A. Derivation of the unrolled gradient

For the case of learning a learning rate, we can derive the unrolled gradient to get some intuition for issues that arise with outer-training. Here, the update rule is given by:

$$w \leftarrow w - \theta \nabla \ell(w),$$

where w are the inner-parameters to train, ℓ is the inner-loss, and θ is a scalar learning rate, the only outer-parameter. We use superscripts to denote the iteration, so $w^{(t)}$ are the parameters at iteration t. In addition, we use $g^{(t)} = \nabla \ell(w^{(t)})$ and $H^{(t)} = \nabla^2 \ell(w^{(t)})$ to denote the gradient and Hessian of the loss at iteration t, respectively.

We are interested in computing the gradient of the loss after T steps of gradient descent with respect to the learning rate, θ. This quantity is given by $\frac{\partial d\ell}{\partial \theta} = \langle g^{(T)}, \frac{dw^{(T)}}{d\theta} \rangle$. The second term in this inner product tells us how changes in the learning rate affect the final parameter value after T steps. This quantity can be defined recursively using the total derivative:

$$\begin{aligned} \frac{dw^{(T)}}{d\theta} &= \frac{\partial w^{(T)}}{\partial w^{(T-1)}} \frac{dw^{(T-1)}}{d\theta} - \frac{\partial w^{(T)}}{\partial \theta} \\ &= \left(I - H^{(T-1)}\right) \frac{dw^{(T-1)}}{d\theta} - g^{(T-1)} \end{aligned}$$

By expanding the above expression from $t = 1$ to $t = T$, we get the following expression for the unrolled gradient:

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

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.

This expression makes intuitive sense if we restrict the number of unrolled steps to one. In this case, the unrolled gradient is the negative inner product between the current and previous gradients: $\frac{d\ell(w^{(T)})}{d\theta} = -\langle g^{(T)}, g^{(T-1)} \rangle$. This means that if the current and previous gradients are correlated (have positive inner product), then updating the learning rate in the direction of the negative unrolled gradient means that we should *increase* the learning rate. This makes sense as if the current and previous gradients are correlated, we expect that we should move faster along this direction.

B. Exploding Gradients on 1D Loss Surfaces

In addition to varying momentum, shown in Figure 2 we also explore the effects of varying learning rate in Figure 9.

