# PRACTICAL TRADEOFFS BETWEEN MEMORY, COMPUTE, AND PERFORMANCE IN LEARNED OPTIMIZERS

**Luke Metz**
Google Research, Brain Team
lmetz@google.com

**C. Daniel Freeman**
Google Research, Brain Team
cdfreeman@google.com

**James Harrison**
Google Research, Brain Team
jamesharrison@google.com

**Niru Maheswaranathan**
Meta *
niru@hey.com

**Jascha Sohl-Dickstein**
Google Research, Brain Team
jaschasd@google.com

## ABSTRACT

Optimization plays a costly and crucial role in developing machine learning systems. In learned optimizers, the few hyperparameters of commonly used hand-designed optimizers, e.g. Adam or SGD, are replaced with flexible parametric functions. The parameters of these functions are then optimized so that the resulting learned optimizer minimizes a target loss on a chosen class of models. Learned optimizers can both reduce the number of required training steps and improve the final test loss. However, they can be expensive to train, and once trained can be expensive to use due to computational and memory overhead for the optimizer itself. In this work, we identify and quantify the design features governing the memory, compute, and performance trade-offs for many learned and hand-designed optimizers. We further leverage our analysis to construct a learned optimizer that is both faster and more memory efficient than previous work. Our model and training code are open source[1].

## 1 INTRODUCTION

Despite the huge computational costs associated with training large neural models, the set of optimization algorithms used to train them has largely been restricted to simple update functions mapping from gradients to parameter updates (e.g. stochastic gradient descent (Robbins and Monro, 1951), Adam (Kingma and Ba, 2014), or RMSProp (Tieleman and Hinton, 2012)). These algorithms typically depend on a small number of hand-designed features and parameters. However, the last decade in machine learning research has repeatedly seen small, hand-designed models outperformed by parameterized models (such as neural networks) trained to purpose on large amounts of data (LeCun et al., 2015). Thus, a promising direction to improve training performance and reduce costs is to replace hand-designed optimizers with more expressive *learned optimizers*, trained on problems similar to those encountered in practice.

Learned optimizers specify parameter update rules using a flexible parametric form and learn the parameters of this function from a "dataset" of optimization tasks—a procedure typically referred to as meta-training or meta-learning (Andrychowicz et al., 2016; Finn et al., 2017; Hochreiter et al., 2001). Learned optimizers represent a path towards improved optimizer performance, and possess the ability to target different objectives (e.g. test loss (Metz et al., 2019a), or robustness (Metz et al., 2019b)), as well as the ability to leverage new features useful for optimization. Despite being an active area of research (Andrychowicz et al., 2016; Wichrowska et al., 2017; Chen et al., 2020; Metz et al., 2020b; 2021; Almeida et al., 2021; Zheng et al., 2022), they are not yet commonly used in practice. Several challenges have limited the widespread application of learned optimizers: they are typically difficult to meta-train on a task family of interest, they can require significant memory and compute overhead when applied, and they often generalize less well to novel tasks than hand-designed optimizers.

In this work, we aim to precisely study the limitations of learned optimizers, and address these limitations via a novel learned optimizer architecture. In particular, we explore and quantify the tradeoffs in terms of memory, compute, and generalization across a variety of optimizers, including hand-designed optimizers (Bottou, 2010; Tieleman and Hinton, 2012; Kingma and Ba, 2014), learned hyper-parameter controllers (Daniel et al., 2016; Hansen, 2016; Xu et al., 2017; 2019; Almeida et al., 2021), and neural network based learned optimizers (Andrychowicz et al., 2016; Wichrowska

* Work done while at Google Research
[1] https://github.com/google/learned_optimization

et al., 2017; Metz et al., 2020a), with the goal of understanding how choices in optimizer design affect performance and usability. Our core contributions are:

1. We present a thorough empirical characterization of the trade-offs inherent in different learned optimizer architectures and features, and a comparison of these learned optimizer architectures against their well-tuned hand-designed counterparts.
2. We develop a new per-parameter learned optimizer architecture, on the Pareto frontier with regards to performance, computational cost, and memory usage among existing learned, and hand designed optimizers.
3. We provide an open source implementation written in JAX (Bradbury et al., 2018) to enable future research and reliable benchmarking[2].

## 2 Optimizers

In this section we review and formalize the class of optimizers that are commonly used in training neural networks. We then define meta-learned optimizers, and highlight differences with standard optimization approaches. We describe several examples of both common, standard neural network optimizers as well as classes of learned optimizers, all of which are investigated in this paper.

### 2.1 Gradient Based Optimizers

Most first-order optimizers[3] used to train neural networks can be viewed as functions mapping from a history of gradients to parameter updates. We will assume the optimizer acts on an underlying model with parameters $\phi \in \Phi$, while maintaining an internal optimizer state $s \in S$. The parameters may be, for example, neural network weights, whereas the optimizer state includes quantities such as the accumulated momentum values in momentum-accelerated optimizers (Polyak, 1964; Nesterov, 1983). The optimizer acts by ingesting gradients $g$ (which arise from a specified loss function and a dataset) and outputting updated parameters $\phi'$.

More precisely, we define an optimizer as a pair of functions. The first, which we call the *Update* function, computes new parameter values $\phi'$ and state $s'$ from stochastic gradients, the current parameter value, and the current optimizer state. The second, which we refer to as the *Init* function, initializes the optimizer state. Both functions have hyperparameters $\theta \in \Theta$, such as the learning rate or the initial value of accumulators. Thus, we write the optimizer as:

$$\text{Optimizer} :: (\text{Init}, \text{Update}) \tag{1}$$

$$\text{Init} ::(\phi; \theta) \rightarrow s \tag{2}$$

$$\text{Update} ::(\phi, g, s; \theta) \rightarrow (s', \ \phi') \tag{3}$$

Optimizers can benefit from problem information beyond stochastic gradients, parameter values, and losses. For instance, methods that utilize line searches (Le et al., 2011), validation loss (Xu et al., 2019; Metz et al., 2020a), or the structure of the underlying computation graph (Martens and Grosse, 2015) all rely on additional information. However, the present work is restricted to optimizers which minimize training loss by mini-batch stochastic gradient descent.

**First-order hand-designed optimizers:** Hand-designed optimizers typically have a simple form, and a small number of hyperparameters ($\theta$), which are tuned by random search (Bergstra and Bengio, 2012), Bayesian optimization (Snoek et al., 2012), or other low-dimensional black-box optimization techniques (Bergstra et al., 2011; Golovin et al., 2017; Akiba et al., 2019). They mostly have low overhead in terms of compute and memory usage. For instance, Adam (Kingma and Ba, 2014) has two accumulators, and SGD has none[4].

In this work, we experiment with four kinds of hand-designed optimizers: SGD (Robbins and Monro, 1951; Bottou, 2010), SGDM (SGD with momentum) (Polyak, 1964), Adam (Kingma and Ba, 2014), and Nesterov accelerated Adam (Dozat, 2016) with AdamW (Loshchilov and Hutter, 2017) style weight decay (NAdamW). For SGD, SGDM, and Adam, we search over learning rates every half order of magnitude between $10^{-7}$ and 1. For SGM we set momentum to 0.9 and for Adam we set $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. For NAdamW we use random search with many more hyperparameter configurations per task (1000) and a much larger search space over 8 hyperparameters controlling: first and second momentum time scales, weight decays, and learning rate schedules. Past work has shown this to be a

[2] http://github.com/google/learned_optimization

[3] Optimizers using only gradient information and not higher order derivatives.

[4] Some hand-designed methods, such as Shampoo (Gupta et al., 2018; Anil et al., 2020), involve considerable compute overhead, but can make more progress per update step.

powerful search space (Metz et al., 2020b) and, in our work, this dramatically outperforms learning rate search. See Appendix F for more details. Many other hand-designed optimizer architectures have been proposed (Ruder, 2016; Zeiler, 2012; Reddi et al., 2018; You et al., 2019; Liu et al., 2019), but their practical benefits are small in most situations (Schmidt et al., 2020).

**Factorized optimizers:** In some settings, having even one additional copy of parameters to use for accumulators is too costly. Recent methods such as AdaFactor (Shazeer and Stern, 2018) and SM3 (Anil et al., 2019) factorize the weights and accumulate statistics using a sub-linear amount of memory with respect to parameters. This style of accumulator has not been explored in the context of learned optimizers, but we will show this provides an effective way to improve performance without meaningfully increasing memory overhead (§4.2).

### 2.2 Meta-Learned Optimizers

The meta-learning problem for optimizers consists of tuning the hyperparameters $\theta$ of a class of parameterized optimizers with respect to some loss function[5]. How is this different from the hyperparameter tuning discussed in the last subsection? While there is no formal difference between the hyperparameter selection problem and training learned optimizers, the learned optimizers we consider in this subsection universally include a black-box component with a (comparatively) large number of parameters (in our case, always parameterized by a neural network). This large number of parameters limits the effectiveness of traditional hyperparameter tuning methods such as random search, and so we focus on local optimization methods (including first-order gradient-based methods as well as zeroth-order methods) which are able to perform better in high dimensional optimization. Below, we outline several types of learned optimizer.

**Hyperparameter controllers:** Optimizing the hyperparameters of a hand-designed optimizer over a broad set of tasks may limit the performance within each specific task. These hand-designed optimizers can be augmented with a meta-learned controller, often parameterized as a neural network, that modulates the hyperparameters of the optimizer over the course of training to yield better performance in each particular problem (Daniel et al., 2016; Hansen, 2016; Xu et al., 2017; 2019; Almeida et al., 2021). This controller takes in summary statistics (e.g. gradient norms, loss values), and can either globally assign identical hyperparameters to all layers, or operate per-layer. One benefit of hyperparameter controllers is that their per-parameter compute overhead is small, as the majority of the computation only needs to be performed once per tensor, or per network rather than scaling with the number of parameters.

We introduce a novel hyperparameter controller architecture which we refer to as **nn_adam**. This architecture consists of an LSTM-based (Hochreiter and Schmidhuber, 1997) hyperparameter controller, operating on features derived from each tensor independently, and outputting Adam hyperparameters consisting of a per-tensor learning rate, $\beta_1$, $\beta_2$, and $\epsilon$. For features, this model uses normalized values derived from the first moment of gradients, the second moment, and the tensor shape. See Appendix C for details.

**Per-parameter learned optimizers:** Per-parameter learned optimizers (Andrychowicz et al., 2016) learn a function, often parameterized by a neural network, which is applied to each parameter independently, though sometimes with normalization performed across parameters in a tensor (Metz et al., 2019a).

**Multi-level approaches:** In an effort to add additional capacity to a learned optimizer while retaining good computational complexity with respect to number of parameters, hierarchical models have been proposed (Wichrowska et al., 2017; Metz et al., 2020a). These models leverage up to three levels of hierarchy: a *global controller*, which sends and receives activations from a *per-layer (or per-tensor) controller*, which finally sends and receives activations to a *per-parameter optimizer*.

