Gradient Descent: The Ultimate Optimizer

Anonymous Author(s)
Affiliation
Address
email

Abstract

Working with any gradient-based machine learning algorithm involves the tedious task of tuning the optimizer's hyperparameters, such as the step size. Recent work has shown how the step size can itself be "learned" on-line by gradient descent, by manually deriving expressions for "hypergradients" ahead of time.

We show how to *automatically* compute hypergradients with a simple and elegant modification to backpropagation. This allows us to apply the method to other hyperparameters besides the step size, such as the momentum coefficient. We can even recursively apply the method to its own *hyper*-hyperparameters, and so on *ad infinitum*. As these towers of optimizers grow taller, they become less sensitive to the initial choice of hyperparameters. We present experiments validating this for MLPs, CNNs, and RNNs.

1 Introduction

When we train deep neural networks by gradient descent, we have to select a step size α for our optimizer. If α is too small, the optimizer runs very slowly, whereas if α is too large, the optimizer fails to converge. Choosing an appropriate α is thus itself an optimization task that machine learning practitioners face every day. Why not apply gradient descent here, too? To do so, we need to compute the derivative of the loss function not only with respect to the neural network's weights, but also with respect to α. Baydin et al. (2018), applying an insight from Almeida et al. (1999), describe how to efficiently compute such "hypergradients" by manually differentiating standard optimizer update rules with respect to the step size hyperparameter. This allows for on-line learning rate adaptation, which generally improves convergence, especially if the initial α is sub-optimal.

This method has three limitations: (1) manual differentiation is tedious and error-prone, and must be re-done for each optimizer variant; (2) the method only tunes the step size hyperparameter, not other hyperparameters such as the momentum coefficient; (3) the method introduces a *new* hyperparameter, the hyper-step-size, which must also be tuned.

In this paper, we address all three limitations by replacing manual differentiation with *automatic differentiation* (AD), which (1) automatically computes correct derivatives without any additional human effort, and (2) naturally generalizes to other hyperparameters (e.g. momentum coefficient) for free. As for (3), we observe that AD can be applied to optimize not only the hyperparameters, but also the *hyper*-hyperparameters, and the hyper-hyper-hyperparameters, and so on. In fact, we can implement arbitrarily tall towers of recursive optimizers, which are increasingly robust to the choice of initial hyperparameter. These "hyperoptimizers" therefore reduce the burden on humans responsible for tuning the hyperparameters. (This was hypothesized by Baydin et al., but not implemented because manual differentiation of complex sequences of nested optimizers is impractical.)

Although "just apply AD" is a seemingly straightforward recipe, an efficient implementation that properly allows for self-application requires some care. To close the loop, we take inspiration from the study of recursion and combinators in programming language theory (and the long tradition

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

of programming language papers named "Lambda: The Ultimate X"). We spell out the details in Section 2 and evaluate our method in Section 3. We find that across a variety of architectures (MLPs, CNNs, and RNNs) our hyperoptimizers are robust to suboptimal choices of initial hyperparameters, and that this robustness increases as we grow the stacks of optimizers taller.

2 Implementing hyperoptimizers

Consider some loss function f that we want to minimize using gradient descent, and let w_i be the weights at the beginning of step i. First, recall the standard weight update rule at step i for SGD, using some (fixed) step size α:

$$w_{i+1} = w_i - \alpha \frac{\partial f(w_i)}{\partial w_i}$$

We would like to update α as well at each step, so we will index it with the step number: let α_i be the step size at the beginning of step i. At each step, we will first update α_i to α_{i+1} using some update rule yet to be derived, and then use the updated step size α_{i+1} to update the weights from w_i to w_{i+1}.

$$\alpha_{i+1} = \alpha_i - \boxed{\text{adjustment for } \alpha_i}$$
$$w_{i+1} = w_i - \alpha_{i+1} \frac{\partial f(w_i)}{\partial w_i}$$

What should the adjustment for α_i be? By analogy to w, we want to adjust α_i in the direction of the gradient of the loss function with respect to α_i, scaled by some *hyper*-step size κ. In other words, the adjustment should be $\kappa(\partial f(w_i)/\partial \alpha_i)$. Our modified update rule is therefore:

$$\alpha_{i+1} = \alpha_i - \kappa \frac{\partial f(w_i)}{\partial \alpha_i} \tag{1}$$
$$w_{i+1} = w_i - \alpha_{i+1} \frac{\partial f(w_i)}{\partial w_i} \tag{2}$$

All that remains is to compute $\partial f(w_i)/\partial \alpha_i$ in equation (1). Below, we first review how Baydin et al. (2018) take this derivative by hand. Then, we show how to obtain the same result automatically and efficiently using AD. Finally, we discuss how this automation allows us to generalize our approach in many different directions.

