promising generalization ability on out of distribution tasks (Figure 3).
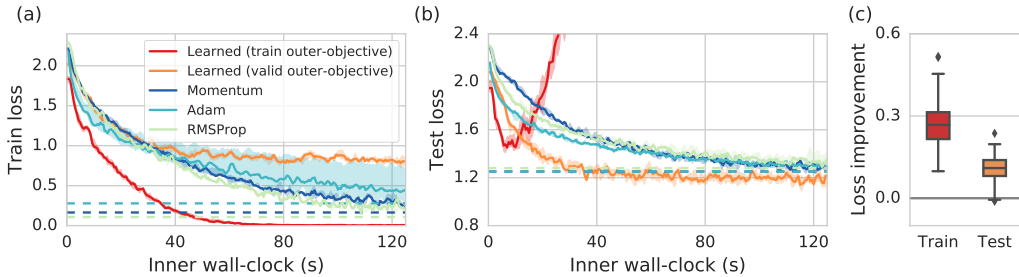

Figure 1: Learned optimizers outperform existing optimizers on training loss (a) and test loss (b). **(a,b)** Training and test curves for a three layer CNN, trained on a subset of 32x32 imagenet classes not seen during outer-training of the optimizer. Dashed lines indicate the best achieved performance over an additional 130 seconds. We show two learned optimizers – one trained to minimize training loss, and the other trained to minimize validation loss on the inner-problem. We compare against Adam, RMSProp, and SGD+Momentum, individually tuned for the train and test loss (Panel (a) and (b), respectively). On training loss (a), our learned optimizer approaches zero training loss, and achieves it's smallest loss values in less than one quarter the wall-clock time. On test loss (b), our learned optimizer achieves a lower minimum, in roughly one third the wall-clock time. Shaded regions correspond to 25 and 75 percentile over five random initializations of the CNN. For plots showing performance in terms of step count rather than wall clock (where we achieve even more dramatic speedups), and for more task instances, see Appendix E. **(c)** Distribution of the performance difference between the learned optimizers and a tuned baseline of either Adam, RMSProp, or Momentum (loss improvement). Positive values indicate performance better than baseline. We show training and test losses for the outer-testing task distribution. On the majority of tasks, the learned optimizers outperform the baseline.

2 Towards stable training of learned optimizers

Our goal is to learn an optimizer which is well suited to some set of target optimization tasks. Learning an optimizer can be thought of as a bi-level optimization problem [28], with *inner* and *outer* levels. The inner minimization consists of optimizing of the weights of a target problem by the repeated application of a learned update rule. The update rule is a parameterized function that defines how to map the weights at iteration t to iteration $t + 1$. In the outer loop, these optimizer parameters are updated so as to minimize some measure of optimizer performance, the outer-objective. For a more detailed problem setup, see Appendix A.

To perform outer-training, we employ gradient descent through this unrolled optimization. Computing outer-gradients through unrolled optimization procedures has been shown to produce poorly conditioned loss surfaces [18]. We expand on this in Appendix A. To optimize in such a space one
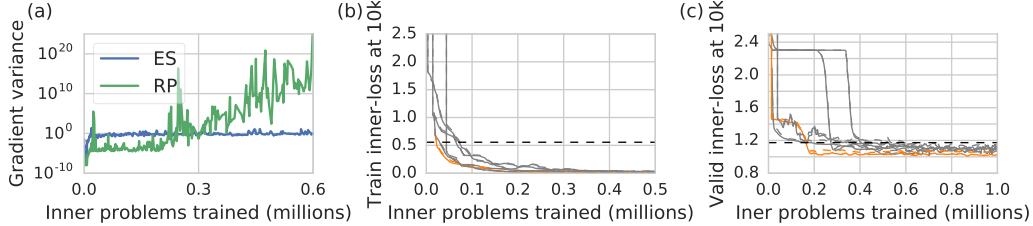
Figure 2: **(a)** As outer-training progresses, the variance of the reparameterization gradient estimator grows, while the variance of the ES estimator remains constant. **(b,c)** Performance after 10k iterations of inner-training for different outer-parameters over the course of outer-training. Each line represents a different random initialization of outer-parameters. Models are trained targeting the train outer-objective (b), and the validation outer-objective (c). Dashed lines indicate performance of hyperparameter tuned Adam. Model's in orange are the best performing and used in §3.

might intuitively want to smooth the outer-objective loss surface. To do this, instead of optimizing the outer-objective directly, we instead optimize a smoothed outer-loss: $\mathbb{E}_{\tilde{\theta} \sim \mathcal{N}(\theta, \sigma^2 I)} \left[L\left(\tilde{\theta}\right) \right]$, where $L\left(\cdot\right)$ is the outer-objective and σ^2 is a fixed variance (set to 0.01) which determines the degree of smoothing. This is the same approach taken in variational optimization [25]. We can construct two different unbiased gradient estimators for $\mathcal{L}\left(\theta\right)$: one via the reparameterization trick [29], and one via the log-derivative trick similar to what is done in evolutionary strategies (ES) [30]. Following the insight from [22] in the context of reinforcement learning, we combine these estimates using inverse variance weighting [31]. When outer-training learned optimizes we find the variances of these estimators can differ by as many as 20 orders of magnitude (Figure 2a). This merged estimator addresses this by having at most the lowest of the two variances.

Finally, an increasing curriculum over steps per truncation is used over the course of outer-training. This introduces bias early in training, but also allows for far more frequent outer-weight updates, resulting in much faster outer-training in terms of wall-clock time. The full outer-training algorithm is described in Appendix C.