**New per-parameter learned optimizer:** Finally, we introduce a new learned optimizer architecture (which we call **small_fc_lopt**) that combines architectural features of per-parameter and factorized optimizers, and outperforms both. This architecture will be directly motivated by the trade-offs among compute, memory, performance, and generalization shown in §4. Our learned optimizer incorporates an extremely *tiny*, per-parameter, MLP-based learned optimizer similar to that used in Metz et al. (2019a). This 197 parameter MLP takes as input 39 input features with 4 per-parameter accumulators (3 momenta at different meta-learned timescales, and 1 gradient second moment accumulator), and 3 AdaFactor accumulators also at 3 different meta-learned timescales. These features are passed into a 1 hidden layer, 4 hidden unit MLP. See Appendix A for additional details.

[5]Common choices of loss function for the meta-optimization problem include the average training loss across inner optimizer iterations, the average validation loss, as well as the terminal train/validation loss.

# 3 TRAINING AND META-TRAINING

In the previous section we specified possible architectures for standard optimizers (with a small number of hyperparameters) as well as learned optimizers. Both learned and hand designed optimizers are iteratively applied to some parameterized model, paired with a loss function and (possibly) a dataset. We refer to this collection as a *task*. We use the loss obtained by an optimizer on these tasks to select hyper-parameters (in the case of hand designed optimizers), and to optimize the learned optimizer weights. In this section, we discuss the tasks used, the measurement of performance by which we can compare optimizers (meta-loss), and discuss how the weights of the learned optimizers are computed (which we refer to as meta-optimization).

## 3.1 TASKS

Throughout this paper, the tasks of interest are neural network training problems. Each task is specified via three quantities. The first is the underlying model architecture and the initial parameter values (or a procedure for initializing the model parameters). The second is a function to generate batches of data, and the third is a loss function. While a more abstract definition of a task could cover more general optimization problems, we aim to address neural network training as a setting and believe generalizations are (in most cases) straightforward. In this work we consider solely supervised learning. We also consider only a single function to generate a batch of data, though this could easily be extended to multiple functions corresponding to, for example, train and validation loss. As discussed in the next subsection, we focus solely on training loss for simplicity.

We primarily consider two tasks in this paper: A 2 hidden layer MLP with 128 hidden units and ReLU activations on Fashion MNIST (Xiao et al., 2017), and a 3 layer convolutional network on CIFAR-10 (Krizhevsky et al., 2009). See Appendix B for more details and implementations. In Section 4.5, to assess generalization, we additionally evaluate optimizers meta-trained on these two tasks on three additional problems.

The tradeoffs inherent in optimizer design are task dependent (see §4.3), and the per-parameter compute and memory requirements of the optimizer must be balanced against the per-parameter compute and memory requirements of the task. These latter requirements are a function of parameter sharing, sparsity in parameter use, model architecture, and minibatch size (compute overhead per parameter can be made arbitrarily small by increasing the minibatch size). The two problems we consider have different compute and memory requirements and were therefore chosen as reasonable baseline tasks providing insight into optimizer performance at different points in the space of possible tasks. Moreover, these (relatively small) tasks were chosen to enable the large-scale evaluation and comparisons done in this paper.

## 3.2 META-LOSS AND META-OPTIMIZATION

To evaluate an optimizer, we apply our optimizer for 2,000 iterations and evaluate the average loss obtained over the course of training. In this work, we exclusively focus on training loss performance as opposed to validation loss. This is to decouple optimization performance tradeoffs from the implicit regularization effects of learned optimizers shown in Metz et al. (2019a; 2020a).

We train optimizers targeting the two tasks described above by randomly sampling from a fixed search space for hand-designed optimizers, and Persistent Evolution Strategies (PES) (Vicol et al., 2021) to train the learned optimizers. See Appendix D.2 for details. To minimize confounds, we focus on the scenario where meta-train matches meta-test (i.e. the tasks presented during training and testing are the same), and examine the overhead and performance tradeoffs inherent in a learned optimizer meta-trained to optimize a single task.

# 4 EXPLORING TRADEOFFS ACROSS OPTIMIZER FAMILIES

In this section we experimentally explore tradeoffs when designing learned optimizers. In §4.1, we show memory and time trade-offs for various hand-designed and learned optimizers. In §4.2, we focus on per-parameter learned optimizers and explore the impact of both feature choice and size of the learned optimizer. In §4.3, we discuss computational costs of running learned optimizers as a function of task features (such as the number of network weights). In §4.4, we tie all these evaluations together and show wall-clock time performance for the different tasks. In §4.5, we explore meta-generalization—applying optimizers to a task different from those in which they were meta-trained.

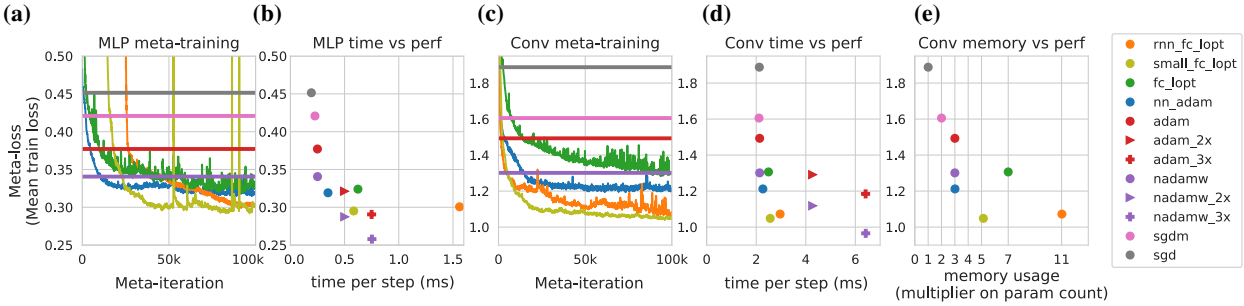


Figure 1: **Optimizer overhead depends on problem specification** We show two different tasks with large, and small, overheads. **(a,c)** Meta-learning curves targeting training an MLP on Fashion MNIST, and a ConvNet on CIFAR-10, respectively. **(b,d)** Per-iteration run-time vs achieved meta-loss for different optimizers for the MLP and ConvNet respectively. For hand designed optimizers (horizontal bars) we show the best performing hyperparameters. Additionally computed is adam_2x, adam_3x, nadamw_2x, and nadamw_3x which show Adam and NAdamW run for 2x and 3x as many iterations. **(e)** Memory vs. best meta-loss for each optimizer. The new learned optimizer we introduce, small_fc_lopt, and our Adam controller baseline nn_adam, lie on the Pareto frontier with respect to both memory and compute time among the optimizers and tasks tested.

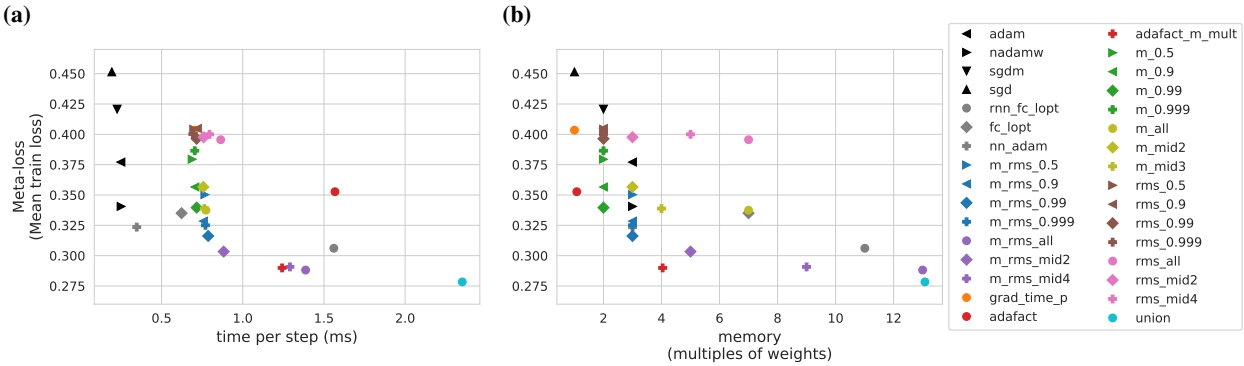


Figure 2: **Trade-offs in performance, compute, and memory overhead for different learned optimizer architectures. (a)** Performance vs. compute and **(b)** performance vs. memory is shown for many choices of input features for an MLP-based learned optimizer, as well as for baselines consisting of both hand-designed (black) and previously published learned optimizers (gray). The best raw performance is achieved by the learned optimizer configuration with the largest memory and compute overhead.

### 4.1 Compute, memory, performance tradeoffs for learned and hand-designed optimizers

We characterize the trade-offs between performance, memory overhead, and compute overhead for both hand-designed and learned optimizers. The optimizers examined here consist of the hand-designed optimizers (**SGD**, **Adam**, and **NAdamW**), the MLP optimizer from Metz et al. (2019a) (**fc_lopt**), the hierarchical optimizer from Metz et al. (2020a), (**rnn_fc_lopt**), a hyperparameter controller described in §2.2 (**nn_adam**), and the per-parameter optimizer proposed in §2.2 (**small_fc_lopt**).

Meta-training curves are shown in Figure 1ac. We additionally show final performance of the fully trained learned optimizer as a function of compute time per step (Figure 1bd) and with respect to memory usage (Figure 1e). We find learned optimizers can achieve lower meta-loss than baselines, but at the cost of more compute time and memory usage. For the MLP task, the cost of the learned optimizer far outstrips the cost of a hand-designed optimizer (taking > 5x more time in the case of rnn_fc_lopt). For the CIFAR-10 ConvNet however, the compute overhead is small relative to overall compute, due to the much larger per-parameter cost for computing gradients for a ConvNet.

### 4.2 Design choices for the MLP learned optimizer

To guide learned optimizer design, we explore the memory, time, and performance trade-offs associated with different choices of input features for an MLP learned optimizer. The dominant source of memory overhead is the inclusion of additional per-parameter accumulators. We explore two kinds of per-parameter accumulators that optimizers use—the exponential moving average of the gradient's first moment, and second moment as used in momentum and RMSProp

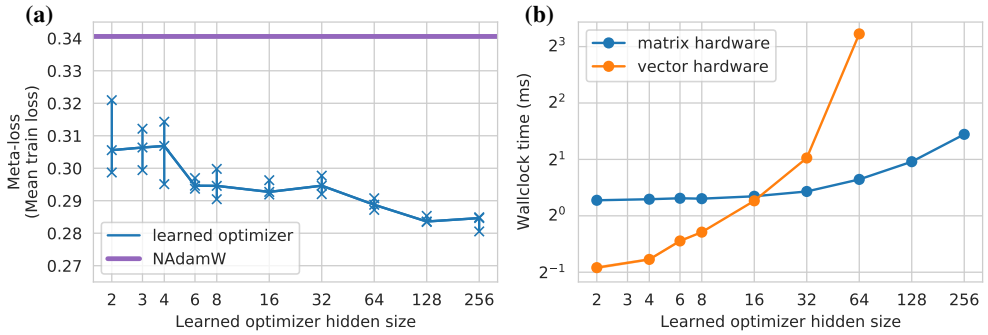


Figure 3: **Trade-offs in performance and compute overhead as a function of learned optimizer size.** A grid search is performed over the hidden state size of an MLP-based learned optimizer, for a fixed set of input features (based on small_fc_lpot, §2.2). **(a)** Performance vs. hidden state size for the MLP-based optimizer, with a baseline of the best hand-designed optimizer (nadamw, §2.1). Each "×" denotes a random seed. As hidden size increases, performance improves, and variance in performance decreases. **(b)** Compute time measured on a TPUv2 as a function of performance achieved for different hidden sizes for two different low-level implementations – one using on-accelerator matrix multiplication hardware, and the other using on-accelerator vector hardware. For matrix hardware, we find roughly constant performance up to 64 hidden units, whereas with vector hardware, we see speedups as hidden size is reduced all the way to two units.