2.1 Computing the step-size update rule by hand

One option to compute $\partial f(w_i)/\partial \alpha_i$, explored by Baydin et al. (2018), is to proceed by direct manual computation of the partial derivative. Applying the chain rule to (1), we have

$$\frac{\partial f(w_i)}{\partial \alpha_i} = \frac{\partial f(w_i)}{\partial w_i} \cdot \frac{\partial w_i}{\partial \alpha_i} = \frac{\partial f(w_i)}{\partial w_i} \cdot \frac{\partial \left(w_{i-1} - \alpha_i \frac{\partial f(w_{i-1})}{\partial w_{i-1}} \right)}{\partial \alpha_i} \tag{3}$$
$$= \frac{\partial f(w_i)}{\partial w_i} \cdot \left(-\frac{\partial f(w_{i-1})}{\partial w_{i-1}} \right) \tag{4}$$

where (3) is obtained by substituting the update rule in (2) for w_i and (4) is obtained by observing that w_{i-1} and $f(w_{i-1})$ do not depend on α_i. As Baydin et al. note, this expression lends itself to a simple and efficient implementation: simply remember the past two gradients from backpropagation, and take their dot product to obtain the hypergradient with respect to the step size.

We were able to take this derivative by hand because the update rule for SGD is simply a multiplication by a constant, whose derivative is trivial. What about other optimizers? Consider the Adam optimizer (Kingma and Ba, 2014), which has a much more sophisticated update rule involving the four hyperparameters $\alpha, \beta_1, \beta_2, \epsilon$ (though ϵ is typically not tuned). Differentiating the update rule by hand,

we obtain *significantly* more complex expressions for the hypergradients:

$$\frac{\partial w_i}{\partial \alpha_i} = -\frac{\hat{m}_i}{\left(\epsilon_i + \sqrt{\hat{v}_i}\right)} \qquad \frac{\partial w_i}{\partial \beta_{1_i}} = -\frac{\alpha_i \left(-\frac{\partial f(w_{i-1})}{\partial w_{i-1}} + m_{i-1} + i\beta_{1_i}^{(i-1)}\hat{m}_i\right)}{\left(1 - \beta_{1_i}^{i}\right)\left(\epsilon_i + \sqrt{\hat{v}_i}\right)}$$

$$\frac{\partial w_i}{\partial \epsilon_i} = \frac{\alpha_i \hat{m}_i}{\left(\epsilon_i + \sqrt{\hat{v}_i}\right)^2} \qquad \frac{\partial w_i}{\partial \beta_{2_i}} = \frac{\alpha_i \hat{m}_i \sqrt{\hat{v}_i}\left(-\left(\frac{\partial f(w_{i-1})}{\partial w_{i-1}}\right)^2 + v_{i-1} + i\beta_{2_i}^{(i-1)}\hat{v}_i\right)}{2v_i\left(\epsilon_i + \sqrt{\hat{v}_i}\right)^2}$$

This manual approach to derive hypergradients simply does not scale: it is tedious and error-prone, and must be repeated for every optimizer variant. However, with a little bit of care, we can compute hypergradients automatically and efficiently alongside the regular gradients.

2.2 Computing the step-size update rule automatically

In order to compute hypergradients automatically, let us first briefly review the mechanics of reverse-mode AD. Frameworks that provide reverse-mode AD typically build up a computation graph as the function is computed forwardly. For example, when a user computes the function $f(w_i)$, the framework internally stores a DAG whose leaves are the weights w_i, whose internal nodes are intermediate computations, and whose root is the concrete loss function. The framework can then backpropagate from the computation graph created for this root node, depositing gradients in each node as it descends, until the weights w_i at the leaf nodes have accumulated their gradients $\partial f(w_i)/\partial w_i$. Once the gradient $\partial f(w_i)/\partial w_i$ is computed by the backwards pass, we update the weights $w_{i+1} = w_i - \alpha \cdot \partial f(w_i)/\partial w_i$, and repeat the cycle for the next step of gradient descent.

An important consideration at this point is for the weights to be "detached" from the computation graph before each iteration of this algorithm — that is, for the weights to be forcibly converted to leaves of the graph by removing any inbound edges. The effect of the "detach" operation is depicted in Figure 1a. If this step were skipped, backpropagation at the next iteration would continue beyond the current weights and into the past. Over time the computation graph would grow taller linearly in the number of steps taken; because backpropagation is linear in the size of the graph, the overall training would become quadratic-time and intractable.

In the built-in optimizer provided by PyTorch (Paszke et al., 2017, optim/sgd.py, commit `ff94c9d`), this is implemented by wrapping the update in the `@torch.no_grad()` context manager. Here, we need finer grained control over gradient flow, so will make the `.detach()` operations explicit. Below is pseudocode for an SGD optimizer that uses `.detach()` as we have discussed. The highlighted calls to `.detach()` correspond to detaching the weights and their gradients.