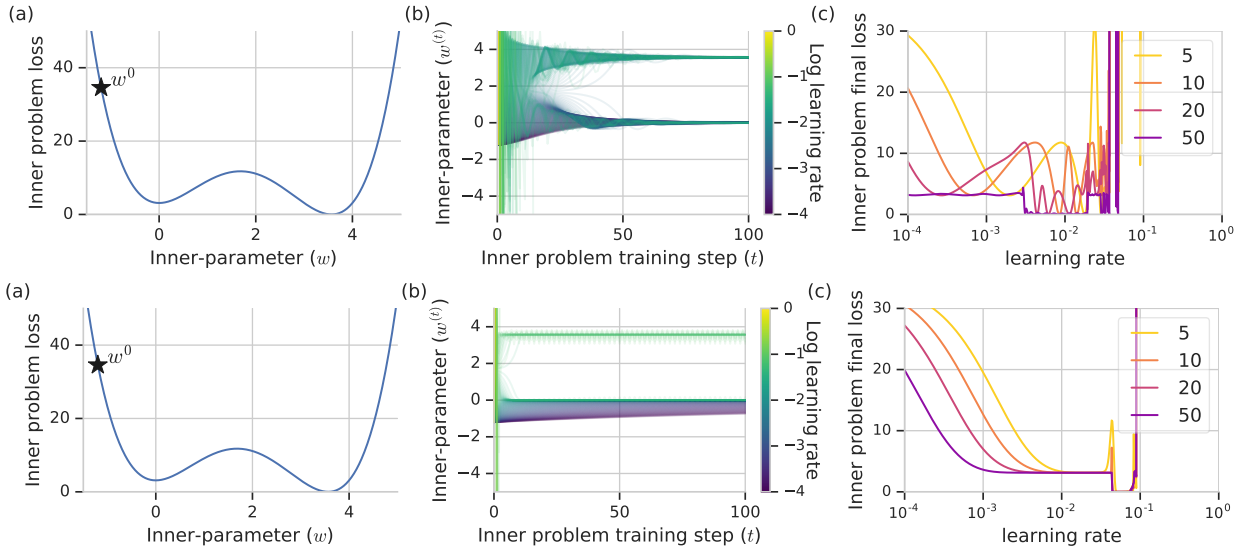

Figure 9. Extension of 2 showing varying learning rate with a fixed momentum (top), and varying learning rate with no momentum (bottom). Outer-problem optimization landscapes can become increasingly pathological with increasing inner-problem step count. **(a)** A toy 1D inner-problem loss surface with 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. The top row SGD+momentum. Color denotes different values of the optimizer's learning rate parameter. In both settings, low learning rates converge to the first local minimum at $w = 0$. Slightly larger learning rates escape this minimum to settle at the global minimum ($w \approx 3.5$). Even larger values (purples) oscillate before eventually diverging. **(c)** The final loss after some number steps of optimization as a function of the learning rate. Larger values of T result in near discontinuous loss surfaces around the transition points between the two minima.

C. Outer-Training Algorithm

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

Initialize outer-parameters (θ).
while Outer-training, for each parallel worker **do**
  Sample a dataset $\mathcal{D}$, from task distribution $\mathcal{T}$.
  Initialize the inner loop parameters $w^{(0)}$ randomly.
  for Each truncation, t, 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 iteration.
    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 with both a positive, and negative antithetic sample (L^+, L^-) using the 2 sequences of w from t to $t + k$ 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 current inner-parameter w from one of the two sequences with the inner-parameter value from the end of the truncation ($w^{(t+k)}$).
  end for
end while
while Outer-training **do**
  When a batch of gradients is available, compute empirical variance and empirical mean of each weight for each estimator.
  Use equation 1 to compute the combined gradient estimate.
  Update outer-parameters with SGD: $\theta \leftarrow \theta - \alpha g_{combined}$ where α is a learning rate.
end while

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 at a few different decay values: [0.5, 0.9, 0.99, 0.999, 0.9999]. A similar idea has been explored in (Lucas et al., 2018). The current gradient as well as the current weight value are also used as features (2 additional 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).

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 (e.g. batch norm) 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 scaling 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. Code for this optimizer can be found at https://github.com/google-research/google-research/tree/master/task_specific_learned_opt.

D.2. 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 (Glorot & Bengio, 2010). At the last convolutional layer, an average pool is performed, and a linear projection is applied to get the 10 output classes.

D.3. Outer-Training

We train using the algorithm described in Appendix C 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 (Kingma & Ba, 2014) 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 (Arjovsky et al., 2017)). 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. See Figure 6. Additionally, we supply 4 additional baselines: RMSProp+Reg+Decay hyper parameter searched for validation and training loss over learning rate, learning rate schedule (both exponential and linear), epsilon, and l1/l2 regularization using 1000 random configurations, and SGDMom+Reg+Decay hyper parameter searched for validation and training loss over learning rate, learning rate schedule, momentum, to use Nesterov momentum, and l1/l2 regularization using 1000 random configurations. The search procedure and search space matches the existing Adam+Reg+Decay procedure described in 4.4.