3 Experiments

3.1 Learned Optimizer Setup

The optimizer architecture used in all experiments consists of a simple fully connected neural network, with one hidden layer containing 32 ReLU units applied to each target problem weight independently. The problem that each learned optimizer is trained against consists of training a three layer convolutional neural network (32 units per layer) inner-trained for ten thousand inner-iterations on 32x32x3 image classification tasks. We split the Imagenet dataset [32] by class into 700 training and 300 validation classes, and sample training and validation problems by sampling 10 classes. See Appendix D for further details.

3.2 Outer-training

Figure 2 shows the performance of the optimizer (averaged over 40 randomly sampled outer-train and outer-test inner-problems) while outer-training. Despite the stability improvements described in the last section, there is still variability in optimizer performance over random initializations of the optimizer parameters. We use training loss to select the best model and use this in the remainder of the evaluation.

3.3 Learned optimizer performance

Figure 1 shows performance of the learned optimizer, after outer-training, compared against other first-order methods on a sampled validation task (classes not seen during outer-training). For all first-order methods, we report the best performance after tuning the learning rate by grid search using 11 values over a logarithmically spaced range from 10^{-4} to 10. When outer-trained against
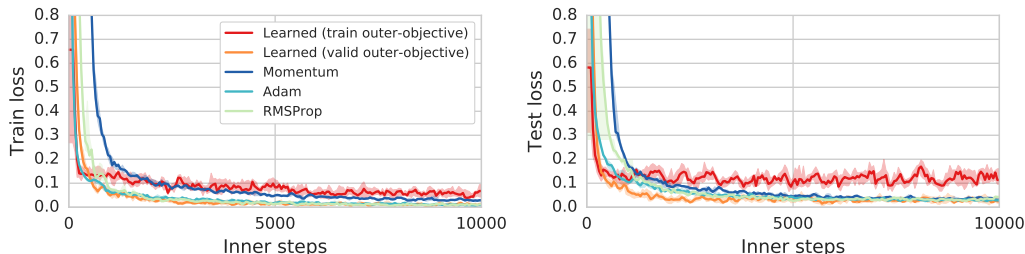
3

Figure 3: Despite outer-training only targeting optimizer performance on a three layer convolutional network trained on 10 way classification problems derived from 32x32 Imagenet, the learned optimizers demonstrate some ability to generalize to new architectures and datasets. Plot shows train and test performance on a six layer convolutional neural network trained on 28x28 MNIST. We compare against hyperparameter tuned Adam, RMSProp, and SGD+Momentum. Note that the optimizer trained to target validation loss generalizes better than the one trained to target train loss.

the training outer-objective, L_{train}, we achieve faster convergence on training loss by a factor of 5x (Figure 1a), but poor performance on test loss (Figure 1b). However, when outer-trained against the validation outer-objective, L_{valid}, we also achieve faster optimization *and* reach a lower test loss (Figure 1b).

Figure 1 summarizes the performance of the learned optimizer across many sampled validation tasks. It shows the difference in final test loss between the *best* first-order method and the learned optimizer. We choose the best first-order method by selecting the best test performance over RMSProp, SGD+Momentum, and Adam. This learned optimizer (which does not require tuning on the validation tasks) outperforms the best baseline optimizer 98% of the time.

Although the focus of our approach was not generalization, we find that our learned optimizer nonetheless generalizes to varying degrees to dissimilar datasets, different numbers of units per layer, different number of layers, and even to fully connected networks. In Figure 3 we show performance on a six layer convolutional neural network trained on MNIST. Despite the different number of layers, different dataset, and different input size, the learned optimizers still reduces the loss, and in the case of the validation outer-objective trains faster to a lower test loss.

4   Discussion

In this work we demonstrate two difficulties when training learned optimizers: "exploding" gradients, and a bias introduced by truncated backpropagation through time. To combat this, we construct a variational bound of the outer-objective and minimize this via a combination of reparameterization and ES style gradient estimators. By using our combined estimator and a curriculum over truncation step we are able to train learned optimizers that achieve more than five times speedup on wallclock time as compared to existing optimizers. In this work, we focused on applying optimizers to a restricted family of tasks. While useful on its own right (e.g. rapid retraining of models on new data), future work will explore the limits of "no free lunch" [33] to understand how and when learned optimizers generalize across tasks.

References

[1] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[3] 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.

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

[5] Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Université de Montréal, Département d'informatique et de recherche opérationnelle, 1990.

[6] Juergen Schmidhuber. On learning how to learn learning strategies. 1995.

[7] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[8] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[9] 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. *International Conference on Machine Learning*, 2017.

[10] Ke Li and Jitendra Malik. Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*, 2017.

[11] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc Le. Neural optimizer search with reinforcement learning. 2017.

[12] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[13] Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. On the optimization of a synaptic learning rule. In *Preprints Conf. Optimality in Artificial and Biological Neural Networks*, pages 6–8. Univ. of Texas, 1992.

[14] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[15] Ke Li and Jitendra Malik. Learning to optimize. *International Conference on Learning Representations*, 2017.

[16] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[17] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

[18] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[19] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[20] Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*, 2017.

[21] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[22] Paavo Parmas, Carl Edward Rasmussen, Jan Peters, and Kenji Doya. Pipps: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*, pages 4062–4071, 2018.

[23] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[24] Yann Ollivier, Corentin Tallec, and Guillaume Charpiat. Training recurrent networks online without backtracking. *arXiv preprint arXiv:1507.07680*, 2015.

[25] Joe Staines and David Barber. Variational optimization. *arXiv preprint arXiv:1212.4507*, 2012.

[26] Ingo Rechenberg. Evolutionsstrategie–optimierung technisher systeme nach prinzipien der biologischen evolution. 1973.