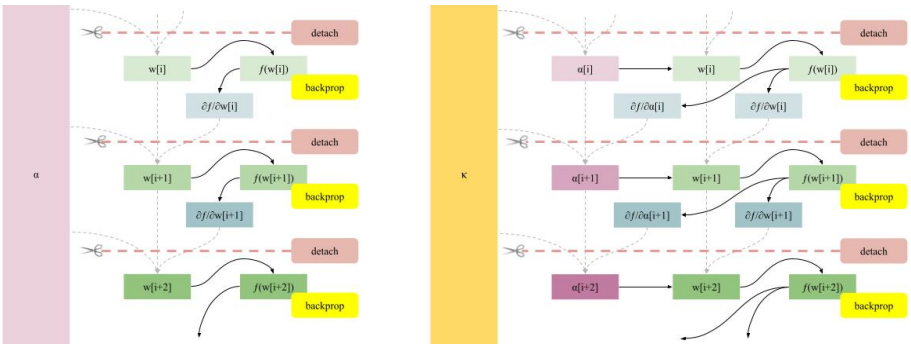
```
def SGD.__init__(self, alpha):
  self.alpha = alpha

def SGD.adjust(w):
  d_w = w.grad.detach()
  w = w.detach() - self.alpha.detach() * d_w
```

Now, in order to have backpropagation deposit the gradient with respect to α_i as well as w_i, we can simply refrain from detaching α_i from the graph, detaching instead *its* parents. This is depicted in Figure 1b. Because we want to compute $\partial f(w_i)/\partial \alpha_i$, the edge from α_i to w_i needs to remain intact. To implement this, instead of calling `.detach()` on `alpha` directly, we detach its parents when applying equation (1). This yields the following fully-automated hyperoptimization algorithm:

```
def HyperSGD.adjust(w):
  # update alpha using Equation (1)
  d_alpha = self.alpha.grad.detach()
  self.alpha = self.alpha.detach() - kappa.detach() * d_alpha

  # update w using Equation (2)
  d_w = w.grad.detach()
  w = w.detach() - self.alpha.detach() * d_w
```



(a) Computation graph of SGD with a single fixed hyperparameter α.

(b) Computation graph of SGD with a continuously-updated hyperparameter α_i.

Figure 1: Visualizing the computation graphs of SGD and HyperSGD.

Since we only extend the computation graph by a little extra amount (corresponding to evaluating the optimizer), the hyperoptimizer's computational overhead is negligible.

2.3 Extending to other optimizers

As suggested by Maclaurin et al. (2015), it should be possible to apply gradient-based methods to tune hyperparameters of common variations on SGD such as AdaGrad (Duchi et al., 2011), AdaDelta (Zeiler, 2012), or Adam (Kingma and Ba, 2014). The above implementation of HyperSGD generalizes quite easily to these optimizers — we simply replace the last line with the Adam update rule.

Unlike previous work, our method also allows for simultaneously optimizing all hyperparameters of Adam (not just step size), by treating them just like `alpha` in the implementation. Our evaluation in Section 3.2 demonstrates that this indeed useful to do. There are, however, two important subtleties: First, because the hyperparameters β_1 and β_2 must be strictly in the domain $(0, 1)$, we clamp the "raw" values to this domain using a scaled sigmoid. Without this step, we might accidentally adjust these values outside their domains. Second, the Adam optimizer involves the term $\sqrt{\hat{v}_i}$, which is continuous but not differentiable at $\hat{v}_i = 0$. Because Adam normally initializes $v_0 = 0$, backpropagation fails on the very first step due to division by zero. We fix this problem by initializing v_0 to ϵ rather than 0.

2.4 Stacking Hyperoptimizers Recursively

At this point it is natural to ask whether the hyperoptimizer can itself be optimized; that is, whether the hyper-hyperparameters can be adjusted by a hyper-hyperoptimizer. The possibility of doing so recursively *ad infinitum* to obtain an optimization algorithm that is highly robust to the human-chosen hyper-parameter was hypothesized by Baydin et al. (2018). Computing the gradients of these higher-order hyperparameters by hand is impossible without knowing the exact sequence of stacked optimizers ahead of time, and as discussed above, would be extremely tedious and error-prone.