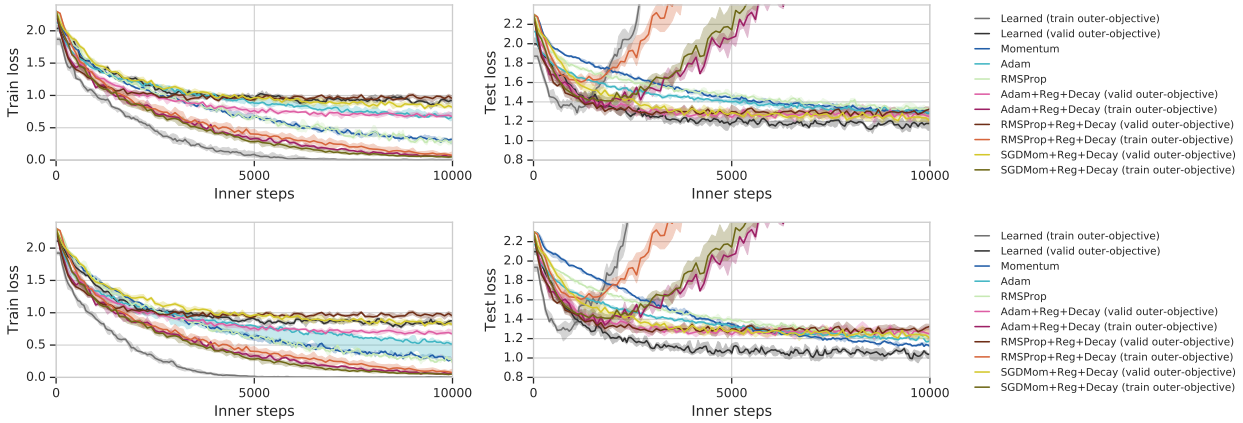

Figure 10. Additional outer-validation problems.

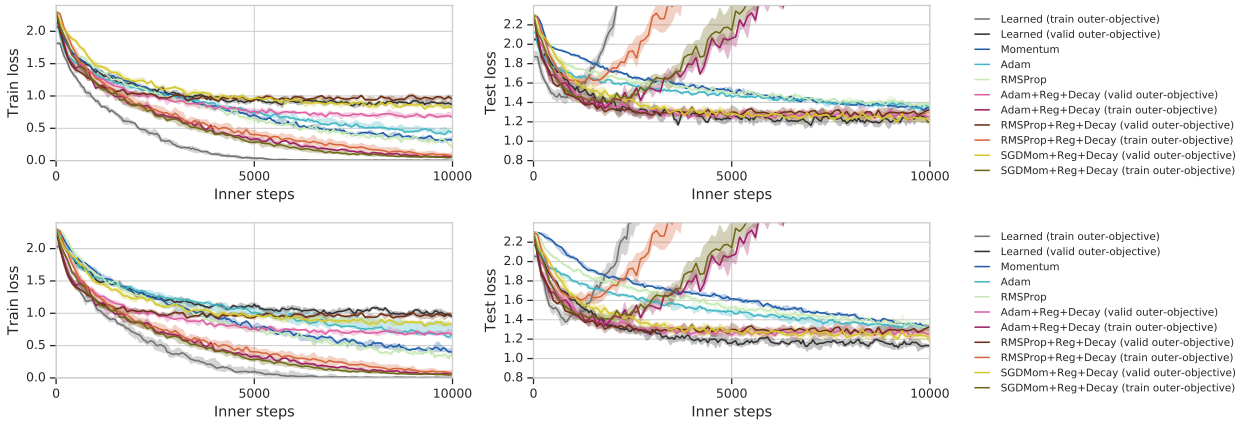

Figure 11. Outer-training problems.

F. Out of domain generalization

In this work, we focus our attention to learning optimizers over a specific task distribution (3 layer convolutional networks trained on ten class subsets of 32x32 Imagenet). In addition to testing on these in domain problems (Appendix E), we test our learned optimizer on a variety of out of domain target problems. Despite little variation in the outer-training task distribution, our models show promising generalization when transferred to a wide range of different architectures (fully connected, convolutional networks) depths (2 layer to 6 layer) and number of parameters (models roughly 16x more parameters). We see these as promising sign that our learned optimizer has a reasonable (but not perfect) inductive bias. We leave training with increased variation to encourage better generalization as an area for future work.

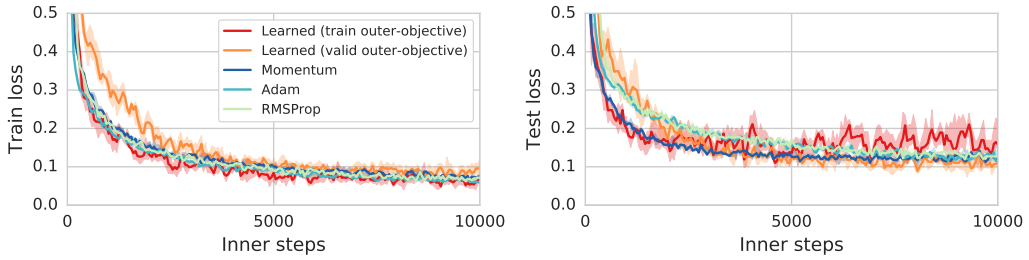

Figure 12. Inner problem: 2 hidden layer fully connected network. 32 units per layer with ReLU activations trained on 14x14 MNIST.