[27] Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. Technical report, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), 2011.

[28] Luca Franceschi, Paolo Frasconi, Saverio Salzo, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. *arXiv preprint arXiv:1806.04910*, 2018.

[29] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[30] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

[31] JL Fleiss. Review papers: The statistical basis of meta-analysis. *Statistical methods in medical research*, 2(2):121–145, 1993.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[33] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[34] Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger B Grosse. Understanding short-horizon bias in stochastic meta-optimization. pages 478–487, 2016.

[35] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[36] James Lucas, Richard Zemel, and Roger Grosse. Aggregated momentum: Stability through passive damping. *arXiv preprint arXiv:1804.00325*, 2018.

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

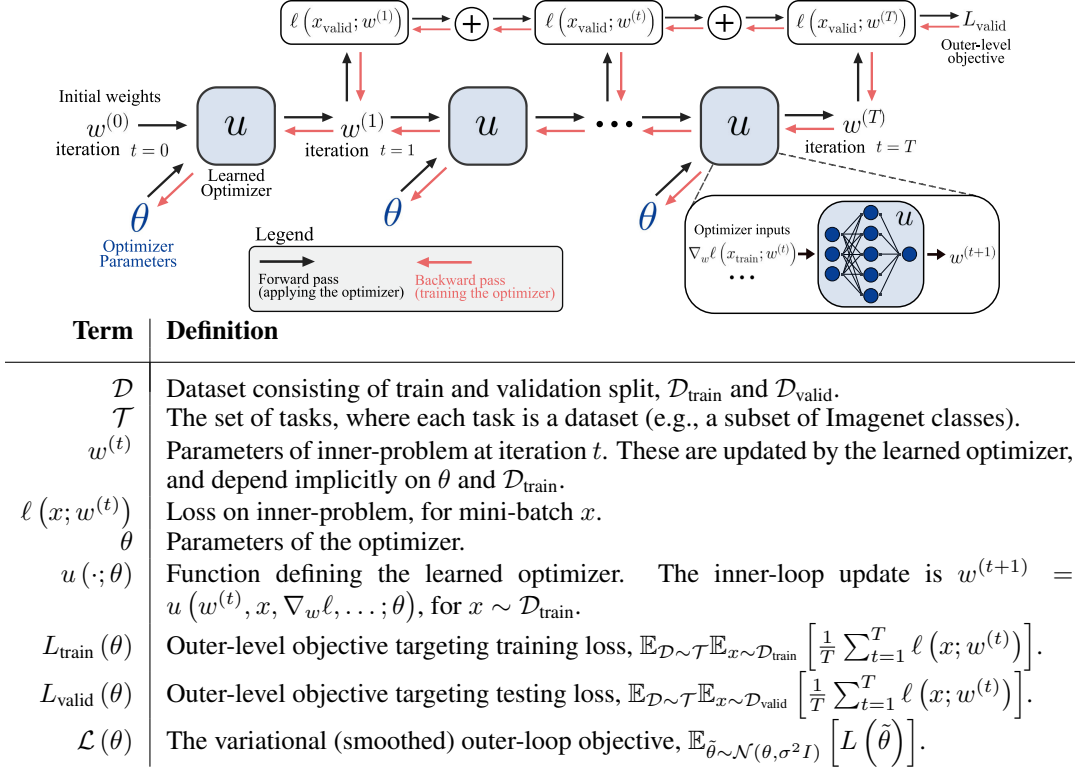[38] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

Figure 4: **Top:** Schematic of unrolled optimization. **Bottom** Definition of terms used in this paper.

Term	Definition
$\mathcal{D}$	Dataset consisting of train and validation split, $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$.
$\mathcal{T}$	The set of tasks, where each task is a dataset (e.g., a subset of Imagenet classes).
$w^{(t)}$	Parameters of inner-problem at iteration t. These are updated by the learned optimizer, and depend implicitly on θ and $\mathcal{D}_{\text{train}}$.
$\ell\left(x; w^{(t)}\right)$	Loss on inner-problem, for mini-batch x.
θ	Parameters of the optimizer.
$u\left(\cdot; \theta\right)$	Function defining the learned optimizer. The inner-loop update is $w^{(t+1)} = u\left(w^{(t)}, x, \nabla_w \ell, \ldots; \theta\right)$, for $x \sim \mathcal{D}_{\text{train}}$.
$L_{\text{train}}\left(\theta\right)$	Outer-level objective targeting training loss, $\mathbb{E}_{\mathcal{D} \sim \mathcal{T}} \mathbb{E}_{x \sim \mathcal{D}_{\text{train}}} \left[\frac{1}{T} \sum_{t=1}^{T} \ell\left(x; w^{(t)}\right)\right]$.
$L_{\text{valid}}\left(\theta\right)$	Outer-level objective targeting testing loss, $\mathbb{E}_{\mathcal{D} \sim \mathcal{T}} \mathbb{E}_{x \sim \mathcal{D}_{\text{valid}}} \left[\frac{1}{T} \sum_{t=1}^{T} \ell\left(x; w^{(t)}\right)\right]$.
$\mathcal{L}\left(\theta\right)$	The variational (smoothed) outer-loop objective, $\mathbb{E}_{\tilde{\theta} \sim \mathcal{N}(\theta, \sigma^2 I)} \left[L\left(\tilde{\theta}\right)\right]$.