However, the ability to compute these gradients automatically by AD makes it possible to realize this vision. To do so, let us revisit our previous implementation of HyperSGD. Notice that there is an opportunity for recursion lurking here: the adjustment to `alpha` can be factored out with a call to `SGD.adjust`, where SGD's hyperparameter is `kappa`.

```
def HyperSGD.adjust(w):
  # update alpha using Equation (1)
  SGD(kappa).adjust(self.alpha)

  # update w using Equation (2)
  d_w = w.grad.detach()
  w = w.detach() - self.alpha * d_w
```

Because SGD is already careful to properly detach its parameter (typically w, but in this case α), this implementation is functionally identical to the one above. Indeed, any optimizer that observes this protocol would suffice, so let us abstract out the optimizer as a parameter to HyperSGD:

```
def HyperSGD.__init__(self, alpha, opt):
  self.alpha = tensor(alpha)
  self.optimizer = opt

def HyperSGD.adjust(w):
  self.optimizer.adjust(self.alpha)
  d_w = w.grad.detach()
  w = w.detach() - self.alpha * d_w

opt = HyperSGD(0.01, opt=SGD(kappa))
```

Finally, after this refactoring, we can recursively feed `HyperSGD` *itself* as the optimizer, obtaining a level-2 hyperoptimizer `HyperSGD(0.01, HyperSGD(0.01, SGD(0.01)))`. Similarly, we can imagine taller towers, or towers that mix and match multiple different kinds of optimizers, such as Adam-optimized-by-SGD-optimized-by-Adam.

A logical concern is whether this process actually exacerbates the hyperparameter optimization problem by introducing even more hyperparameters. Baydin et al. (2018) predicted that as the towers of hyperoptimizers grow taller, the resulting algorithms would be less sensitive to the human-chosen hyperparameters. This is indeed the case; Section 3.4 presents an empirical evaluation.

3 Experiments

In this section we evaluate the hyperoptimizers made possible by our system, exploring in particular the benefits of being able to optimize hyperparameters beyond just step size, and to stack them to multiple levels. Each of these experiments was conducted on a single NVIDIA TITAN Xp GPU. Our implementation provides a simple wrapper for PyTorch `nn.Module` subclasses, which makes them compatible with our hyperoptimizers. The full source code for these experiments will be made available to reviewers via Supplementary Material.

3.1 Hyperoptimization for SGD

First, we establish some basic properties about hyperoptimizers: (1) whether an SGD hyperoptimizer performs better than a regular SGD optimizer, and (2) whether the final learned step size is better than the initial human-chosen step size. We test the latter property by running a fresh regular SGD optimizer with the final learned step size of the hyperoptimizer. Following authors of previous work in this area (Maclaurin et al., 2015; Baydin et al., 2018), we conducted initial experiments on the MNIST dataset (Lecun et al., 1998) using a neural network with one fully-connected hidden layer of

Optimizer	Test error
SGD(0.01)	8.99±0.22%
SGD(0.01) / SGD(0.01)	4.13±0.07%
SGD(0.1414±0.0042)	4.08±0.13%
SGD(0.01) / Adam(...)	4.40±0.09%
SGD(0.1198±0.0144)	4.48±0.28%

(a) Experiments with SGD (Section 3.1)

Optimizer	Test error
Adam(...)	3.50±0.02%
Adam(...) / SGD(10^{-5})	3.03±0.08%
Adam(0.0043[3], 0.8999[9], 0.9989[9])	3.01±0.15%
Adam$^{\alpha}$(...) / SGD(10^{-5})	3.18±0.13%
Adam$^{\alpha}$(0.004[3], ...)	3.23±0.19%
Adam(...) / Adam(...)	3.03±0.11%
Adam(0.005106[3], 0.82[4], 0.999048[7])	2.96±0.23%
Adam$^{\alpha}$(...) / Adam(...)	3.00±0.16%
Adam$^{\alpha}$(0.006[8], ...)	2.93±0.14%

(b) Experiments with Adam (Section 3.2)

Table 1: Hyperoptimizing MNIST. We denote hyperoptimizers by their constituent optimizers separated by slashes (the leftmost item adjusts the model's weights). Adam$^{\alpha}$ is an Adam optimizer where only α is optimized as by Baydin et al. (2018). "..." stands for the standard Adam hyperparameters ($\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$). Each hyperoptimizer experiment is repeated using the final hyperparameters (typeset in pink) learned by the algorithm.

size 128, tanh activations, and a batch size of 256. We trained all networks for 30 epochs, and as a baseline used SGD with $\alpha = 0.01$.

Table 1a summarizes the results of our experiments. We find that hyperoptimized SGD outperforms the baseline by a significant margin (bringing test error down from 8.99% to 4.13%). This holds even if we use an Adam optimizer to adjust the step size of the SGD optimizer (final test error 4.40%). Furthermore, when we re-ran the regular optimizers with the new learned hyperparameters, we found that they also performed better than the initial hyperparameter (4.08% and 4.40% test error for SGD / SGD and SGD / Adam, respectively).