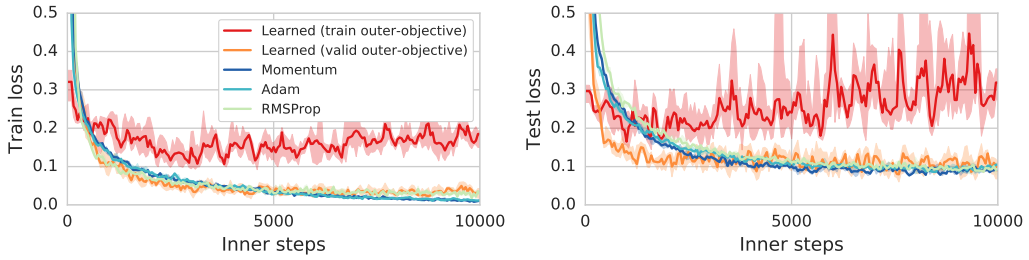

Figure 13. Inner problem: 3 hidden layer fully connected network. 128 units per layer with ReLU activations trained on 14x14 MNIST.

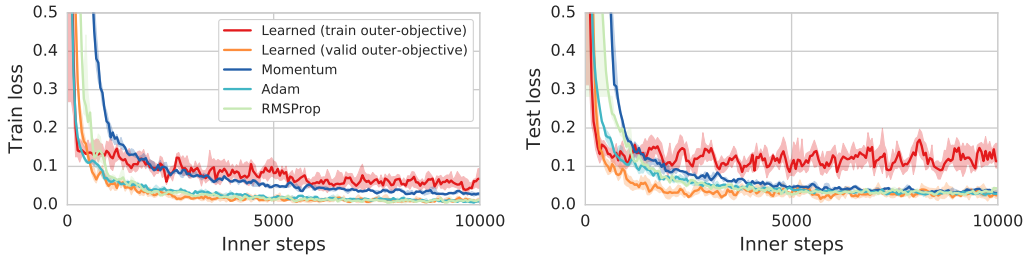

Figure 14. Inner problem: 6 convolutional layer network. 32 units per layer, strides: [2,1,2,1,1,1] with ReLU activations on 28x28 MNIST.

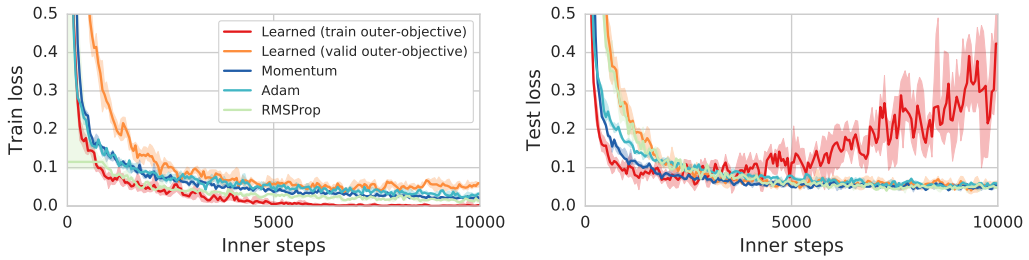

Figure 15. Inner problem: 3 convolutional layer network. 32 units per layer, strides: [2,2,1] with ReLU activations on 28x28 MNIST.

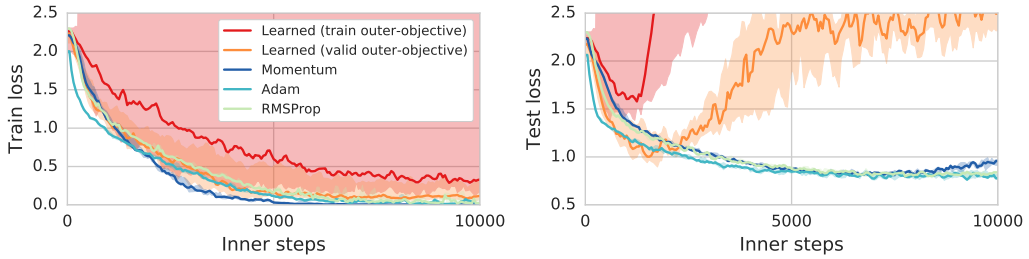

Figure 16. Inner problem: 3 convolutional layer network. 128 units per layer, strides: [2,2,1] with ReLU activations trained on a ten-way classification sampled from 32x32 Imagenet (using holdout classes).