A Unrolled optimization for learning optimizers

A.1 Problem Framework

Our goal is to learn an optimizer which is well suited to some set of target optimization tasks. Throughout the paper, we will use the notation defined in Figure 4. Learning an optimizer can be thought of as a bi-level optimization problem [28], with *inner* and *outer* levels. The inner minimization consists of optimizing of the weights (w) of a target problem $\ell(w)$ by the repeated application of an update rule ($u\left(\cdot\right)$). The update rule is a parameterized function that defines how to map the weights at iteration t to iteration $t+1$: $w^{(t+1)} = u(w^{(t)}; \theta)$. Here, θ represents the parameters of the learned optimizer. In the outer loop, these optimizer parameters (θ) are updated so as to minimize some measure of optimizer performance, the outer-objective ($L(\theta)$). Our choice for L will be the average value of the target loss ($\ell\left(\cdot\right)$) measured over either training or validation data. Throughout the paper, we use *inner-* and *outer-* prefixes to make it clear when we are referring to applying a learned optimizer on a target problem (inner) versus training a learned optimizer (outer).

A.2 Unrolled optimization

In order to train an optimizer, we wish to compute derivatives of the outer-objective L with respect to the optimizer parameters, θ. Doing this requires unrolling the optimization process. That is, we can form an unrolled computational graph that consists of iteratively applying an optimizer (u) to optimize the weights (w) of a target problem (Figure 4). Computing gradients for the optimizer parameters involves backpropagating the outer loss through this unrolled computational graph. This is a costly operation, as the entire inner-optimization problem must be unrolled in order to get a single outer-gradient. Partitioning the unrolled computation into separate segments, known as truncated backpropagation, allows one to compute multiple outer-gradients over shorter segments. That is, rather than compute the full gradient from iteration $t = 0$ to $t = T$, we compute gradients in windows from $t = a$ to $t = a + \tau$. The choice for the number of inner-steps per truncation is challenging. Using a large number of steps per truncation can result in exploding gradients making outer-training difficult, while using a small number of steps can produce biased gradients resulting in poor performance. In the following sections we analyze these two problems.
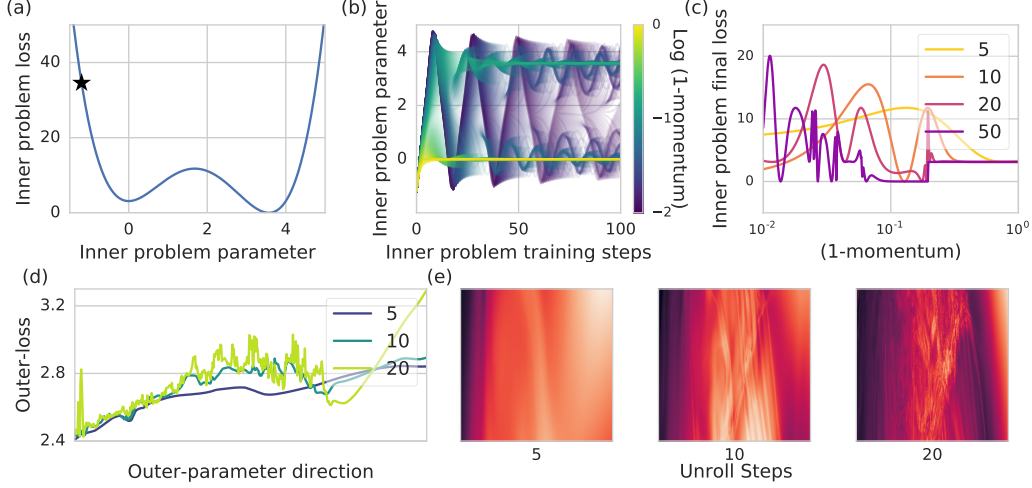
Figure 5: Outer-problem optimization landscapes can become increasingly pathological with increasing inner-problem step count. **(a)** A toy 1D inner-problem loss surface with two local minimum. Initial parameter value $(w^{(0)})$ is indicated by the star. **(b)** Final inner-parameter value $(w^{(T)})$ as a function of the number of inner problem training steps T, when inner-problem training is performed by SGD+momentum. Color denotes different values of the optimizer's momentum parameter. Low momentum values (yellows) converge to the first local minimum at $w = 0$. Slightly higher momentum values (greens) escape this minimum to settle at the global minimum ($w \approx .5$). Even larger values (purples) oscillate before eventually settling in one of the two minimum. **(c)** The final loss after some number steps of optimization as a function of the momentum. The final loss surface is smooth for small number of training steps T. However, larger values of T result in near discontinuous loss surfaces around the transition points between the 2 minimum. **(d)** Similar to **c**, where the inner problem is a MLP, the learned optimizer is the one used this paper, and for a 1D slice through the outer-parameters θ along the gradient direction. **(e)** 2D rather than 1D slices through θ, for different numbers of inner-loop steps. Intensity indicates value of $L_{\text{train}}(\theta)$; darker is lower. Similar pathologies are observed to those which manifest in the toy problem.

A.3 Exponential explosion of gradients with increased sequence length

We can illustrate the problem of exploding gradients analytically with a simple example: learning a learning rate. Following the notation in Figure 4, we define the optimizer as

$$w^{(t+1)} = u(w^{(t)}; \theta) = w^{(t)} - \theta \nabla \ell \left(w^{(t)} \right),$$

where θ is a scalar learning rate that we wish to learn for minimizing some target problem $\ell(w^{(t)})$. For simplicity, we assume a deterministic loss $(\ell(\cdot))$ with no batch of data (x).