3.2 Hyperoptimization for Adam

In Section 2.3, we described how to apply our system to the Adam optimizer, simultaneously optimizing not only the learning rate α, but also the momentum coefficients $\beta_{1,2}$. Here, we ask three questions: (1) whether hyperoptimized Adam optimizers perform better than regular Adam optimizers, (2) whether the learned hyperparameters outperform the baseline, and (3) whether there is a benefit to optimizing all the hyperparameters, as opposed to only optimizing the learning rate as Baydin et al. (2018) do. Because Adam has significantly faster convergence than SGD, we only run these experiments for 10 epochs to avoid overfitting.

Table 1b summarizes the results of our experiments. We find that indeed the hyperoptimized Adam optimizer outperforms the regular Adam optimizer on its "default" settings. As with SGD, the learned hyperparameters perform better than the initial hyperparameters when re-run with the regular optimizer. Inspecting the learned hyperparameters, we find that the algorithm raises the learning rate α and slightly lowers β_1, but does not significantly affect β_2. Nevertheless, learning β_1 does help in some circumstances: our all-parameters-hyperoptimized Adam (test error 3.03%) outperforms the only-step-size-hyperoptimized Adam$^\alpha$ (test error 3.18%), though not if the top-level optimizer was itself another Adam optimizer (3.03% vs 3.00%).

3.3 Hyperoptimization at scale

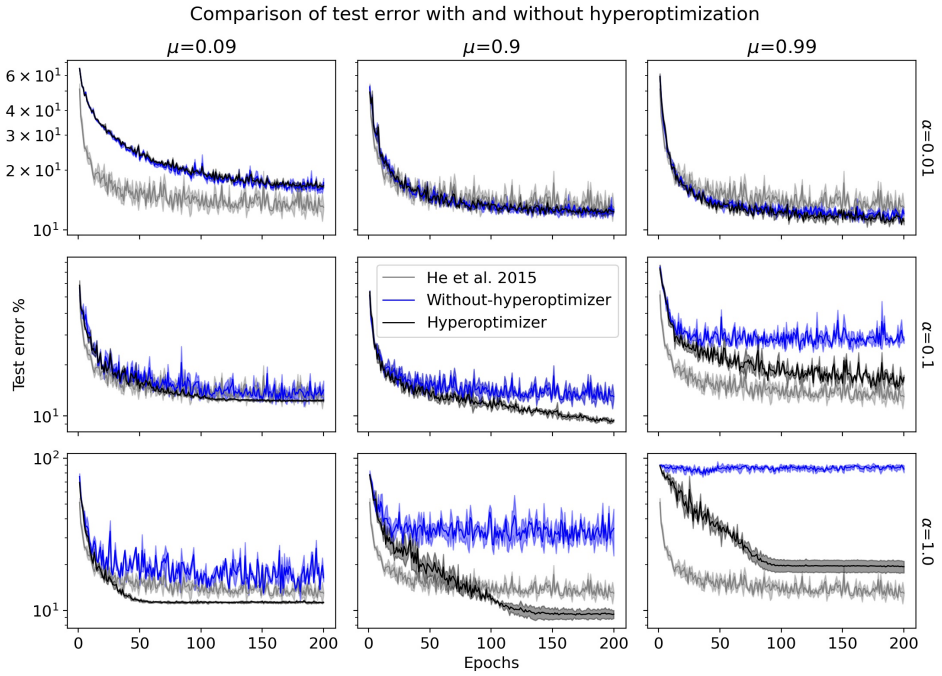
Next, we move beyond our small MNIST experiments and demonstrate the effect of using hyperoptimizers on two different real-world neural network architectures.

3.3.1 Convolutional neural networks for computer vision

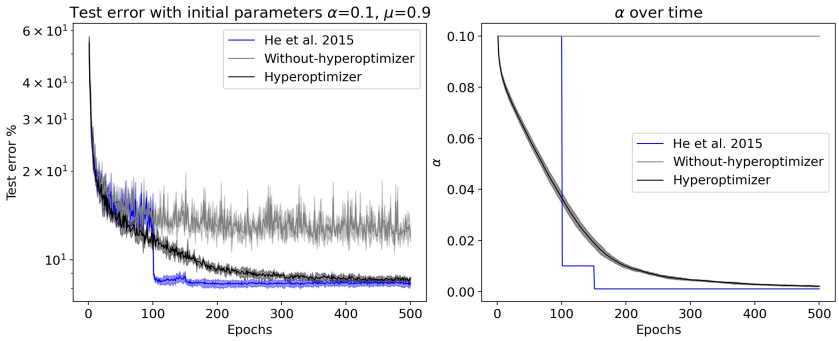
We train a ResNet-20 (He et al., 2016) with and without hyperoptimization on the CIFAR-10 dataset (Krizhevsky, 2012). As a baseline, we replicate the training procedure of He et al. (2016): we use the same network architecture, optimizer (SGD), step size (0.1), momentum (0.9), and weight decay (10^{-4}), though *without* their custom learning rate decay schedule (which we will address later). Experiments were run for 200 epochs, which takes around 3 hours.

