Gradient-based Hyperparameter Optimization Over Long Horizons

Anonymous Author(s)
Affiliation
Address
email

Abstract

Gradient-based hyperparameter optimization has earned a widespread popularity in the context of few-shot meta-learning, but remains broadly impractical for tasks with long horizons (many gradient steps), due to memory scaling and gradient degradation issues. A common workaround is to learn hyperparameters online, but this introduces greediness which comes with a significant performance drop. We propose forward-mode differentiation with sharing (FDS), a simple and efficient algorithm which tackles memory scaling issues with forward-mode differentiation, and gradient degradation issues by sharing hyperparameters that are contiguous in time. We provide theoretical guarantees about the noise reduction properties of our algorithm, and demonstrate its efficiency empirically by differentiating through $\sim 10^4$ gradient steps of unrolled optimization. We consider large hyperparameter search ranges on CIFAR-10 where we significantly outperform greedy gradient-based alternatives, while achieving $\times 20$ speedups compared to the state-of-the-art black-box methods. Code is provided and will be made public.

1 Introduction

Deep neural networks have shown tremendous success on a wide range of applications, including classification [1], generative models [2], natural language processing [3] and speech recognition [4]. This success is in part due to effective optimizers such as SGD with momentum or Adam [5], which require carefully tuned hyperparameters for each application. In recent years, a long list of heuristics to tune such hyperparameters has been compiled by the deep learning community, including things like: how to best decay the learning rate [6], how to scale hyperparameters with the budget available [7], and how to scale learning rate with batch size [8]. Unfortunately these heuristics are often dataset specific and architecture dependent [9]. They also don't apply well to new optimizers [10], or new tools, like batch normalization which allows for larger learning rates [11].

With so many ways to choose hyperparameters, the deep learning community is at risk of adopting models based on how much effort went into tuning them, rather than their methodological insight. The field of hyperparameter optimization (HPO) aims to find hyperparameters that provide a good generalization performance automatically. Unfortunately, existing tools are rather unpopular for deep learning, largely owing to their computational cost. Our method falls into gradient-based HPO, which calculates hypergradients, i.e. the gradient of some validation loss with respect to each hyperparameter. Gradient-based HPO should be more efficient than black-box methods as the dimensionality of the hyperparameter space increases, since it relies on gradients rather than trial and error. In practice however, learning hyperparameters with gradients has only been popular in few-shot learning tasks where the horizon is short, i.e. where the underlying task is solved with a few gradient steps. This is because long horizons cause hypergradient degradation, and incur a memory cost that makes reverse-mode differentiation prohibitive.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

Greedy gradient-based methods alleviate both of these issues by calculating local hypergradients based on intermediate validation losses. Unfortunately, this introduces some bias [15] and results in a significant performance drop, which we are able to quantify in this work. We make use of forward-mode differentiation, which has been shown to offer a memory cost constant with horizon size. However, previous forward-mode methods don't address gradient degradation explicitly and are thus limited to the greedy setting [16, 17].

We introduce FDS (Foward-mode Differentiation with hyperparameter Sharing), which to the best of our knowledge demonstrates for the first time that hyperparameters can be differentiated non-greedily over long horizons. Specifically, we make the following contributions: **(1)** we propose to share hyperparameters through time, both motivating it theoretically and with various experiments, **(2)** we combine the above in a forward-mode differentiation algorithm, and **(3)** we show that our method can significantly outperform various HPO algorithms, for instance when learning the hyperparameters of the SGD-momentum optimizer.

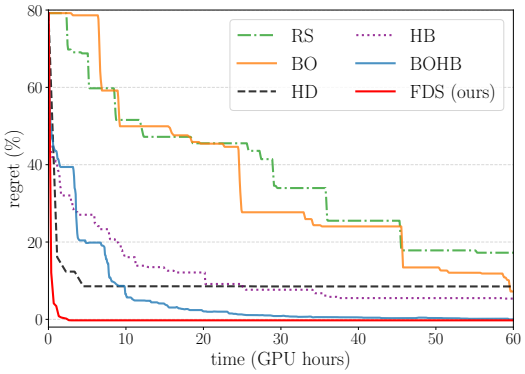

Figure 1: Performance of the most popular hyperparameter optimization methods when learning the learning rate, momentum and weight decay on 50 epochs of CIFAR-10, for a Wide ResNet of 16 layers. Non-greedy methods (RS, BO, HB [12], BOHB [13]) solve for global hyperparameters but rely on trial-and-error which makes them slow. Gradient-based methods (e.g. HD [14]) are much faster but solve for local hyperparameters which makes them suboptimal. Our method combines the strengths of these two paradigms and outperforms the next best method while converging 20 times faster. Each curve is the average of 8 seeds.