The quantity we are interested in is the derivative of the loss after T steps of gradient descent with respect to θ. We can compute this gradient (see Appendix B) as:

$$\frac{d\ell(w^{(T)})}{d\theta} = \left\langle -g^{(T)}, \sum_{i=0}^{T-1} \left(\prod_{j=i+1}^{T-1} (I - \theta H^{(j)}) \right) g^{(i)} \right\rangle,$$

where $g^{(i)}$ and $H^{(j)}$ are the gradient and Hessian of the target problem $\ell(w)$ at iteration i and j, respectively. We see that this equation involves a sum of products of Hessians. In particular, the first term in the sum involves a product over the *entire sequence* of Hessians observed during training. That is, the outer-gradient becomes a matrix polynomial of degree T, where T is the number of gradient descent steps. Thus, the outer-gradient grows exponentially with T.

We can see another problem with long unrolled gradients empirically. Consider the task of optimizing a loss surface with two local minima defined as $\ell(w) = (w-4)(w-3)w^2$ with initial condition $w^{(0)} = -1.2$ using a momentum based optimizer with a parameterized momentum value θ (Figure 5a). At low momentum values the optimizer converges in the first of the two local minima, whereas
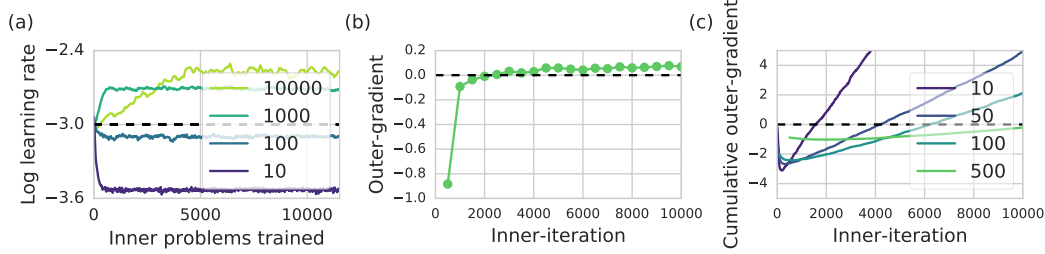
8

Figure 6: Large biases can result from reducing the number of steps per truncation in unrolled optimization. **(a)** Each line represents an experiment with the same total number of inner-steps (10k) but a different number of unrolling steps per truncation. In all cases, the initial learning rate is 0.001 (dashed line). We find low truncation amounts move away from the optimal learning rate (0.004), increasing the outer-loss. **(b)** Outer-gradient at each truncation over the course of inner-training. Initial outer-gradients are highly negative, trying to increase the learning rate, while later outer-gradients are slightly above zero, decreasing learning rate. The total outer-gradient is the sum of these two competing directions. **(c)** Cumulative value of outer-gradient vs inner-training step. When a line crosses the zero point, the outer-learned learning rate is at an equilibrium. For low unrolling steps, the cumulative sum is positive after a small number of steps, resulting in a increasing learning rate, and decreasing outer-loss. As more inner-steps are taken, the bias increases, eventually flipping the outer-gradient direction resulting in increasing outer-loss.

for larger momentum values the optimizer settles in the second minima. With even larger values of momentum, the iterate oscillates between the two minima before settling. We visualize both the trajectory of $w^{(t)}$ over training and the final loss value for different momentum values in Figure 5b and 5c. With increasing unrolling steps, the loss surface as a function of the momentum θ becomes less and less smooth, and develops near-discontinuities at some values of the momentum.

In the case of neural network inner-problems and neural network optimizers, the outer-loss surface can grow even more complex with increasing number of unrolling steps. We illustrate this in Figure 5d and 5e for slices through the loss landscape $L(\theta)$ of the outer-problem for a neural network optimizer.

A.4 Increasing bias with truncated gradients

Existing work on learned optimizers often avoids exploding gradients §A.3 by using a short truncation window. Here, we demonstrate the bias short truncation windows can introduce in unrolled optimization. These results are similar to those presented in **(author?)** [34], except that we utilize *multiple* truncations rather than a single, shortened unroll. First, consider outer-learning the learning rate of Adam when optimizing a small two layer neural network on MNIST[35]. Grid search can be used to find the optimal learning rate which is 0.004. We initialize Adam with a learning rate of 0.001 and outer-train using increasing truncation amounts (Figure 6a). Despite initializing close to the optimal learning rate, when outer-training with truncated backprop the resulting learning rate decreases, increasing the outer-loss. The sum of truncated outer-gradients are anti-correlated to the true outer-gradient. We visualize the per-truncation gradients for 500 step unrolls in Figure 6b and cumulative truncated gradients in Figure 6c. Early in inner-training there is a large negative outer-gradient which increases the learning rate. Later in inner-training, the outer-gradients are positive, decreasing the learning rate. Thus, the optimizer parameter is pulled in *opposite* directions by truncated gradients early versus late in inner-training, revealing an inherent tension with truncated unrolled optimization.

B Derivation of the unrolled gradient

After T steps of gradient descent, the final loss becomes $\ell(w^{(T)})$. We can update the parameters of the optimizer (θ) by computing the derivative of this final loss with respect to θ: $\frac{d\ell(w^{(T)})}{d\theta} =$