First, we test how robust the hyperoptimizer is to "bad" initial choices of step size and momentum. We vary the initial step size and the momentum among "small," "good," and "large" values (that is, $\alpha \in \{0.01, 0.1, 1.0\}$ and $\mu \in \{0.09, 0.9, 0.99\}$), and add a hyperoptimizer ($\alpha_\alpha = \alpha^2 \cdot 10^{-3}$, $\alpha_\mu = 1/(1-\mu) \cdot 10^{-6}$). The results of this experiment are shown in Figure 2a. In every configuration, the hyperoptimizer matches or outperforms the regular optimizer in final test accuracy. Furthermore, in nearly all of the configurations, the hyperoptimizer matches or exceeds the "good" hyperparameters' final test accuracy. Only when *both* hyperparameters are bad in the same direction (too small or too large) is it unable to manage this, and even then for the too-large case it dramatically lowers the loss compared to no hyperoptimizer. **We conclude that hyperoptimizers are indeed beneficial for tuning both step size and momentum in this real-world setting.**

Next, we add in the learning rate decay schedule hand-engineered by He et al. (2016): the step size is divided by 10 at epochs 100 and 150. We compare this with a hyperoptimizer initialized with the same starting hyperparameters, training both variants for 500 epochs. Our results are shown in Figure 2b. Both networks with variable step size significantly outperform the one trained with the fixed-step-size optimizer. **The hyperoptimizer not only matches the final test loss of the hand-engineered learning rate decay schedule, but also learns a decay schedule strikingly similar to one hand-engineered by He et al.**



(a) For a wide range of "bad" initial hyperparameter configurations, the hyperoptimizer improves on (or at least matches) final test accuracy, and often matches or even outperforms the "good" initial hyperparameters.



(b) The hyperoptimizer matches performance of the hand-engineered learning rate decay schedule by He et al. (2016), learning a strikingly similar decay schedule (right plot).

Figure 2: Training ResNets on CIFAR-10 with hyperoptimizers (Section 3.3.1).

3.3.2 Recurrent neural networks for language modeling

We train a character-level RNN ("Char-RNN") on the Tolstoy dataset, as proposed by Karpathy et al. (2015) as a convenient testbed for language models, which is now often used to benchmark optimizers (Schneider et al., 2018; Schmidt et al., 2021). We took the architecture (2-layer LSTM with 128 hidden nodes) and "expert" optimizer (Adam optimizer with $\alpha = 2 \times 10^{-3}$, run for 50,000 gradient descent steps) directly from Johnson (2017) as recommended by Karpathy et al. We compare against our HyperAdam optimizer on a wide range of initial learning rates $\alpha \in \{10^{-4}, 2 \times 10^{-3}, 10^{-2}\}$, with $\alpha_\alpha = \alpha \cdot 10^{-2}$. We do not vary *initial* $\beta_{1,2}$ because in our experience these hyperparameters are typically left at their default values. However, we *do* allow the hyperoptimizer to vary $\beta_{1,2}$ over the course of training (with $\alpha_{\beta_1} = 10^{-4}$ and $\alpha_{\beta_2} = 2 \times 10^{-4}$). All runs took around 1 hour to train.

The results of this experiment are shown in Figure 3. We find that the hyperoptimizer performs comparably to the expert-chosen fixed step size (perplexity 5.41 ± 0.26 with hyperoptimizer vs