2 Related Work

There are many ways to perform hyperparameter optimization (HPO) [18], including Bayesian optimization (BO) [19], reinforcement learning [20], evolutionary algorithms [21] and gradient-based methods [22]. The state-of-the-art in HPO depends on the problem setting, but black-box methods like Hyperband (HB) [12], and its combination with BO into a method called BOHB [13] have been the most popular. Modern work in meta-learning deals with various forms of gradient-based HPO [23], but usually focuses on the few-shot regime [24] where horizons are conveniently short (~ 10 steps) while we focus on long horizons ($\sim 10^4$ steps).

Gradient-based HPO. Using the gradient of some validation loss with respect to the hyperparameters is typically the preferred choice when the underlying optimization is differentiable. This is a type of bilevel optimization [25] which stems from earlier work on backpropagation through time [26] and real-time recurrent learning [27]. Unfortunately, differentiating optimization is an expensive procedure in both time and memory, and most proposed methods are limited to small models and toy datasets [28–30]. Efforts to make the problem more tractable include optimization shortcuts [31], truncation [32] and implicit gradients [33, 34]. Truncation can be combined with our approach but produces biased gradients [35], while implicit differentiation is only applicable to hyperparameters that define the training loss (e.g. augmentation) but not to hyperparameters that define how the training loss is minimized (e.g. optimizer hyperparameters). Forward-mode differentiation [27] boasts a memory cost constant with horizon size, but gradient degradation has restricted its use to the greedy setting [16, 17].

Greedy Methods. One trick that prevents gradient degradation and significantly reduces compute and memory cost is to solve the bilevel optimization greedily. This has become the default trick in various long-horizon problems, including HPO over optimizers [14, 16, 17, 36], architecture search [37], dataset distillation [38] or curriculum learning [39]. Greediness refers to finding the best hyperparameters locally rather than globally. In practice, it involves splitting the inner optimization problem into smaller chunks (often just one batch), and solving for hyperparameters over these smaller horizons instead; often in an online fashion. We formalize greediness in section 3.2 In this paper we expand upon previous observations [15] and take the view that greediness fundamentally solves

for the wrong objective. Instead, the focus of our paper is to extend forward-mode differentiation methods to the non-greedy setting.

Gradient Degradation. Gradient degradation of some scalar w.r.t a parameter is a broad issue that arises when that parameter influences the scalar in a chaotic fashion, such as through long chains of nonlinear mappings. This manifests itself in HPO as vanishing or exploding hypergradients, due to low or high curvature components of the validation loss surface. This leads to hypergradients that have a large variance, making differentiation through long horizons impractical. This is usually observed in the context of recurrent neural networks [40, 41], but also in reinforcement learning [42] and HPO [29]. Solutions like LSTMs [43] and gradient clipping [44] have been proposed, but are respectively inapplicable and insufficient to long-horizon HPO. Variational optimization [35] and preconditioning warp-layers [45] can mitigate gradient degradation, but these methods are expensive in memory and therefore are limited to small architectures and/or a few hundred inner steps. In comparison, we differentiate over $\sim 10^4$ inner steps for Wide ResNets [46].

3 Background

3.1 Problem statement

Consider a neural network with weights $\boldsymbol{\theta}$, trained to minimize a loss $\mathcal{L}_{train}$ over a dataset $\mathcal{D}_{train}$, with a gradient-based optimizer $\Phi : \boldsymbol{\theta}_{t+1} = \Phi(\boldsymbol{\theta}_t(\boldsymbol{\lambda}_{[0:t-1]}), \boldsymbol{\lambda}_{[t]})$. Here $\boldsymbol{\lambda} \in \mathbb{R}^T$ is a hyperparameter vector and T is the total number of gradient steps taken on $\boldsymbol{\theta}$. We write vector indices in brackets to differentiate them from a variable evaluated at time t. Note that for any step t, $\boldsymbol{\theta}_t$ is a function of all the hyperparameters used in the previous steps $t = 0, 1, ..., t-1$.