$\nabla \ell(w^{(T)})^T \frac{dw^{(T)}}{d\theta}$. The second term in this expression can be defined recursively as:

$$\frac{dw^{(T)}}{d\theta} = \left(I - \theta \nabla^2 \ell \left(w^{(T-1)}\right)\right) \frac{dw^{(T-1)}}{d\theta} - \nabla \ell \left(w^{(T-1)}\right),$$

where $\nabla \ell$ and $\nabla^2 \ell$ are the gradient and Hessian of the target problem, respectively. This expression highlights where the exploding outer-gradient comes from: the recursive definition of $\frac{dw^{(T)}}{d\theta}$ means that computing it will involve a product of the Hessian at every iteration.
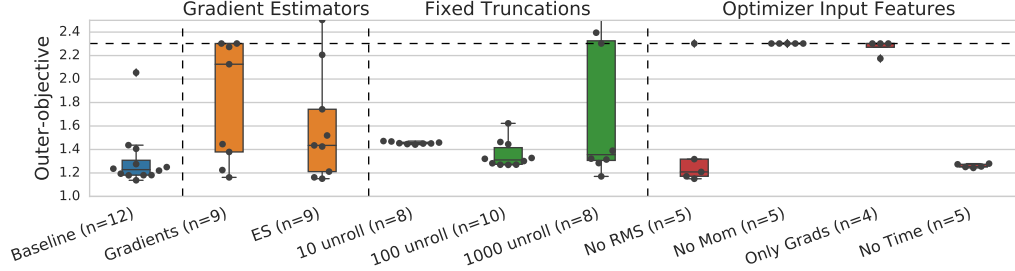
B.1 Ablations

Figure 7: We compare the model described in §3.1 with different features removed. Shown above is the distribution of outer-loss performance across n random seed averaged between 380k and 420k inner problem trained. For full learning curves, see Appendix G. First, with regard to gradient estimators we find the combined estimator is more stable and out performs both the analytic gradient, and evolutionary strategies. With regard to a curriculum, we find poor convergence with lower numbers of unrolling steps (10, 100) and high variance with larger numbers of truncation's steps (1000). Finally with regard to optimizer features, we find our learned optimizers perform nearly as well *without* the RMS terms and *without* the time horizon input, but fails to converge when not given access to momentum based features.

To assess the importance of the gradient estimator discussed in §2, the unrolling curriculum §**??**, as well as the features fed to the optimizer enumerated in §3.1, we re-trained the learned optimizer removing each of these additions. In particular, we trained optimizers with: only the analytic gradient (Only Gradients), only with evolutionary strategies (Only ES), a fixed number unrolled steps (10, 100, 1000) as opposed to a schedule, no RMS gradient scaling (No RMS), no momentum terms (No Mom), no momentum and no RMS scaling (Only Grads), and without the current iteration (No Time). To account for variance, each configuration is repeated with multiple random seeds. Figure 7 summarizes these findings, showing the learned optimizer performance for each of these ablations. We find that the gradient estimator (in §2) and a increasing schedule of unroll steps are critical to performance, along with including momentum as an input to the optimizer.

C   Outer-Training Algorithm

D   Architecture details

D.1   Architecture

In a similar vein to diagonal preconditioning optimizers, and existing learned optimizers our architecture operates on each parameter independently. Unlike other works, we do not use a recurrent model as we have not found applications where the performance gains are worth the increased computation. We instead employ a single hidden layer feed forward MLP with 32 hidden units. This MLP takes as input momentum terms, as well as rms terms inspired by Adam computed at a few different decay values: [0.5, 0.9, 0.99, 0.999, 0.9999] [9]. A similar idea has been explored with regard to momentum parameters in [36]. We also pass in 2 terms: $r = \dfrac{1}{\text{rms} + \epsilon}$ and $r$ms also from computations performed in Adam. Despite being critical in [9] we find these features of minimal

Initialize outer-parameters (θ).
while Outer-training, for each parallel worker **do**
    Sample a dataset $\mathcal{D}$, from $\mathcal{T}$.
    Initialize the inner loop parameters $w^{(0)}$ randomly.
    for Each truncation in the inner loop **do**
        Sample ES perturbation: $e \sim N(0, \sigma^2 I)$.
        Sample a number of steps per truncation, k, based on current outer-training index.
        Compute a positive, and negative sequence starting from $w^{(t)}$ by iteratively applying (for k steps), $u(\cdot; \theta + e)$, to $w^{(t)}$
        Compute a pair of outer-objectives L^+, and L^- using the 2 sequences of w from t to k using either the train or validation inner-problem data.
        Compute a single sample of $g^{rp} = \nabla_\theta \frac{1}{2}(L^+ + L^-)$.
        Compute a single sample of $g^{es} = \frac{1}{2}(L^+ - L^-)\nabla_\theta log(N(e; \theta, \sigma^2 I))$
        Store the sample of (g_{rp}, g_{es}) in a buffer until a batch of samples is ready.
        Assign the final w from one of the two sequences to $w^{(t+k)}$.
    end for
end while
while Outer-training and in parallel **do**
    When a batch of gradients is available, compute empirical variance and empirical mean of each weight for each estimator.
    Use inverse variance weighting to compute the combined gradient estimate.
    update meta-parameters with SGD: $\theta \leftarrow \theta - \alpha g_{combined}$ where α is a learning rate.
end while

Algorithm 1: Outer-training algorithm using the combined gradient estimator.

impact (see §B.1). This is $5 \cdot 4 = 20$ features in total. The current gradient as well as the current weight value are also used as features. By passing in weight values, the optimizer can learn to do arbitrary norm weight decay. To emulate learning rate schedules, the current training iteration is fed in transformed via applying a tanh squashing functions at different timescales: $tanh(t/\eta - 1)$ where η is the timescale. We use 9 timescales logarithmicly spaced from (3, 300k). This leaves us in total with 31 features.

All non-time features are normalized by the second moment with regard to other elements in the "batch" dimension (the other weights of the weight tensor). We choose this over other normalization strategies to preserve directionality. These activations are then passed the into a hidden layer, 32 unit MLP with relu activations. Many existing optimizer hyperparameters (such as learning rate) operate on an exponential scale. As such, the network produces two outputs, and we combine them in an exponential manner: $exp(\lambda_{exp}o_1)\lambda_{lin}o_2$ making use of two temperature parameters λ_{exp} and λ_{lin} which are both set to $1e-3$. Without these scaling terms, the default initialization yields steps on the order of size 1 – far above the step size of any known optimizer and result in highly chaotic regions of θ. It is still possible to optimize given our estimator, but training is slow and the solutions found are quite different.