(Tieleman and Hinton, 2012) respectively. Unlike existing optimizers, the learned optimizers we explore accumulate these statistics over multiple timescales[6]. In addition to these, we also explore preconditioning features based on AdaFactor (Shazeer and Stern, 2018) which use sub-linear memory in parameter count.

We plot performance vs. compute cost, and performance vs. memory, in Figure 2. We plot baselines with hyperparameters found via random grid search (in black) (adam, sgd, sgdm, nadamw), and baseline learned optimizers (in gray) (fc_lopt (Metz et al., 2019a), rnn_fc_lopt (Metz et al., 2020a), nn_adam (§2.2). All other conditions consist of differently parameterized learned optimizers with different input feature. Each configuration is trained with PES (Vicol et al., 2021) for 100k meta-training iterations. We test optimizers using only a single momentum accumulator with different decays (in green), multiple momentum accumulators (in yellow), a single second moment accumulator (in brown), multiple second moment accumulators (in pink), two accumulators with the same decay for first and second moments (in blue), multiple decay first and second moments (in purple), using AdaFactor features with and without additional momentum accumulators (in red), using only gradient features (in orange), and finally the union of all features (in gray). See Appendix D.3 for more experimental details.

We find the general trend that providing more features to a learned optimizer leads to better performance. However, including more accumulators increases the computational and memory overhead of using these optimizers. AdaFactor features by themselves (**adafact**) use very little memory, but do not perform well. Combining a small number of momentum features with AdaFactor features (**adafact_m_mult**) recovers the performance of using second moment accumulators, without the need for second moment accumulators.

Finally, we explore varying the hidden size of the MLP (Figure 3). Using the same features as in small_fc_lpot (§2.2), we sweep the hidden size of the MLP from 2 to 256 units. For each width we perform a small hyperparameter search over meta-learning rate selecting between $3 \cdot 10^{-5}$,$10^{-4}$, $3 \cdot 10^{-4}$ and take the best performing learning rate for each width. Surprisingly, an extremely narrow MLP is sufficient to outperform the best hand-designed baseline (NAdamW). Increasing width boosts performance, but performance improvements diminish.

The relationship between learned optimizer width and compute overhead depends heavily on implementation details. TPU and other accelerator hardware often have specialized matrix multiplication units that operate on fixed dimensional matrices (e.g. TPUv2 has 128x128 systolic arrays (Norrie et al., 2020)). For a naive implementation of the learned optimizer using matrix multiplication kernels on TPUv2, there are no significant speedups from shrinking the optimizer width below approximately 64 units. These matrix are too small and thus don't fully utilize the matrix units. If matrix multiplication is instead expanded explicitly in terms of element-wise operations, then continued speedups can be achieved even for optimizer hidden state vectors of two units, achieving a nearly 2x speedup over the use of matrix multiplication primitives as these element wise computations use a different subset of the accelerator hardware for the same computation. Profiling suggests that even greater speedups should be possible using custom kernels (as are frequently written for hand-designed optimizers). In a sense, these tiny learned optimizers, with matrix multiplications

[6]This is similar to what is done in the AggMo (Lucas et al., 2018) hand-designed optimizer.

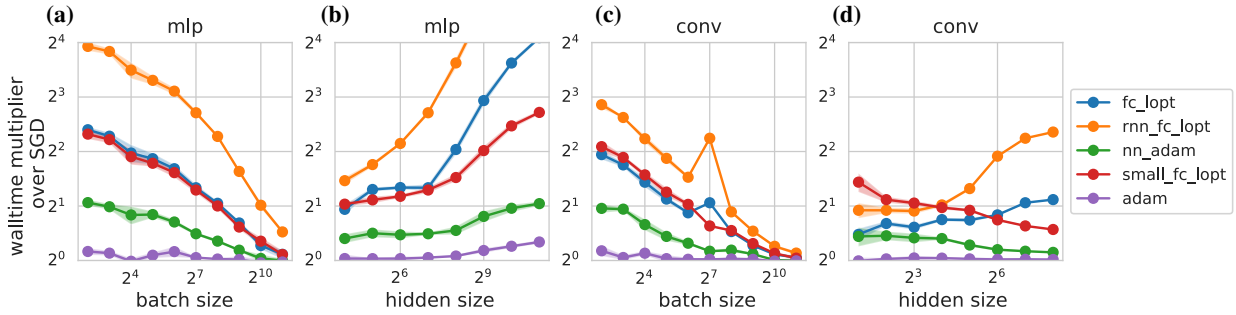


Figure 4: **Optimizer compute overhead shrinks with increasing batch size, and has a model-specific interaction with model size.** Plot shows overhead of different optimizers relative to training with SGD, as a function of width and batch size for both an MLP and ConvNet. Shaded regions denote standard deviation over 5 measurements.

Table 1: **On realistic tasks, the overhead of a learned optimizer is typically small.** We show parameter count, time per step for SGD, and overhead of the small_fc_lopt learned optimizer for four tasks: two different sized ResNets, with different batch sizes (BS), as well as two "small" Transformers (Vaswani et al., 2017) with different word sequence lengths (L) and batch sizes (BS). All numbers are medians over 10 timings. Standard error is under the reported number of digits.

| Task | Params | SGD step time (ms) | LOpt step time (ms) | LOpt multipler |
|---|---|---|---|---|
| ResNet18(BS=128) | 11.7M | 159 | 180 | 1.13 |
| ResNet50(BS=32) | 25.6M | 99.5 | 137.9 | 1.39 |
| Transformer(L=256,BS=16) | 43.1M | 91.2 | 132.4 | 1.45 |
| Transformer(L=512,BS=2) | 43.1M | 29.7 | 70.9 | 2.39 |

expanded, blur the line between hand designed and learned optimizers as both implementations are a handful of element wise floating point operations.

### 4.3 Overhead of learned optimizers on different tasks

To explore the dependence on task identity, we measure the relative overhead of training with a learned optimizer compared to SGD for different widths and batch sizes of the Fashion MNIST MLP and CIFAR-10 ConvNet. Results are shown in Figure 4. We find that, in all cases, increasing batch size lowers the overhead, as the cost to compute gradients increases but the cost of applying the optimizer remains constant. In the case of the ConvNets, increased the hidden size (channel count) of the target problem lowers the overhead for small_fc_lopt and nn_adam, but increases for fc_lopt and rnn_fc_lopt. For MLP target problems, increasing the target MLP's hidden size increases overhead for all optimizers including Adam. The asymptotic scaling of this behavior is due to both the computational complexity, and memory bandwidth, of the underlying hardware.

Next, we explore overheads for some common, large scale models. Table 1 show results for ResNets and Transformers, trained on a single TPUv2 chip. Distributed training would allow us to split the optimizer computation across devices and thus achieve even lower optimizer overhead. See Appendix D.4 for further details.

### 4.4 Performance with respect to wall clock time

In the previous sections, we measured the performance achieved by optimizers, and the computational overhead required to achieve that performance. In practice, one often cares most about the total wall clock time required to reach a given performance. Further, the optimal meta-parameters $\theta$ change depending on the length of inner-training. To quantify the achievable performance as a function of wall clock time, we compare training trajectories for both our learned and hand-designed optimizers. We apply a learned optimizer meta-trained for length 2k unrolls, and optimize hyperparameters of hand-designed optimizers to perform well on 1k, 2k, 3k, 4k, and 5k length unrolls. In Figure 5 we show the resulting performance for both the Fashion MNIST MLP task and CIFAR-10 ConvNet tasks. Our learned optimizer is always faster with respect to step count. With respect to wall-clock time, on the CIFAR-10 ConvNet task we also see faster training, while for the Fashion MNIST MLP task, where the relative overhead of the learned optimizer is large, NAdamW performs best.

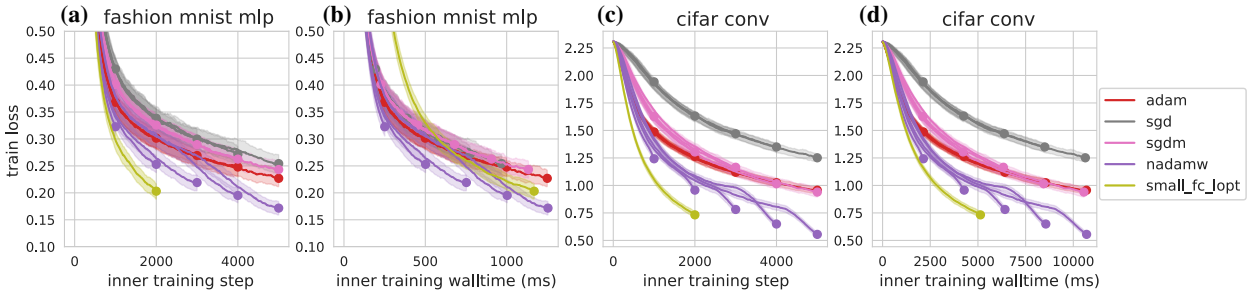


Figure 5: Inner-training curves for **(a,b)** the Fashion MNIST MLP task, and **(c,d)** the CIFAR-10 ConvNet task. We show training loss with respect to iteration and wall clock time. For each baseline, we show runs where the optimizer was meta-trained to achieve the best loss for varying inner-training lengths. Solid circles denote the final training performance after the number of training steps targeted for that baseline. For example, the yellow curve in (a) stops at 2000 steps because this small_fc_opt was meta-optimized for performance at 2000 inner training steps. Shaded region show standard deviation across 10 seeds.