G. Inner-loop training speed

When training models, often one cares about taking less wall-clock time as compared to loss decrease 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 *features*x32, 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 parameter count. 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 re-implement Adam, SGD, and our learned optimizer in TensorFlow (no fused ops) 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 exceed the slowdown, resulting in an optimizer that is still considerably faster when measured in wall-clock time.

For the wall-clock figures presented in this paper we rescale the step vs performance curves by the steps per second instead of directly measuring wall-clock time. When running evaluations, we perform extensive logging which dominates the total compute costs.

H. Ablation learning curves

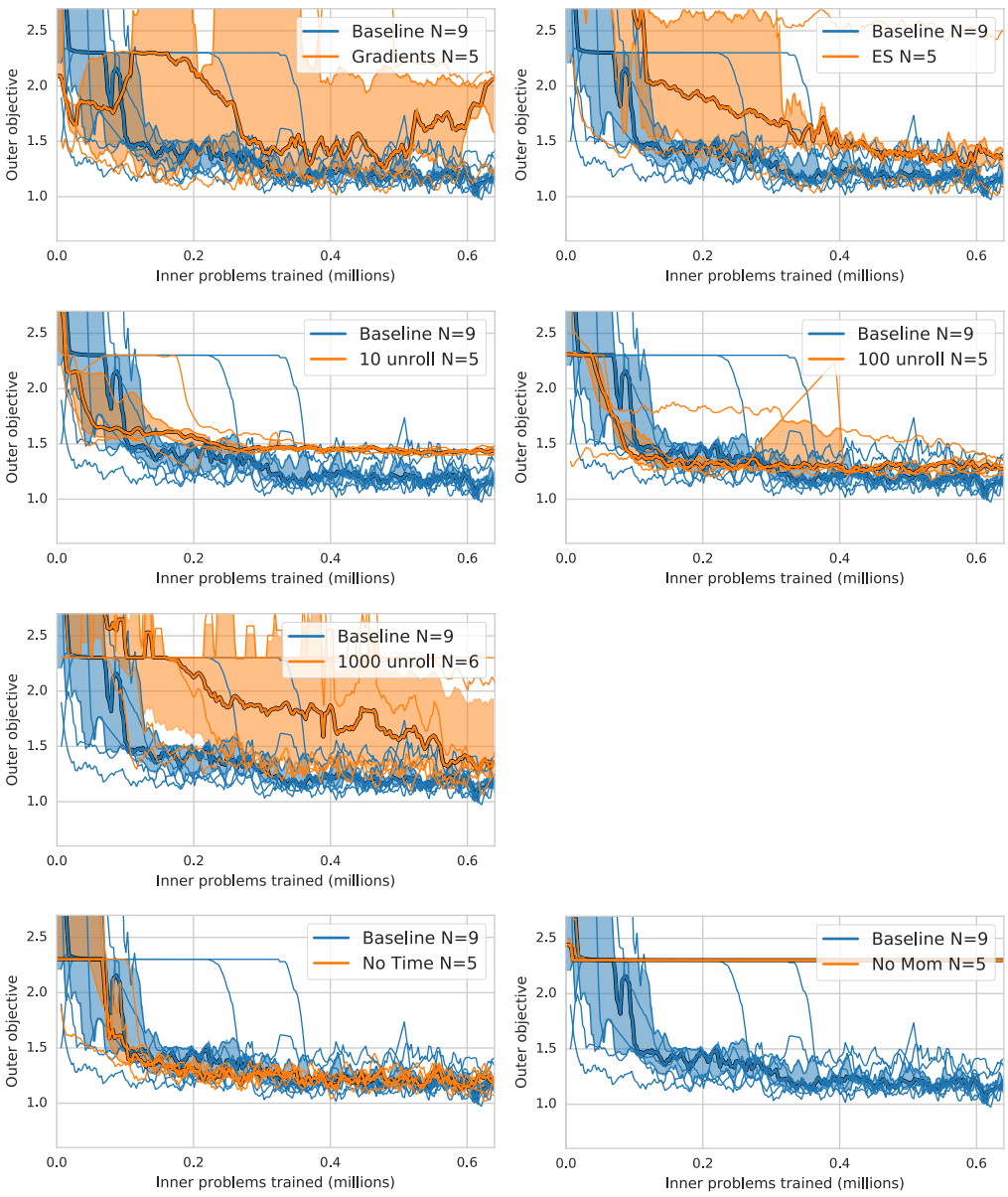

Figure 17. Training curves for ablations described in §4.5. The thick line bordered in black is the median performance, with the shaded region containing the 25% and 75% percentile. Thinner solid lines are individual runs.