We would like to find the hyperparameters $\boldsymbol{\lambda}^*$ such that the minimizer of $\mathcal{L}_{train}$, namely $\boldsymbol{\theta}^* \simeq \boldsymbol{\theta}_T(\boldsymbol{\lambda}^*)$, also minimizes some generalization loss $\mathcal{L}_{val}$ on a validation dataset $\mathcal{D}_{val}$. Bilevel optimization [47] is the framework of choice to express this problem:

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \mathcal{L}_{val}(\boldsymbol{\theta}_T(\boldsymbol{\lambda}), \mathcal{D}_{val}) \tag{1}$$

$$\text{subject to} \quad \boldsymbol{\theta}_T = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_{train}(\boldsymbol{\theta}, \mathcal{D}_{train}) \tag{2}$$

$$\text{solved with} \quad \boldsymbol{\theta}_{t+1} = \Phi(\boldsymbol{\theta}_t(\boldsymbol{\lambda}_{[0:t-1]}), \boldsymbol{\lambda}_{[t]}) \tag{3}$$

The inner loop (lower level) in Eq 2 expresses a constraint on the outer loop (upper level) in Eq 1. In gradient-based HPO, our task is to compute the hypergradient $d\mathcal{L}_{val}/d\boldsymbol{\lambda}$ and update $\boldsymbol{\lambda}$ accordingly.

3.2 Greediness

Let H be the horizon, which corresponds to the number of gradient steps taken in the inner loop (to optimize $\boldsymbol{\theta}$) before one gradient step is taken in the outer loop (to optimize $\boldsymbol{\lambda}$). When solving Eq 1 non-greedily we have $H = T$. However, most modern approaches are greedy [14, 16, 17, 36–38], in that they rephrase the above problem into a sequence of several independent problems of smaller horizons, where $\boldsymbol{\lambda}^*_{[t:t+H]}$ is learned in the outer loop subject to an inner loop optimization from $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+H}$ with $H \ll T$.

Greediness has many advantages: it mitigates gradient degradation, makes hypergradients cheaper to compute, and it requires less memory in reverse-mode differentiation. However, these methods look for $\boldsymbol{\lambda}^*$ that does well locally rather than globally, i.e. they constrain $\boldsymbol{\lambda}^*$ to a subspace of solutions such that $\boldsymbol{\theta}_H, \boldsymbol{\theta}_{2H}, ..., \boldsymbol{\theta}_T$ all yield good validation performances. In our experiments, we found that getting competitive hyperparameters with greediness often revolves around tricks like online learning with a very low outer learning rate combined with hand-tuned initial hyperparameter values, to manually prevent convergence to small values. But solving the greedy objective correctly leads to poor solutions, a special case of which was previously described as the "short-horizon bias" [15]. In FDS, we learn global hyperparameters in a non-greedy fashion, which means that the hypergradient $d\mathcal{L}_{val}/d\boldsymbol{\lambda}$ is only ever calculated at $\boldsymbol{\theta}_T$.

3.3 Forward-mode differentiation of modern optimizers

The vast majority of meta-learning applications use reverse-mode differentiation in the inner optimization problem (Eq 2) to optimize $\boldsymbol{\theta}$. However, the memory cost of using reverse-mode differentiation for the outer optimization (Eq 1) is $\mathcal{O}(FH)$, where F is the memory used by one forward pass through the network (weights plus activations). This is often referred to as backpropagation through time (BPTT). In the non-greedy setting where $H = T$, BPTT is extremely limiting: for large networks, only $T \sim 10$ gradient steps could be solved with modern GPUs, while problems like CIFAR-10 require $T \sim 10^4$. Instead, we make use of forward-mode differentiation, which scales in memory as $\mathcal{O}(DN)$, where D is the number weights and N is the number of learnable hyperparameters. The additional scaling with N is a limitation if we learn one hyperparameter per inner step ($N = T$). Sharing hyperparameters (section 4.1) mitigates gradient degradation, but also conveniently allows for smaller values of N.