D.2 Outer-Data Distributions

The outer-training set consists of a family of 10 way classification problems using cross entropy loss on subsets of 32x32 Imagenet. To form a outer-train and outer-test set, we randomly split imagenet into 700 classes for outer-train, and 300 classes for outer-test.

D.3 Inner-problem

The optimizer targets a 3 layer convolutional neural network with 3x3 kernels, and 32 units per layer. The first 2 layers are stride 2, and the 3rd layer has stride 1. We use relu activations and glorot initializations [37]. At the last convolutional layer, an average pool is performed, and a linear projection is applied to get the 10 output classes.

D.4 Outer-Training

We train using the algorithm described in 1 using a linear schedule on the number of unrolling steps from 50 - 10k over the course of 5k outer-training iterations. To add variation in length, we additionally shift this length by a percentage uniformly sampled between (-20%, 20%). We optimize the outer-parameters, θ, using Adam [4] with a batch size of 128 and with a learning rate of 0.003 for the training outer-objective and 0.0003 for the validation outer-objective, and $\beta_1 = 0.5$(following existing literature on non-stationary optimization [38]). While both values of learning rate work for both outer-objectives, we find the validation outer-objective to be *considerably* harder, and training is more stable with the lower learning rate.

E   Additional inner loop problem learning curves

We plot additional learning curves from both the outer-train task distribution and the outer-validation task distribution. The horizontal lines represent the minimum performance achieved over 20k steps. See Figure 1.
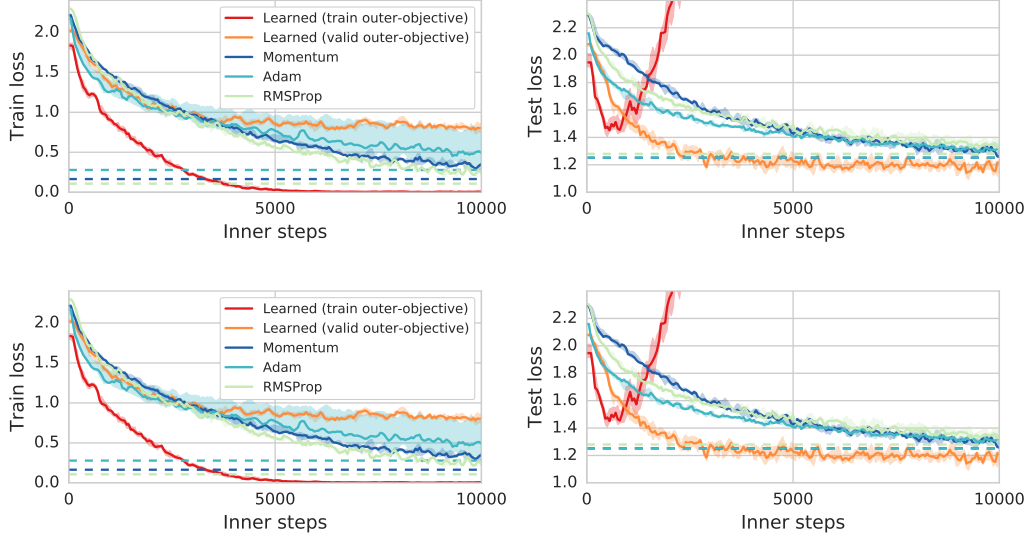

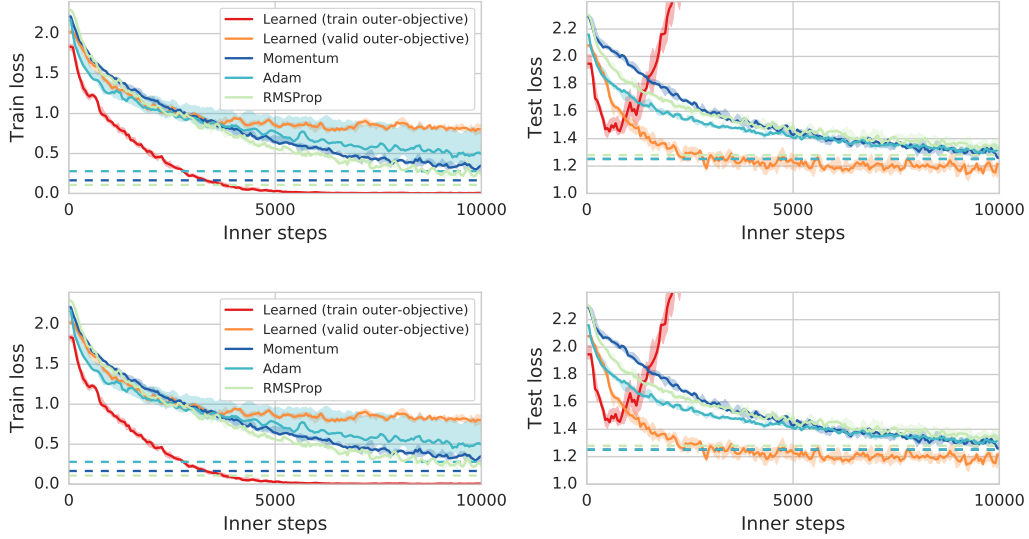Figure 8: Additional outer-validation problems.

Figure 9: Outer-training problems.