I. Additional Truncation Bias Experiments

In §2.4 we show the effect of truncation bias when learning Adam hyper parameters using the Adam outer-optimizer. When using truncated gradients, the directions passed into this outer-optimizer are not well behaved, and are not even guaranteed to be a conservative vector field. As such, different outer optimizers might behave differently. To test this, we test multiple configurations of outer-optimizer in figure 18.

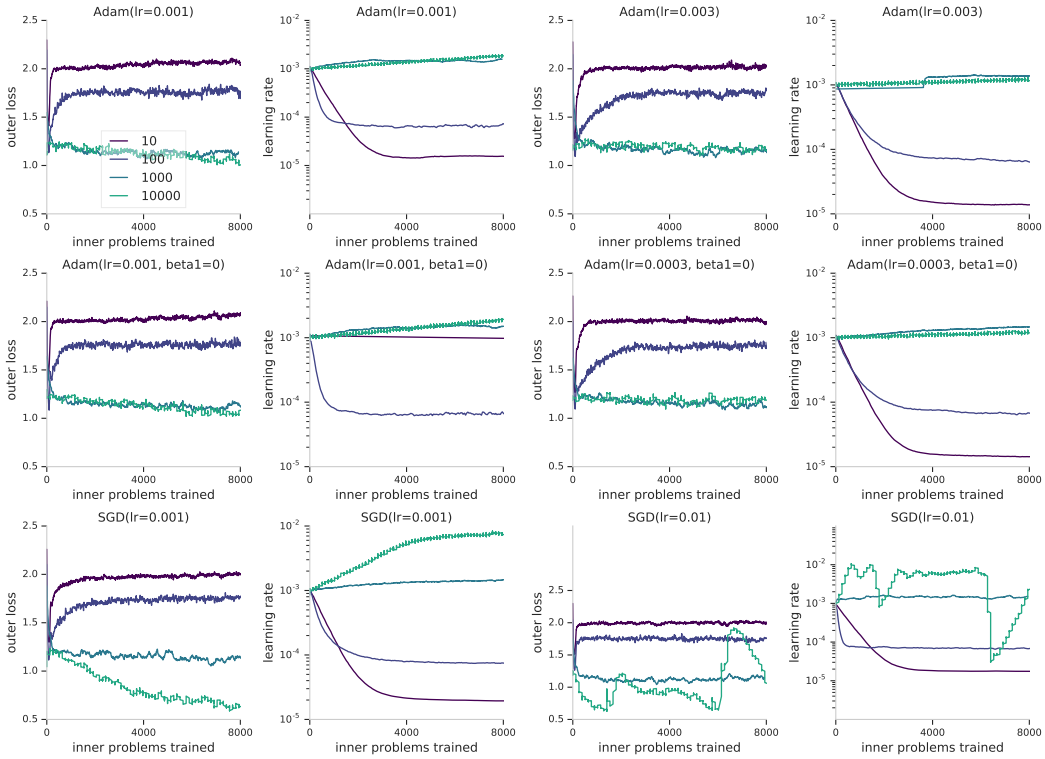

Figure 18. Additional experiments showing truncation bias when attempting to learn Adam hyper-parameters with different outer-optimizers. We show 4 configurations of Adam (2 learning rates, with and without beta1), and 2 configurations of SGD (2 different learning rates) with outer-loss and learning rate plotted for each. In all experiments we see similar, and significant truncation bias when using low amounts of steps per truncation. For some of the higher learning rate experiments (e.g. SGD with lr=0.01), we see diverging loss.