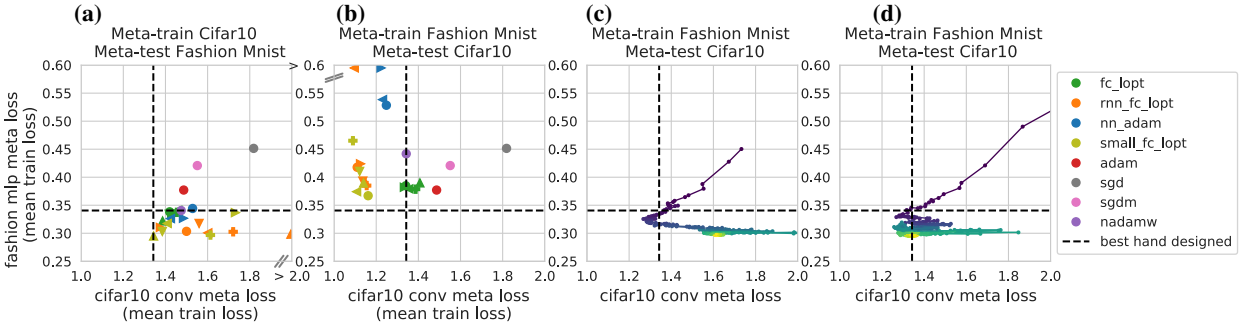


Figure 6: **Meta-generalization and meta-overfitting of learned and hand-designed optimizers. (a)** Optimizers are meta-trained (/hyperparameter-tuned) on the Fashion MNIST MLP task, and tested on the CIFAR-10 ConvNet task. **(b)** Optimizers are meta-trained on the CIFAR-10 ConvNet task, and tested on the Fashion MNIST MLP task. Each marker represents a different optimizer. Dashed lines denote best tuned performance across all hand-designed optimizers on the task. **(c,d)** Meta-training trajectory for the "small_fc_lopt" optimizer on the Fashion MNIST MLP task. The $y$-axis shows the meta-loss (performance after training the target task with the learned optimizer) on the Fashion MNIST MLP task, while the $x$-axis shows the meta-loss on the CIFAR-10 ConvNet task. Purple is early in meta-training, yellow is at the end of meta-training. Each pane shows a different random seed. Early in meta-training the learned optimizer generalizes, outperforming hand-designed optimizers on both the meta-training and meta-testing task. As meta-training continues, the learned optimizer meta-overfits, doing better on the Fashion MNIST MLP task but worse on the CIFAR-10 ConvNet task.

### 4.5 Meta-generalization: optimizer performance on holdout tasks

One final trade-off is the interaction between optimizer design choice and generalization performance. Generalization performance, in this context, refers to the ability of the optimizer to perform well at training a novel task, different from the task distribution used to meta-train the optimizer. To quantify this, we measure the performance of diverse optimizers meta-trained on one task, but then used as the optimizer for a novel task.

We meta-train each of the different learned optimizers on either the Fashion MNIST MLP or the CIFAR-10 ConvNet tasks. Over the course of meta-training we evaluate performance on 5 tasks:

1. The Fashion Mnist MLP described in 3.1.
2. The CIFAR-10 ConvNet described in 3.1 using 16x16 images for computational reasons.
3. A three hidden layer MLP trained on 16x16 imagenet.
4. A CIFAR-10 auto-encoder trained with mean squared error loss.
5. An LSTM (Hochreiter and Schmidhuber, 1997) language model trained on byte level LM1B(Chelba et al., 2013).

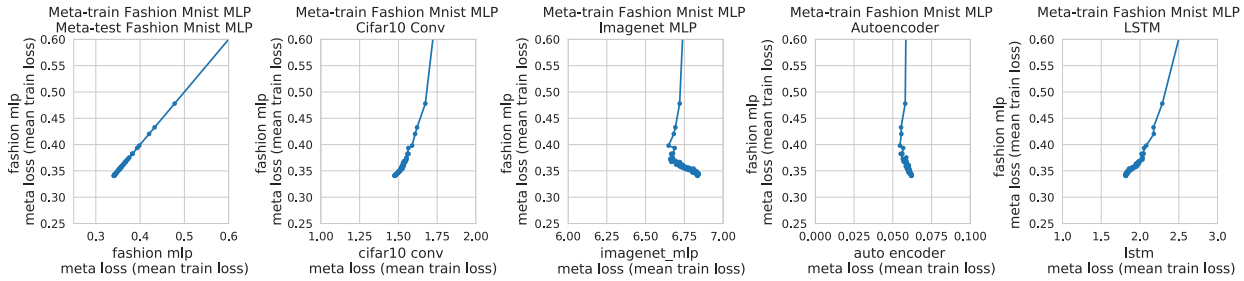


Figure 7: We show meta-training vs meta-test performance after, for a given budget, finding hyper-parameter for the NAdamW optimizer on the meta-training task (in this case Fashion MNIST MLP) and evaluating the best found hyperparameter for the given budget on various held out tasks. Similar to learned optimizers, we find meta-overfitting. We see this when evaluating on the Imagenet MLP and the Autoencoder but not when transferring to CIFAR-10 ConvNet nor the LSTM.

See Appendix E for more details and implementations.

First, we train a number of learned optimizers of different types and select the checkpoint which performs best on the meta-training task, and evaluate their performance on different held out tasks. We show transfer performance when meta-training and meta-testing on Fashion MNIST MLP and CIFAR-10 ConvNet in Figure 6, and the remainder of the comparisons in Appendix E. While there is some correlation across learned optimizer architecture, in general we find poor meta-generalization. Additionally, meta-generalization seems to depend more strongly on details of the meta-training process than it does on learned optimizer architectural choices. This variation poses challenges when reporting results due to the cost of meta-training.

To explore variance in the meta-training process, we plot performance over the course of meta-training on both the target-task, and the held out tasks. We show these dynamics for two different learned optimizer seeds in Figure 6cd. In both cases, meta-training trajectories exhibit high variability. However, they also both show an initial phase of correlated performance improvement, culminating in better performance than the baselines for both the target and held out task, before the optimizer finally overfits to the target-task.

This type of meta-overfitting is not unique to learned optimizers and happens even when trying to transfer hand designed optimizers from task to task. To show this, we simulate meta-training on the Fashion MNIST MLP by randomly sampling subsets of parameters from the original NAdamw search space for different budgets. We then find the best performance on the meta-training task and apply the best hyperparameters to the meta-test task. We show meta-train vs meta-test performance for different budgets in Figure 7. We see signs of meta-overfitting for some tasks, such as the ImageNet MLP and the MLP Autoencoder. For the others, CIFAR-10 ConvNet and LSTM, we continue to see correlation between meta-train and meta-test.

## 5 Related Work

Originally proposed in Bengio et al. (1992); Runarsson and Jonsson (2000), interest in learned optimizers has undergone a recent revival. Proposed learned optimizer architectures have included per-parameter RNNs (Andrychowicz et al., 2016), hierarchical models enabling sharing of information across parameters (Wichrowska et al., 2017; Metz et al., 2020a), and a simplified architecture consisting just of an MLP (Metz et al., 2019a). Optimizer meta-training techniques have included gradient descent (Maclaurin et al., 2015; Andrychowicz et al., 2016; Wichrowska et al., 2017), reinforcement learning (Li and Malik, 2017a;b), and more advanced training procedures (Lv et al., 2017; Maheswaranathan et al., 2019; Metz et al., 2019a; Chen et al., 2020; Vicol et al., 2021; Metz et al., 2021) leveraging both Evolution Strategies (ES), and gradients. Learned optimizers have been targeted at applications including model robustness (Metz et al., 2019b), chemistry (Merchant et al., 2021), min-max optimization (Shen et al., 2021), adversarial training (Xiong and Hsieh, 2020), few-shot learning (Ravi and Larochelle, 2016), swarm optimization (Cao et al., 2019), unsupervised learning (Metz et al., 2018), black box optimization (Chen et al., 2016), and MCMC sampling (Levy et al., 2017; Wang et al., 2017; Gong et al., 2018). Other work has analyzed learned optimizer behavior (Maheswaranathan et al., 2020). Bello et al. (2017) takes a different approach, and meta-learns symbolic rather than neural-network driven parameter update rules.

In an effort to understand computational costs, we look to Pareto frontiers of computation and memory vs performance. The concept of Pareto optimally was originally proposed in economics to understand how individuals can prosper with

finite resources (Newman, 1998) and has since become a useful tool in computer science. Studying trade-offs in this way is common in computer vision and natural language processing, where performance as a function of model size is often explored (Simonyan and Zisserman, 2014; He et al., 2016; Vaswani et al., 2017). Building efficient frontiers of models has been a target for meta-learning as well (Tan and Le, 2019). In the scope of learned optimizers, Metz et al. (2019a) explored training wall-clock efficiency, but on limited hardware (CPU) and with respect to a single target problem instance. Wichrowska et al. (2017) showed that the relative overhead of computing updates with a learned optimizer shrinks as batch size is increases. Zheng et al. (2022) propose a symbolic distillation meta-training step which converts neural network parameterized optimizers to a symbolic form resulting in both lower memory and compute costs. There has also been significant research exploring the trade-offs between different optimization techniques outside of deep learning, especially between stochastic, full batch, and between different second order methods. For example, Newtons method has led to a number of approximations – e.g. diagonal approximations (Duchi et al., 2011), block diagonal (Martens and Grosse, 2015; Gupta et al., 2018), as well as a large family of quasi newton methods (Dennis and Moré, 1977) (e.g. BFGS (Broyden, 1970) and its low memory counterpart, L-BFGS (Liu and Nocedal, 1989)).

## 6 Conclusion

In this work, we characterized practical trade-offs involved in designing learned optimizers, including those between performance optimizing a target task, compute and memory overhead associated with the learned optimizer, training time, choice of target task, and generalization to new tasks. Using the lessons learned from our careful exploration, we introduce an architecture that strikes a better balance between memory usage, compute, and performance. We then show that this learned optimizer architecture can be used to accelerate training on accelerator hardware.

The goal of this work was to provide a thorough investigation of the fundamental tradeoffs associated with learned optimizers. We view this paper as a first step toward the empirical characterization necessary for principled comparisons, but our experiments were limited in several ways. First, to control the number of covariates and perform the required experiments within a reasonable computation budget, we have limited ourselves to (primarily) two tasks, which themselves are simple compared with state-of-the-art neural network models. Moreover, we have provided only a limited investigation of meta-generalization and validation performance. In general, while this paper serves as a first step toward rigorous empirical comparison within this novel class of learned optimizers, further work is required to extend our results.

In order to make it easier for future research to build on our work, and to include better grounded empirical comparisons, all optimizers, tasks, and training code are open sourced in learned_optimization[7], an open source library written in JAX for designing, training, and testing learned optimizers.

## Acknowledgements

We would like to thank Chip Huyen, Ben Poole, Amil Merchant, and Wenqing Zheng for their support and comments on this work, as well as the entire Brain team for providing a wonderful research environment. We would also like to thank the authors of the python scientific computing stack including Numpy (Van Der Walt et al., 2011), and Matplotlib (Hunter, 2007).

[7] https://github.com/google/learned_optimization

## REFERENCES


Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Diogo Almeida, Clemens Winter, Jie Tang, and Wojciech Zaremba. A generalizable approach to learning optimizers. *arXiv preprint arXiv:2106.00958*, 2021.

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.

Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory-efficient adaptive optimization. *arXiv preprint arXiv:1901.11150*, 2019.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Second order optimization made practical. *arXiv preprint arXiv:2002.09018*, 2020.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc Le. Neural optimizer search with reinforcement learning. 2017. URL https://arxiv.org/pdf/1709.07417.pdf.

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.

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

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation, 2011.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

Yue Cao, Tianlong Chen, Zhangyang Wang, and Yang Shen. Learning to optimize in swarms. *arXiv preprint arXiv:1911.03787*, 2019.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013. URL http://arxiv.org/abs/1312.3005.

Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training stronger baselines for learning to optimize. *Advances in Neural Information Processing Systems*, 33, 2020.

Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. *arXiv preprint arXiv:1611.03824*, 2016.

Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. Learning step size controllers for robust neural network training. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

John E Dennis, Jr and Jorge J Moré. Quasi-newton methods, motivation and theory. *SIAM review*, 19(1):46–89, 1977.

Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

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

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

Wenbo Gong, Yingzhen Li, and José Miguel Hernández-Lobato. Meta-learning for stochastic gradient mcmc. *arXiv preprint arXiv:1806.04522*, 2018.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

Samantha Hansen. Using deep q-learning to control optimization hyperparameters. *arXiv preprint arXiv:1602.04062*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020. URL http://github.com/deepmind/dm-haiku.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

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

Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 and cifar-100 datasets. *URl: https://www. cs. toronto. edu/kriz/cifar. html*, 6, 2009.

Quoc V Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *ICML*, 2011.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Daniel Levy, Matthew D Hoffman, and Jascha Sohl-Dickstein. Generalizing hamiltonian monte carlo with neural networks. *arXiv preprint arXiv:1711.09268*, 2017.

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

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

Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

James Lucas, 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.

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

Niru Maheswaranathan, Luke Metz, George Tucker, Dami Choi, and Jascha Sohl-Dickstein. Guided evolutionary strategies: Augmenting random search with surrogate gradients. In *International Conference on Machine Learning*, pages 4264–4273. PMLR, 2019.

Niru Maheswaranathan, David Sussillo, Luke Metz, Ruoxi Sun, and Jascha Sohl-Dickstein. Reverse engineering learned optimizers reveals known and novel mechanisms. *arXiv preprint arXiv:2011.02159*, 2020.

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

Amil Merchant, Luke Metz, Sam Schoenholz, and Ekin Dogus Cubuk. Learn2hop: Learned optimization on rough landscapes. *ICML*, 2021.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Learning unsupervised learning rules. *arXiv preprint arXiv:1804.00222*, 2018.

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*, pages 4556–4565, 2019a.

Luke Metz, Niru Maheswaranathan, Jonathon Shlens, Jascha Sohl-Dickstein, and Ekin D Cubuk. Using learned optimizers to make models robust to input noise. *arXiv preprint arXiv:1906.03367*, 2019b.

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. *arXiv preprint arXiv:2009.11243*, 2020a.

Luke Metz, Niru Maheswaranathan, Ruoxi Sun, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*, 2020b.

Luke Metz, C Daniel Freeman, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Training learned optimizers with randomly initialized learned optimizers. *arXiv preprint arXiv:2101.07367*, 2021.

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.

Peter Newman. *The new Palgrave dictionary of economics and the law*. Springer, 1998.

Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P Jouppi, and David Patterson. Google's training chips revealed: Tpuv2 and tpuv3. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–70. IEEE Computer Society, 2020.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference on Learning Representations*, 2016.

S Reddi, Manzil Zaheer, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In *Proceeding of 32nd Conference on Neural Information Processing Systems (NIPS 2018)*, 2018.

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

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Thomas Philip Runarsson and Magnus Thor Jonsson. Evolution and design of distributed learning rules. In *Combinations of Evolutionary Computation and Neural Networks, 2000 IEEE Symposium on*, pages 59–63. IEEE, 2000.

Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley–benchmarking deep learning optimizers. *arXiv preprint arXiv:2007.01547*, 2020.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *arXiv preprint arXiv:1804.04235*, 2018.

Jiayi Shen, Xiaohan Chen, Howard Heaton, Tianlong Chen, Jialin Liu, Wotao Yin, and Zhangyang Wang. Learning a minimax optimizer: A pilot study. In *International Conference on Learning Representations (ICLR)*, 2021.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

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.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10553–10563. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/vicol21a.html.

Tongzhou Wang, Yi Wu, David A Moore, and Stuart J Russell. Meta-learning mcmc proposals. *arXiv preprint arXiv:1708.06040*, 2017.

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.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Yuanhao Xiong and Cho-Jui Hsieh. Improved adversarial training via learned optimizer. In *European Conference on Computer Vision*, pages 85–100. Springer, 2020.

Chang Xu, Tao Qin, Gang Wang, and Tie-Yan Liu. Reinforcement learning for learning rate control. *arXiv preprint arXiv:1705.11159*, 2017.

Zhen Xu, Andrew M Dai, Jonas Kemp, and Luke Metz. Learning an adaptive learning rate schedule. *arXiv preprint arXiv:1909.09712*, 2019.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Wenqing Zheng, Tianlong Chen, Ting-Kuei Hu, and Zhangyang Wang. Symbolic learning to optimize: Towards interpretability and scalability. In *International Conference on Learning Representations (ICLR)*, 2022.

## A SMALL_FC_LOPT ARCHITECTURAL DETAILS

We describe the full details of our proposed learned optimizer. The source code for this optimizer can be found in https://github.com/google/learned_optimization/blob/aa15091066aa5b3f45e6b7f4bee1c41fb7d467a0/learned_optimization/learned_optimizers/adafac_mlp_lopt.py.

Our learned optimizer consists of features, concatenated then fed into an MLP. These features contain:

- the parameter values
- the 3 momentum values ($m$)
- the second moment value ($v$)
- 3 values consisting of momenta normalized by rms gradient norm – $m/\sqrt{v}$
- the $\left(\sqrt{\text{v} + \epsilon}\right)^{-1}$ value
- 3 AdaFactor normalized gradient values
- the tiled, AdaFactor row features (3 features)
- the tiled, AdaFactor column features (3 features)
- $1/\sqrt{\text{adafact feats}}$ of these previous 6 features
- 3 features consisting AdaFactor normalized momentum values
- 11 features formed by taking the current timestep, $t$, and computing $\tanh(t/x)$ where $x \in \{1, 3, 10, 30, 100, 300, 1000, 3000, 10k, 30k, 100k\}$

All but the time features are normalized to have a second moment of 1 across the tensor and fed into a 4 hidden unit, 2 hidden layer MLP with a ReLU activation function then projected to 2 hidden dimensions representing a magnitude, $m$, and a scalar direction $d$ to be combined to form a predicted step: $\Delta\phi = \lambda_1 d\exp(\lambda_2 m)$ where $\lambda_1$ and $\lambda_2$ are constants set to 0.001 to keep initial step-sizes small.

In order to reduce computational overhead, this optimizer is dramatically smaller than past learned optimizers, containing only 197 meta-parameters.

# B TASKS WE META-TRAIN ON

## B.1 FASHION MNIST MLP

The MLP we use consists of 2 hidden layers with 128 hidden units and ReLU activations. It is trained on batches of Fashion MNIST re-scaled to lie between [0, 1] and on batch sizes of 128. We train with cross entropy loss. Our network was built in Haiku (Hennigan et al., 2020) and the implementation can be found at https://github.com/google/learned_optimization/blob/32c4f21ec238a12756afe70e3d699017ea938f5d/learned_optimization/tasks/fixed/image_mlp.py#L35.

## B.2 CIFAR-10 CONVNET

The ConvNet consists of 3 hidden layers with ReLU activations. All layers have kernel sizes of 3x3. The first layer has 32 units and stride 2. The following 2 layers have 64 hidden units and stride 1. All convolutions have same padding. We average over the spatial dimensions then linearly project to 10. We train on batches of 128 CIFAR-10 re-scaled between [0, 1] and use cross entropy loss. The implementation can be found at https://github.com/google/learned_optimization/blob/ba2b56565fb507368652d2e4a12ab305a6d99ded/learned_optimization/tasks/fixed/conv.py#L96.

# C NN_ADAM ARCHITECTURE

The nn_adam learned optimizer is a hyper parameter controller based learned optimizer. In addition to the description that follows, we provide an implementation at https://github.com/google/learned_optimization/blob/ba2b56565fb507368652d2e4a12ab305a6d99ded/learned_optimization/learned_optimizers/nn_adam.py#L161. For each tensor of the target problem, we compute some set of features (see C.1), feed them into a 32 unit LSTM, and output 4 values – log learning rate, beta 1, beta 2 (both parameterized as log(1 - beta)) and log epsilon. These hyperparameters are then fed into Adam where the update to each weight and accumulator follows the Adam update equations.

## C.1 PER TENSOR FEATURES

We use a same set of per-tensor features as used by the hierarchical learned optimizer in Metz et al. (2020a). For many features, we employ a simple transformation to obtain magnitudes of features. This transformation involves computing the log of the absolute value, clipping between -5 and 5, and rescaling by 0.5.

For each tensor, we use the following as a feature set:

- Transformed mean momentum value
- Sign of mean momentum
- Transformed variance squared of momentum
- Transformed mean of the accumulator of second moment
- Sign of the mean of the accumulator of second moment
- Transformed mean parameter value
- Sign of mean parameter value
- Transformed variance squared of parameter value.
- Transformed mean gradient value
- Sign of mean gradient value
- Transformed variance squared gradient
- Transformed mean absolute value of gradient.

# D EXPERIMENT DETAILS

## D.1 COMMON

For all experiments in this paper we meta-train with Persistent Evolutionary Strategies (Vicol et al., 2021) with a standard deviation of 0.01 and length 20 truncations with length 2k inner training steps. The meta-objective we target is mean training loss (clipped at the initialization value – $ln(10)$ in the classification problems). We use 4 distributed workers (each using a single TPU accelerator chip) in an async batched fashion. We use a meta-batch size of 4 with each meta gradient being an average from each worker which is itself an average over 8 tasks. For all models, we have an additional learner job (also a TPU chip) which averages meta-gradients and performs PES updates. We use Adam as the meta-optimizer in all experiments with gradient clipping of 3.0 done on each value of the gradient independently. Each training job has an additional 3 machines (15 for meta-generalization experiments), each with a single TPU chip to perform evaluations by training a task with the current meta-parameters. The meta-training curves we report are from these machines. All meta-training in this work took 2-4 days per experiment.

## D.2 §4.1 DETAILS: OPTIMIZER OVERHEAD VS OPTIMIZER TYPE

For each task, and learned optimizer pair we train 3 random seeds for 5 learning rates: [1e-5, 3e-5, 1e-4, 3e-4, 1e-3]. We show the best performing optimizer. We do this as opposed to mean as the low dimensional hidden size of small_fc_lopt (4 hidden units) can result in unstable training. We found using a simple learning rate schedule improves meta-training stability, but for a fair comparison to other optimizers we do not use any schedule here.