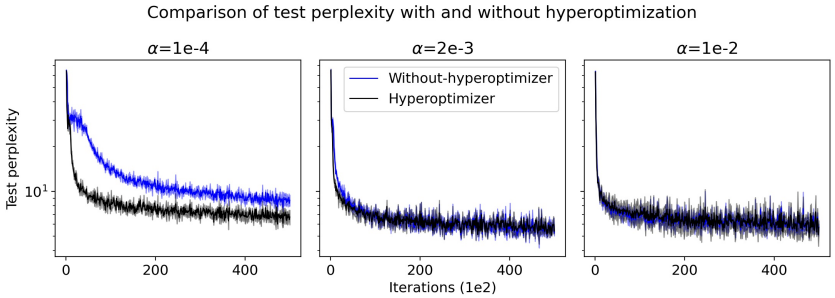
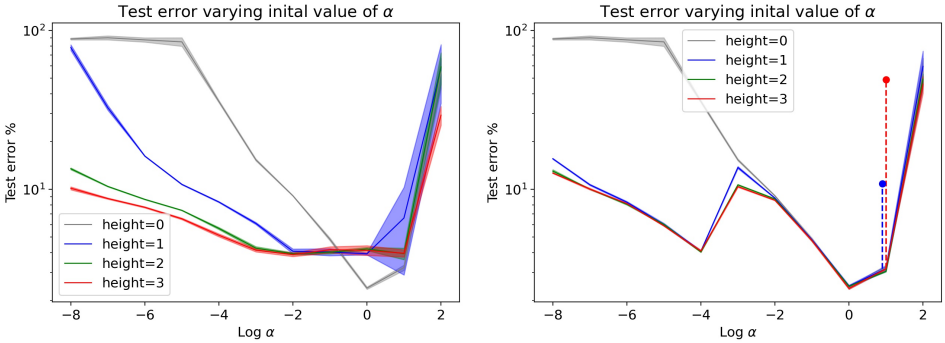
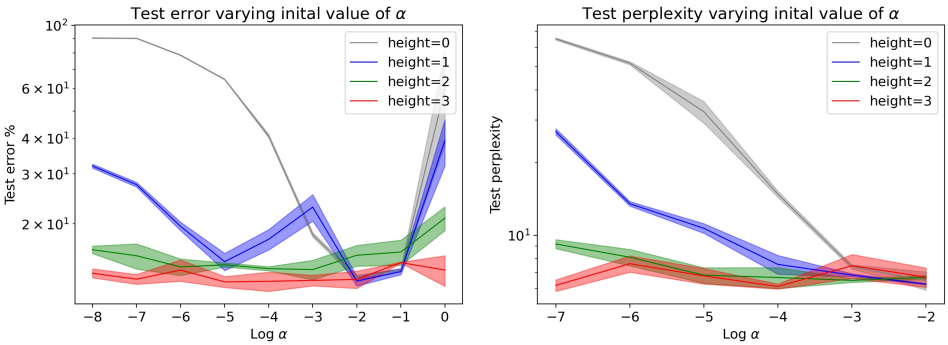


Figure 3: Training RNNs with hyperoptimizers (Section 3.3.2). As the initial learning rate is lowered, the regular Adam optimizer's convergence slows, but the hyperoptimizer is able to accelerate it. The hyperoptimizer also slightly improves convergence when the initial learning rate is too high.



(a) Results on an MLP (Sec 3.1), where all layers are initialized with the same step size.

(b) Results on an MLP (Sec 3.1), where all layers are initialized as in Sec 3.4.



(c) Results on a ResNet (Sec 3.3.1)

(d) Results on a Char-RNN (Sec 3.3.2)

Figure 4: Evaluating higher-order hyperoptimization (Section 3.4). As we stack more layers of optimizers, the resulting hyperoptimizer is less sensitive to the initial choice of hyperparameters.

5.27 ± 0.31 without), and improves upon "bad" initial step sizes in both directions (5.45 ± 0.76 vs 5.77 ± 0.34 when too high; 6.51 ± 0.88 vs 8.71 ± 0.91 when too low).

3.4 Higher-order hyperoptimization

In Section 2.4 we developed an interface for building arbitrarily tall towers of optimizers. Baydin et al. (2018) hypothesized that taller towers would yield hyperoptimizers that were more robust to the top-level human-chosen hyperparameters than elementary optimizers are. To validate this behavior of higher-order hyperoptimizers, we ran each of our benchmarks from above (MLP on MNIST, CNN on CIFAR-10, RNN on Tolstoy) with towers of hyperoptimizers of increasing heights, and initialized with initial step sizes α across many orders of magnitude. We strove to initialize each layer of the

stack with the same initial step size, but (aside from the MLP, see Figure 4a) this strategy proved unstable. Instead, we scaled step sizes at higher levels as follows: from $\alpha = 10^{-8}$ to 10^{-4} the higher layers' step sizes were initialized with $[\alpha \cdot 10^2, \alpha \cdot 10^0, \alpha \cdot 10^{-2}]$ respectively. For $\alpha \geq 10^{-3}$ even those initializations were unstable, so we lowered to $[\alpha \cdot 10^{-3}, \alpha \cdot 10^{-4}, 10^{-8}]$ respectively.

Figure 4 shows our results. It is indeed the case across these different benchmarks (each of which has a different dataset, architecture, and optimizer type) that the taller the hyperoptimizer stack, the less sensitive the results become to the human-chosen hyperparameters. **With a three-level optimizer stack, a single hyperoptimizer obtains reasonable results in all of our benchmarks across several orders of magnitude of base-level step size.**

For much higher initial learning rates ($\geq 10^0$ for MLP and CNN, $\geq 10^{-1}$ for RNN) the hyperoptimizer was unable to tune the learning rate. This is because the first few steps are so noisy and unstable that on some runs, the trainable parameters become irrecoverably large or `nan` before hyperoptimization can have any effect (this effect also causes the outliers in Figure 4b).