F   Inner-loop training speed

When training models, often one cares about taking less wallclock time as compared to performance per weight update. Much like existing first order optimizers, the computation performed in our learned optimizer is linear in terms of number of parameters in the model being trained and smaller than the cost of computing gradients. The bulk of the computation in our model consists of two batched matrix multiplies of size 31x32, and 32x2. When training models that make use of weight sharing, e.g. RNN or CNN, the computation performed per weight often grows super linearly with model size. As the learned optimizer methods are scaled up, the additional overhead in performing more complex weight updates will vanish.

For the specific models we test in this paper, we measure the performance of our optimizer on CPU and GPU. We implement Adam, SGD, and our learned optimizer in TensorFlow for this comparison. Given the small scale of problem we are working at, we implement training in graph in a *tf.while_loop* to avoid TensorFlow Session overhead. We use random input data instead of real data to avoid any data loading confounding. On CPU the learned optimizer executes at 80 batches a second where Adam runs at 92 batches a second and SGD at 93 batches per second. The learned optimizer is 16% slower than both.

On a GPU (Nvidia Titan X) we measure 177 batches per second for the learned and 278 batches per second for Adam, and 358 for sgd. This is or 57% slower than adam and 102% slower than SGD.

Overhead is considerably higher on GPU due to the increased number of ops, and thus kernel executions, sent to the GPU. We expect a fused kernel can dramatically reduce this overhead. Despite the slowdown in computation, the performance gains (greater than 400% faster in steps) far exceed the slowdown, resulting in an optimizer that is still considerably faster when measured in wallclock time.

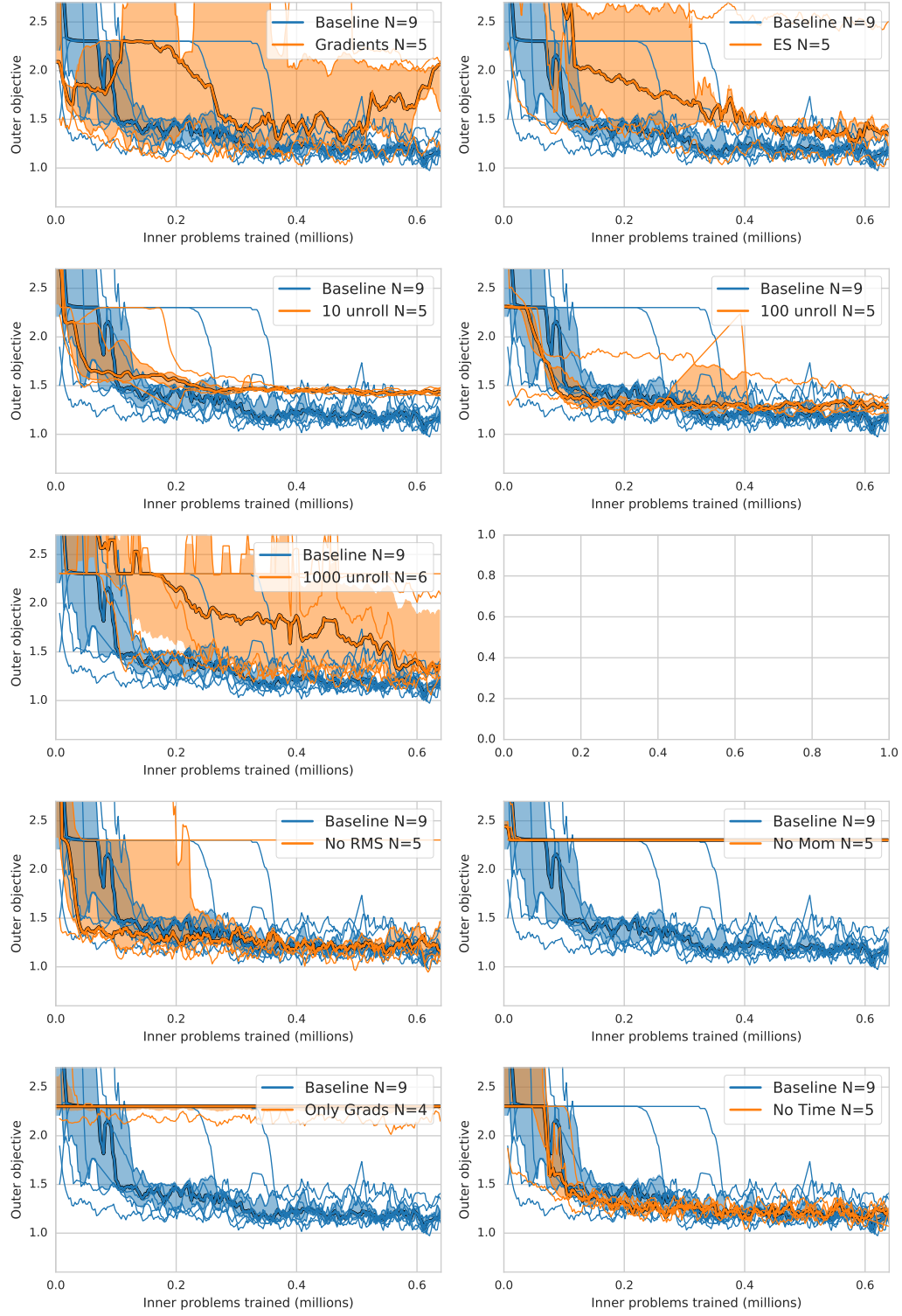# G   Ablation learning curves

Figure 10: Training curves for ablations described in §B.1. The thick line bordered in black is the median performance, with the shaded region containing the 25% and 75% percentile. Thinner solid lines are individual runs.