## D.3 §4.2 DETAILS: INPUT FEATURES EXPERIMENT DETAILS

For these experiments we use a fixed learning rate, set to $10^{-4}$ as this was found to be the best performing model for fc_lopt from §D.2. Given the amount of variations tried, we could not afford to search over learning rate for each configuration. For each configuration we compute 3 random seeds.

**grads_time_p**: Just using parameter, and gradient, and time step features.

**m_0.1, m_0.5, m_0.9, m_0.99, m_0.999**: Using parameter value, gradient, momentum with the listed decay value, and time step features.

**m_all**: Same as before but using multiple momentum values. In this case five values: 0.1, 0.5, 0.9, 0.99, 0.999.

**m_mid2**: Same as before but with 2 momentum values 0.5 and 0.9.

**m_mid3**: Same as before but with 3 momentum values 0.5, 0.9 and 0.99.

**rms_0.1, rms_0.5, rms_0.9, rms_0.99, rms_0.999**: Using parameter value, gradient, the second moment accumulator with the listed decay value, 1 over the sqrt of this feature, and time step features.

**rms_all**: Same as before but using multiple second moment accumulator values. In this case six values: 0.1, 0.5, 0.9, 0.99, 0.999, 0.9999.

**rms_mid2**: Same as before but with 2 second moment values 0.9 and 0.99.

**rms_mid4**: Same as before but with 3 second moment values 0.5, 0.9, 0.99, and 0.999.

**m_rms_0.1, m_rms_0.5, m_rms_0.9, m_rms_0.99, adams_0.999**: Using parameter value, gradient, the second moment accumulator with the listed decay value, 1 over the sqrt of this feature, momentum with the listed decay, as well as the product of momentum and 1 over the square root of the second moment (similar the Adam update) and time step features.

**m_rms_all**: Same as before but using multiple second moment and momentum accumulator values. In this case six values: 0.1, 0.5, 0.9, 0.99, 0.999, 0.9999.

**m_rms_mid2**: Same as before but with 2 accumulator timescales: 0.9 and 0.99.

**m_rms_mid4**: Same as before but with 3 accumulator timescales: 0.5, 0.9, 0.99 and 0.999.

**adafact**: Using parameter values, 6 adafactor accumulator decays (0.1, 0.5, 0.9, 0.99, 0.999, 0.9999) which are fed to the learned optimizer in the form 3 multiplications: 1 over sqrt, 1 over sqrt multiplied by the gradient, and by tiling both of the low rank accumulators.

**adafact_m_mul**: Same as before, but with 3 adafactor accumulators and 3 momentum accumulators of decays (0.5, 0.9, 0.99). In addition to the previous features, we also include the multiplication of momentum value by the preconditioner from adafactor.

**union**: The union of all features. This includes parameter value, gradient value, time features, all momentum and second moment accumulators (0.1, 0.5, 0.9, 0.99, 0.999, 0.9999), all features from adafactor computed with these same timescales, as well as multiplications of adafactor and momentum features.

### D.4 §4.3 DETAILS: LARGE SCALE OVERHEAD TIMINGS

We use the ResNet18, and ResNet50 implementations from Haiku (Hennigan et al., 2020) with the V2 flag set to true.

For transformers, we use vocab size of 256 to emulate byte level training, a hidden size of 768, 6 layers, and 12 self attention heads per layer. When applying dense layers we use a 4x widening factor.

## E EXTENDED META-GENERALIZATION EXPERIMENTS

### E.1 EXPERIMENTAL DETAILS

Over the course of training a learned optimizer on a particular task, we monitor performance on a variety of held out tasks described here.

**Fashion Mnist MLP**: This is a 2 hidden layer, 128 unit MLP trained on fashion mnist. Source code can be found https://github.com/google/learned_optimization/blob/32c4f21ec238a12756afe70e3d699017ea938f5d/learned_optimization/tasks/fixed/image_mlp.py#L35.

**CIFAR-10 Convnet**: This is convnet with 3 hidden layers trained on 16x16 CIFAR-10. It contains 3 hidden layers starting with a 32 channels stride 2, and followed by two 64 channel, stride 1 convolutions. Average pooling is then performed before linearly mapping to the number of output channels. An implementation can be found at https://github.com/google/learned_optimization/blob/78f25e8f1e9c6236a1f559b7b0b36859c59542d2/learned_optimization/tasks/fixed/conv.py#L86

**Imagenet MLP**: This is an MLP operating on 16x16 resized imagenet images. The network has 3 hidden layers, of size 256. An implementation can be found at https://github.com/google/learned_optimization/blob/aa15091066aa5b3f45e6b7f4bee1c41fb7d467a0/learned_optimization/tasks/fixed/image_mlp.py#L94.

**Auto Encoder**: This is an auto encoder trained on CIFAR-10 with mean squared error. The network consists 3 hidden layers with sizes 128, 32, 128. A full implementation can be found in https://github.com/google/learned_optimization/blob/aa15091066aa5b3f45e6b7f4bee1c41fb7d467a0/learned_optimization/tasks/fixed/image_mlp_ae.py#L101.

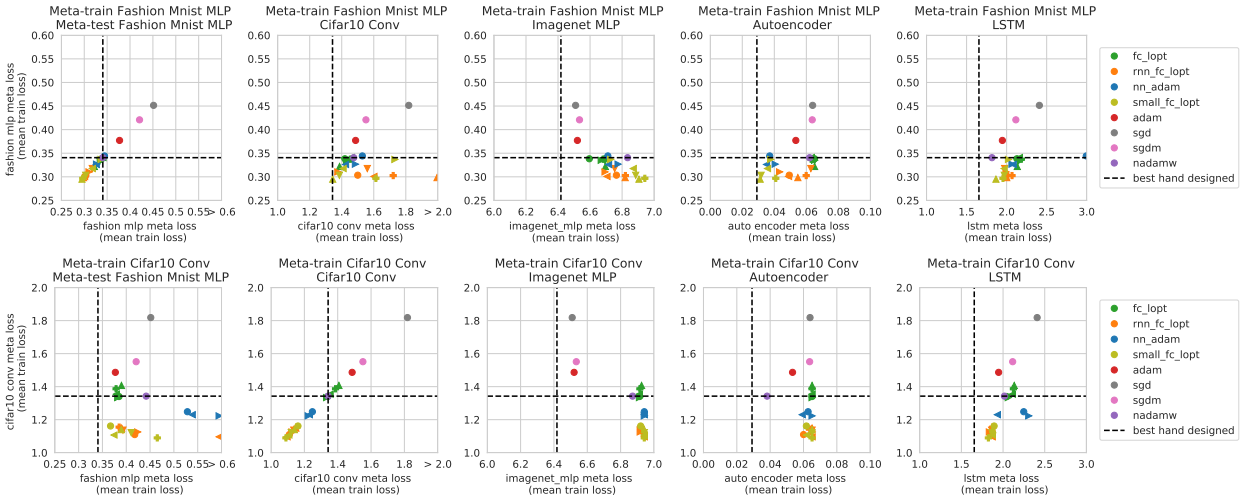


Figure 8: Further generalization results for more tasks. **Top row**: Meta-training on Fashion MNIST MLP and testing on 5 tasks. **Bottom row**: Meta-training on CIFAR-10 ConvNet and meta-testing on 5 tasks.

**LSTM language modeling**: This is a language model trained on (Chelba et al., 2013). The language is tokenized as bytes, and sliced into length 32 sequences. The model consists of embedding the tokens with a 64 dimensional lookup table, followed by a size 128 LSTM tasked to predict the next token. See https://github.com/google/learned_optimization/blob/aa15091066aa5b3f45e6b7f4bee1c41fb7d467a0/learned_optimization/tasks/fixed/rnn_lm.py#L142 for the impementation.

### E.2 Additional figures

In this section we provide additional experimental results for meta-generalization similar to §4.5. First, in Figure 8 we show additional performance measurements on more held out tasks. As in §4.5, we see poor meta-generalization and high variability.

We plot evaluations over the course of meta-training for each different learned optimizer type and multiple random seeds in Figure 9 when meta-training on the fashion Mnist MLP, and Figure 10 for the CIFAR-10 conv net. When meta-training and evaluating on the same distribution, we find stable evaluation loss. When evaluating on other kinds of tasks, we see wide variability in performance across both architecture, and even among different initializations of the learned optimizer weights holding all else fixed. In some cases, such the learned optimizers switches between performing optimization on the target task, and diverging as shown by the rapid spikes in the meta-loss.

Finally, we show an alternative plot of the same data discussed in the previous paragraph. This time, we plot meta-evaluation performance against meta-train performance. For each figure we show each learning rate, and each seed in a separate pane. We show the small_fc_lopt optimizer in Figure 11, the rnn_fc_lopt in Figure 12, the fc_lopt in Figure 13, and nn_adam in Figure 14. Once again we find high variability across architecture, learning rate, and random seed. In these figures, meta-overfitting is highlighted by a "c" shaped curve – meta-training performance continues to improve, but meta-evaluation performance gets worse after some point. Jagged lines / instability suggest a high sensitivity in performance on the evaluation task.