J. Batch-Normed Base Model

In this section, we present experiments targeting a different task family. In particular, we add batch normalization to the convolution layers. We employ the same meta-training procedure and same learned optimizer architecture as used in the rest of this paper and target the validation loss outer-objective.

Meta-training curves can be found in Figure 19. We find we outperform learning rate tuned Adam, but do not out perform the 8 parameter tuned Adam baseline. At this point, we are unsure the source of this gap but suspect hyper parameter tuning would improve this result.

J.1. Longer Unrolls

All of the learned optimizers presented in this paper are outer-trained using 10k inner-iterations. In Figure 20 we show an application of a learned optimizer outside of the outer-training regime – up to 100k inner iterations using the learned optimizer shown in orange in figure 19. Unlike hand designed optimizers, our learned optimizer does not completely minimize the training loss. As a result, the test performance remains consistent far outside the outer-training regime.

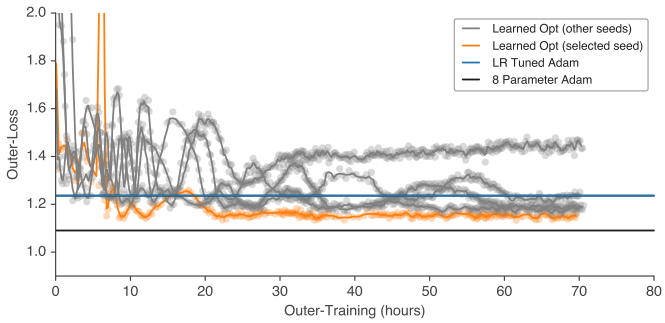

Figure 19. Outer-training curves for 5 different random seeds. The learned optimizers outperform LR tuned Adam on 4/5 random seeds but do not yet out perform the 8 parameter Adam baseline.

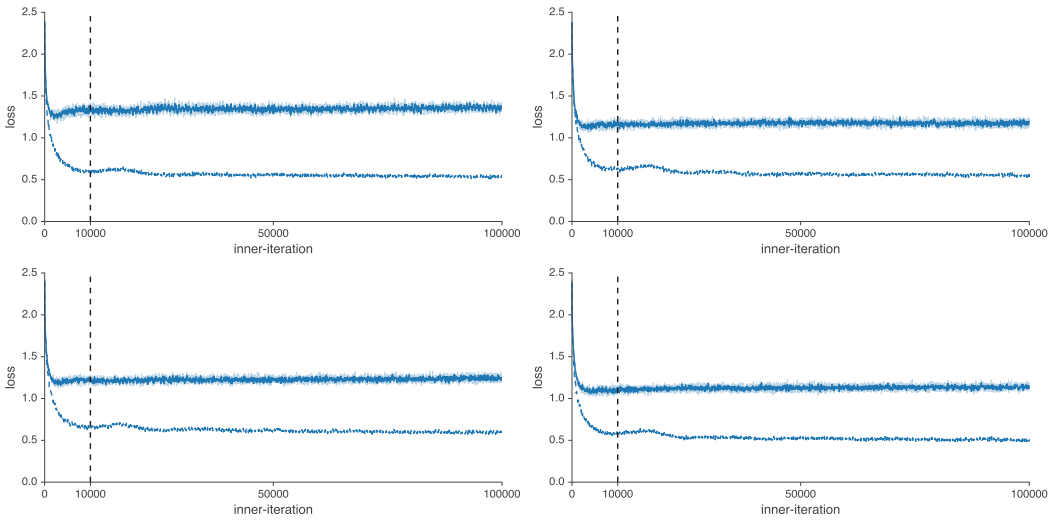

Figure 20. Example unrolls on held out outer-tasks when inner-trained for 10x more inner steps. Each panel represents a different held out task. The dashed vertical line denotes the max number of steps seen at outer-training time. The solid line shows inner-test performance, where as the dashed denotes inner-train. We find that our learned optimizers keep a consistent test and train loss even after the 10k iterations used for outer-training.