4 Related work

Hyperparameter optimization has a long history, and we refer readers interested in the full story to a recent survey by Feurer and Hutter (2019). Most existing work on gradient-based hyperparameter optimization (Bengio, 2000; Domke, 2012; Maclaurin et al., 2015; Pedregosa, 2016; Franceschi et al., 2017) has focused on computing hyperparameter gradients after several iterations of training, which is computationally expensive because of the need to backpropagate through much more computation. Baydin et al. (2018), building on a technique first published by Almeida et al. (1999), propose instead updating hyperparameters at *each* step. Luketina et al. (2016) apply a similar technique to regularization hyperparameters, though they note that their proposed method could work in principle for any continuous hyperparameter. As discussed above, we expand upon this line of work in three directions: (1) by fully automating this process, rather than requiring manual derivative computations; (2) by optimizing hyperparameters beyond just the learning rate; and (3) by realizing the vision of recursive higher-order hyperoptimizers and evaluating the resulting algorithms.

5 Limitations and future work

As discussed in Section 3.4, a major limitation of hyperoptimizers is that they cannot yet handle initial hyperparameters that are set far too high, because the system is unstable and diverges before the hyperoptimizer can have an effect. Designing hyperoptimizers robust in this regime requires further research. Our implementation also requires some care in avoiding certain bugs related to computation graph management. For example, loggers must detach what is logged to avoid memory leaks (because tensors are not garbage collected unless all children are detached), and certain PyTorch modules (e.g. the built-in LSTM) cannot be used because they silently modify the computation graph, which may lead to incorrect gradients with our system. Further research is needed to design differentiable programming languages where methods like ours can be expressed in a modular and composable manner that minimizes risk of such bugs.

6 Broader impact

Training a modern large-scale deep learning system consumes a tremendous amount of energy, and hyperparameter searches can multiply that energy impact by many orders of magnitude (Strubell et al., 2019). We hope that advances in on-line hyperparameter tuning can reduce this impact.

7 Conclusion

We presented a technique to enhance optimizers such as SGD and Adam by allowing them to tune their own hyperparameters by gradient descent. Unlike prior work, our proposed hyperoptimizers require no manual differentiation by the user, learn hyperparameters beyond just learning rates, and can be stacked recursively to many levels. We described an elegant recursive implementation of hyperoptimizers in a reverse-mode AD system and evaluated it on a variety of benchmarks.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Sec 5
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Sec 6
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

References

L. E. Almeida, T. Langlois, J. F. M. do Amaral, and A. Plakhov. Parameter adaptation in stochastic optimization. In *On-Line Learning in Neural Networks*, 1999.

A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood. Online learning rate adaptation with hypergradient descent. In *Sixth International Conference on Learning Representations (ICLR), Vancouver, Canada, April 30 – May 3, 2018*, 2018.

Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8): 1889–1900, 2000. doi: 10.1162/089976600300015187. URL `https://doi.org/10.1162/089976600300015187`.

J. Domke. Generic methods for optimization-based modeling. In N. D. Lawrence and M. Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22 of *Proceedings of Machine Learning Research*, pages 318–326, La Palma, Canary Islands, 21–23 Apr 2012. PMLR. URL `http://proceedings.mlr.press/v22/domke12.html`.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2021068.

M. Feurer and F. Hutter. *Hyperparameter Optimization*, pages 3–33. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_1. URL https://doi.org/10.1007/978-3-030-05318-5_1.

L. Franceschi, M. Donini, P. Frasconi, and M. Pontil. Forward and reverse gradient-based hyperparameter optimization. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1165–1173, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/franceschi17a.html.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

J. Johnson. torch-rnn. Github repository, 2017. URL https://github.com/jcjohnson/torch-rnn.

A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015. URL https://arxiv.org/pdf/1506.02078.pdf.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

A. Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

J. Luketina, M. Berglund, K. Greff, and T. Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2952–2960. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045701.

D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2113–2122. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045343.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

F. Pedregosa. Hyperparameter optimization with approximate gradient. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 737–746. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045469.

R. M. Schmidt, F. Schneider, and P. Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2021.

F. Schneider, L. Balles, and P. Hennig. Deepobs: A deep learning optimizer benchmark suite. In *International Conference on Learning Representations*, 2018.

E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019. URL https://arxiv.org/pdf/1906.02243.pdf.

M. D. Zeiler. ADADELTA: An adaptive learning rate method. *CoRR*, abs/1212.5701, 2012. URL http://dblp.uni-trier.de/db/journals/corr/corr1212.html#abs-1212-5701.