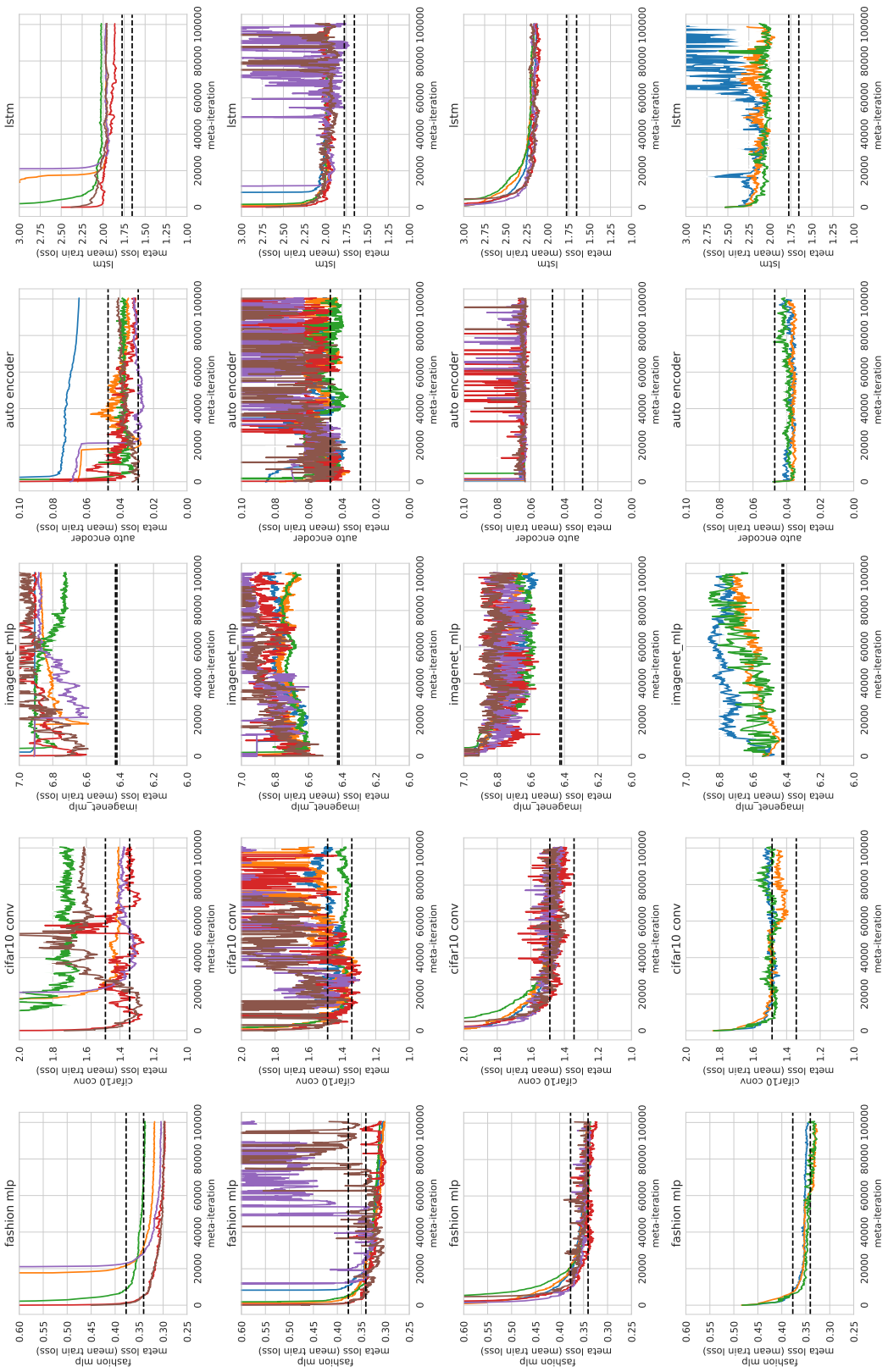


Figure 9: Meta-train and meta-test meta-training curves for different kinds of learned optimizers. In each row we show a different learned optimizer, with different colors denoting different random seed and or learning rate. From top to bottom we show small_fc_lopt, rnn_lopt, fc_lopt, nn_adam. In black we show both Adam (top line) and NAdamW (bottom line) baselines tuned to the task being tested on. Error bars denote standard error across different random seeds when evaluating a given set of learned optimizer weights. Note how there is very little variation in evaluation of meta-loss, but large variation between different meta-iterations. In this pane we meta-train each learned optimizer on **Fashion MNIST MLP**.

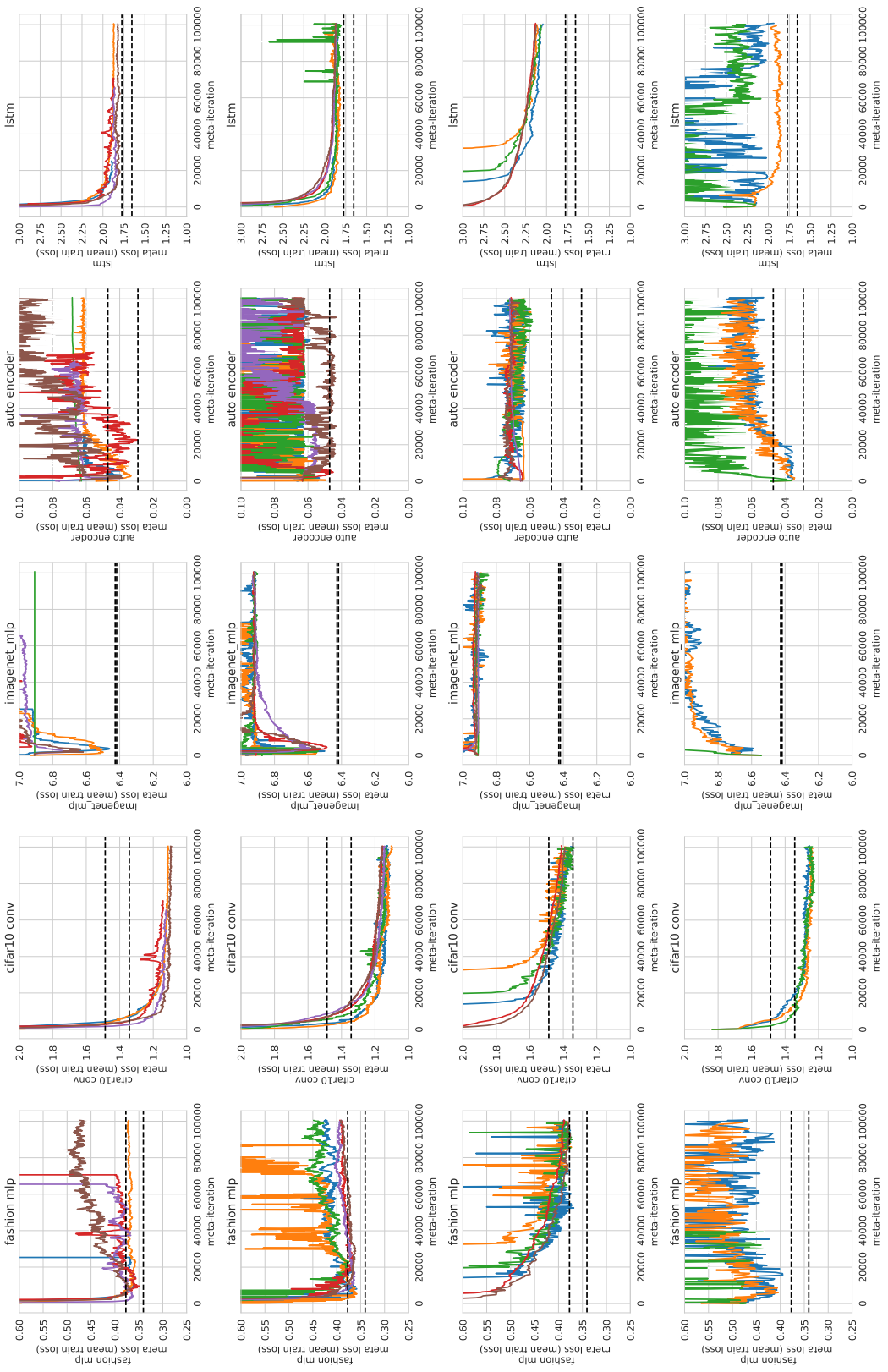


Figure 10: Meta-train and meta-test meta-training curves for different kinds of learned optimizers. In each row we show a different learned optimizer, with different colors denoting different random seed and or learning rate. From top to bottom we show small_fc_lopt, rnn_lopt, fc_lopt, nn_adam. In black we show both Adam (top line) and NAdamW (bottom line) baselines tuned to the task being tested on. Error bars denote standard error across different random seeds when evaluating a given set of learned optimizer weights. Note how there is very little variation in evaluation of meta-loss, but large variation between different meta-iterations. In this pane we meta-train each learned optimizer on **CIFAR-10 ConvNet**.

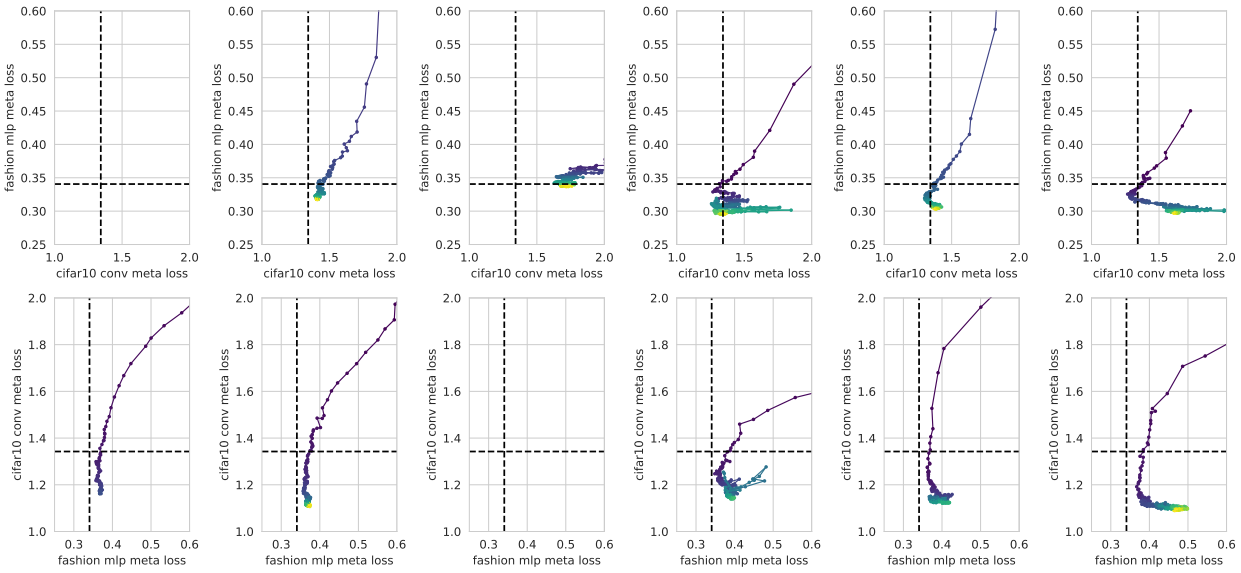


Figure 11: **small_fc_lopt Meta-generalization**: We show meta-training and meta-test performance plotted over the course of meta-training. In the top row we show meta-training on Fashion MNIST and meta-testing on the CIFAR-10 ConvNet. In the bottom we show meta-training on the CIFAR-10 ConvNet and meta-testing on Fashion MNIST MLP. Each column represents a random seed or learning rate – the 3 left most columns are a smaller learning rate than the 3 right most columns. An empty plot indicates the model for the given seed did not converge. Purple is earlier in training, yellow is late in meta-training. We see meta-overfitting in all cases denoted by the C shaped curves.

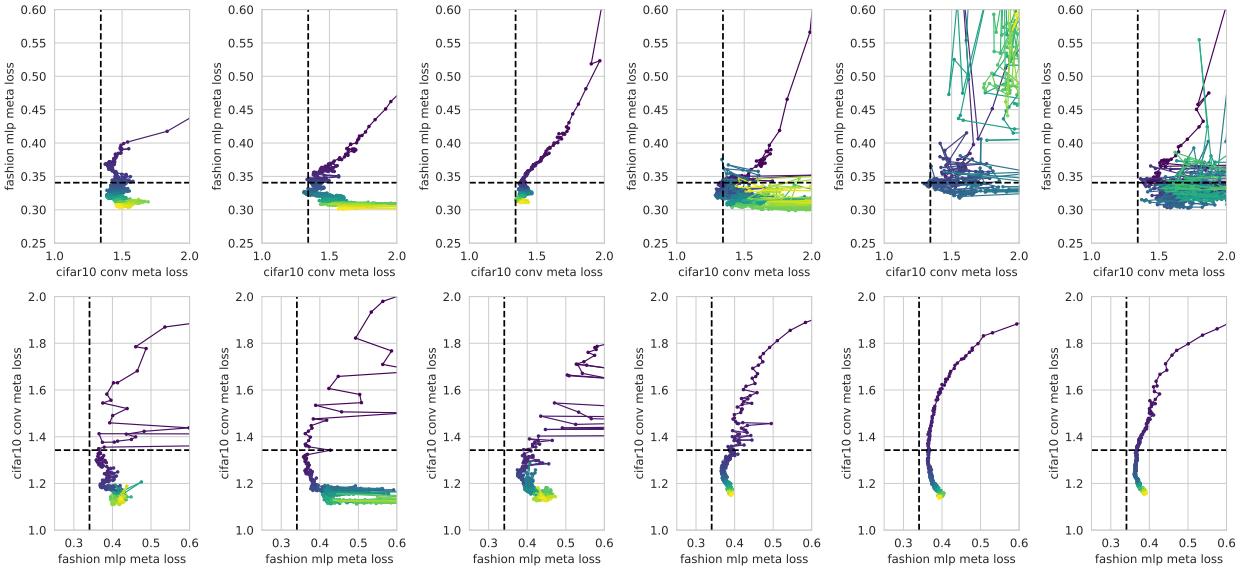


Figure 12: **rnn_fc_lopt Meta-generalization**: We show meta-training and meta-test performance plotted over the course of meta-training. In the top row we show meta-training on Fashion MNIST and meta-testing on the CIFAR-10 ConvNet. In the bottom we show meta-training on the CIFAR-10 ConvNet and meta-testing on Fashion MNIST. Each column represents a random seed or learning rate – the 3 left most columns are a smaller learning rate than the 3 right most columns. An empty plot indicates the model for the given seed did not converge. Purple is earlier in training, yellow is late in meta-training. We see meta-overfitting in all cases denoted by the C shaped curves.

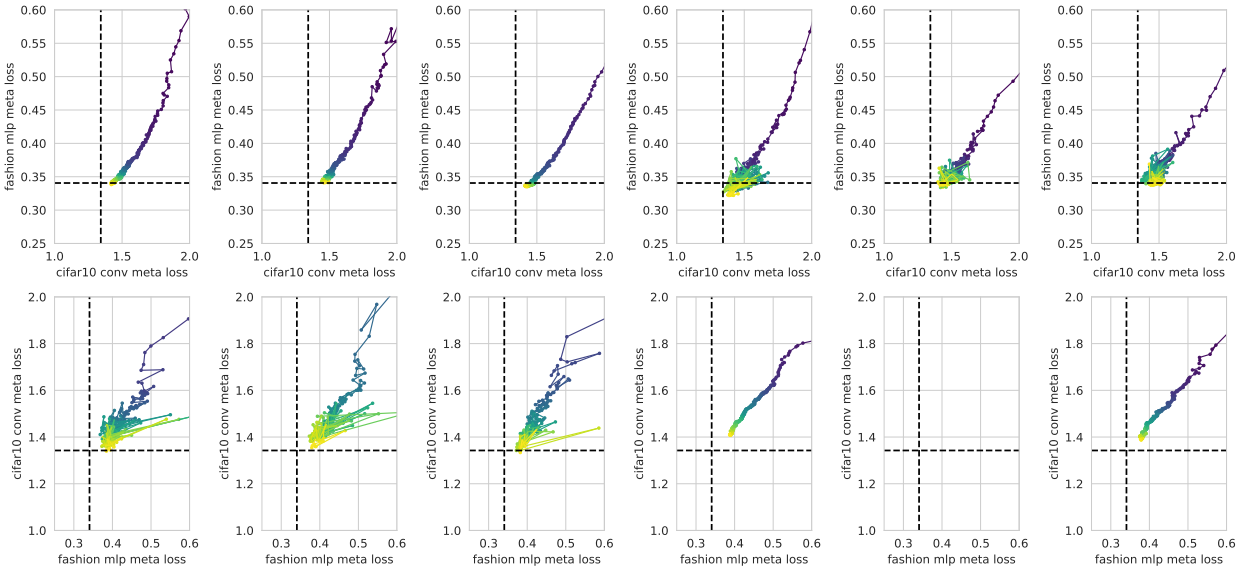


Figure 13: **fc_lopt Meta-generalization**: We show meta-training and meta-test performance plotted over the course of meta-training. In the top row we show meta-training on Fashion MNIST and meta-testing on the CIFAR-10 ConvNet. In the bottom we show meta-training on the CIFAR-10 ConvNet and meta-testing on Fashion MNIST. Each column represents a random seed. Purple is earlier in training, yellow is late in meta-training. Here we see less meta-overfitting likely due to the learned optimizer not fully fitting the meta-training distribution. The empty plot denotes a learned optimizer which did not converge.

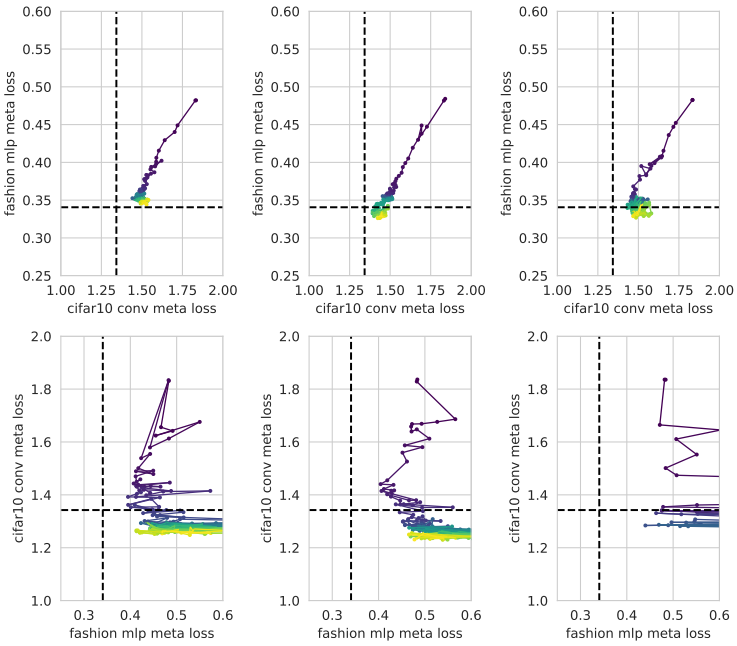


Figure 14: **nn_adam Meta-generalization**: We show meta-training and meta-test performance plotted over the course of meta-training. In the top row we show meta-training on fashion MNIST and meta-testing on the CIFAR-10 ConvNet. In the bottom we show meta-training on the CIFAR-10 ConvNet and meta-testing on Fashion MNIST MLP. Each column represents a random seed. The fifth figure on the second row is empty as meta-training diverged. Purple is earlier in training, yellow is late in meta-training.

# F NADAMW UPDATE EQUATIONS AND SEARCH SPACE

For our NAdamW baseline, we use the same implementation, and search space described in Metz et al. (2020b). We repeat the functional form here for convenience.

## F.1 UPDATE EQUATIONS

This optimizer architecture has 10 hyperparameters. The base learning rate, $\alpha_{base}$, first and second moment momentum, $\beta_1$, $\beta_2$, the numerical stability term, $\epsilon$, $\ell_{2WD}$ $\ell_2$ regularization strength, $\ell_{2AdamW}$ AdamW style weight decay, and a boolean to switch between NAdam and Adam, $b_{\text{use nesterov}}$. The learning rate schedule is based off of a single cycle cosine decay with a warmup. It is controlled by 3 additional parameters – $c_{\text{warmup}}$, $c_{\text{constant}}$, and $c_{\text{min learning rate mult}}$.

The learning rate is defined by:

$$
\begin{aligned}
u =& c_{\text{warmup}} T > t && (4)\\
\alpha_{\text{decay\&constant}} =& (\alpha_{base} - c_{\text{min learning rate mult}})(0.5 && (5)\\
& \cos(t\pi/(T - c_{\text{constant}})) + 0.5) + && (6)\\
& c_{\text{min learning rate mult}} && (7)\\
\alpha_{\text{warmup}} =& \frac{t}{(T c_{\text{warmup}})} && (8)\\
\alpha =& (1-u)\alpha_{\text{decay\&constant}} + u\alpha_{\text{warm}} && (9)
\end{aligned}
$$

The update equations of NAdamW follow.

$$
\begin{aligned}
\phi^{(0)} =& \text{problem specified random initialization} && (10)\\
m^{(0)} =& 0 && (11)\\
v^{(0)} =& 0 && (12)\\
g^{(t)} =& \frac{d}{d\phi^{(t)}}(f(x;\phi^{(t)}) + \ell_{2wd}||\phi^{(t)}||_2^2) && (13)\\
m^{(t)} =& \beta_1 m^{(t-1)} + g^{(t)}(1-\beta_1) && (14)\\
v^{(t)} =& \beta_2 v^{(t-1)} + (g^{(t)})^2(1-\beta_2) && (15)\\
\hat{m}^{(t)} =& \frac{m^{(t)}}{1-\beta_1^{t+1}} && (16)\\
\hat{v}^{(t)} =& \frac{v^{(t)}}{1-\beta_2^{t+1}} && (17)\\
u^{(t)}_{\text{heavy ball}} =& \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} && (18)\\
u^{(t)}_{\text{nesterov}} =& \frac{\beta_1 \hat{m}^{(t)} + (1-\beta_1) g^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} && (19)\\
\phi^{(t+1)} =& \phi^{(t)} - (1 - b_{\text{use nesterov}})\alpha u^{(t)}_{\text{heavy ball}} + && (20)\\
& b_{\text{use nesterov}} \alpha u^{(t)}_{\text{nesterov}} - \alpha \ell_{2AdamW} \phi^{(t)} && (21)
\end{aligned}
$$

## F.2 HYPERPARAMETER SEARCH SPACE

The initial learning rate, $\alpha_{base}$ is sampled from log space between $1e-5$ and 1.0. $1-\beta_1$ is sampled logarithmically between $1e-3$, and 1.0. $1-\beta_2$ is sampled between $1e-5$, and 1.0. $\epsilon$ is sampled logarithmically between $1e-8$ and $1e4$. We sample using nesterov ($b_{\text{use nesterov}}$) 50% of the time. We sample $\ell_{2WD}$ and $\ell_{2AdamW}$ logarithmically between $1e-5$ and $1e-1$. Equal probabilities of a third we either use both terms, zero out $\ell_{2WD}$, or zero out $\ell_{2AdamW}$. With 50% probability we use a nonzero min learning rate multiplier sampled logarithmically between $1e-5$ and 1.0. With 50% probability we sample the warm up fraction, $c_{\text{warmup}}$ between 1e-5 and 1e-1, otherwise it is set to zero. Finally, we uniformly sample the amount of time the learning rate is held constant ($c_{\text{constant}}$) between 0 and 1.