For clarity of notation, we consider forward-mode hypergradients for the general case of using one hyperparameter per steps, i.e. $\boldsymbol{\lambda} \in \mathbb{R}^T$. First, we use the chain rule to write $d\mathcal{L}_{val}/d\boldsymbol{\lambda} = (\partial\mathcal{L}_{val}/\partial\boldsymbol{\theta}_T)(d\boldsymbol{\theta}_T/d\boldsymbol{\lambda})$ where the direct gradient has been dropped since $\partial\mathcal{L}_{val}/\partial\boldsymbol{\lambda} = 0$ for optimizer hyperparameters. The first term on the RHS is trivial and can be obtained with reverse-mode differentiation as usual. The second term is more problematic because $\boldsymbol{\theta}_T = \boldsymbol{\theta}_T(\boldsymbol{\theta}_{T-1}(\boldsymbol{\theta}_{T-2}(...), \boldsymbol{\lambda}_{[T-2]}), \boldsymbol{\lambda}_{[T-1]})$. We use the chain rule again to calculate this term recursively:

$$\frac{d\boldsymbol{\theta}_t}{d\boldsymbol{\lambda}} = \left.\frac{\partial\boldsymbol{\theta}_t}{\partial\boldsymbol{\theta}_{t-1}}\right|_{\boldsymbol{\lambda}} \frac{d\boldsymbol{\theta}_{t-1}}{d\boldsymbol{\lambda}} + \left.\frac{\partial\boldsymbol{\theta}_t}{\partial\boldsymbol{\lambda}}\right|_{\boldsymbol{\theta}_{t-1}} \quad \text{which we write as} \quad \boldsymbol{Z}_t = \boldsymbol{A}_t\boldsymbol{Z}_{t-1} + \boldsymbol{B}_t \tag{4}$$

where $\boldsymbol{\theta}_t \in \mathbb{R}^D$, $\boldsymbol{\lambda} \in \mathbb{R}^T$, $\boldsymbol{Z}_t \in \mathbb{R}^{D\times T}$, $\boldsymbol{A}_t \in \mathbb{R}^{D\times D}$ and $\boldsymbol{B}_t \in \mathbb{R}^{D\times T}$. Note that the columns of $\boldsymbol{Z}$ at step t are zeros for indices $t+1, t+2, ..., T$, since the hyperparameters at those steps haven't been used yet.

The expressions for $\boldsymbol{A}_t$ and $\boldsymbol{B}_t$ depend on the specific hyperparameters we are differentiating. In this work, we consider the most popular optimizer, namely SGD with momentum and weight decay. To the best of our knowledge, previous work focuses on simpler versions of this optimizer, usually by removing momentum and weight decay, and only learns the learning rate, greedily. We use Pytorch's update rule for SGD [48], namely $\boldsymbol{\theta}_t = \Phi(\boldsymbol{\theta}_{t-1}) = \boldsymbol{\theta}_{t-1} - \alpha\boldsymbol{v}_t$ with learning rate α_t, momentum β_t, weight decay μ_t and velocity $\boldsymbol{v}_t = \beta_t\boldsymbol{v}_{t-1} + (\partial\mathcal{L}_{train}/\partial\boldsymbol{\theta}_{t-1}) + \mu_t\boldsymbol{\theta}_{t-1}$. Consider the case when we learn the learning rate schedule, namely $\boldsymbol{\lambda} = \boldsymbol{\alpha}$. If we use the update rule without momentum [17], $\boldsymbol{B}_t$ is conveniently sparse: it is a $D \times T$ matrix that only has one non-zero column at index t corresponding to $\partial\boldsymbol{\theta}_t/\partial\alpha_t$. However, we include terms like momentum and therefore the velocity depends on the hyperparameters of previous steps. In that case, a further recursive term $\boldsymbol{C}_t = (\partial\boldsymbol{v}_t/\partial\boldsymbol{\lambda})$ must be considered to get exact hypergradients. Putting it together (see Appendix A) we obtain:

$$\begin{cases} \boldsymbol{A}_t^{\boldsymbol{\alpha}} = \mathbf{1} - \alpha_t\left(\dfrac{\partial^2\mathcal{L}_{train}}{\partial\boldsymbol{\theta}_{t-1}^2} + \mu_t\mathbf{1}\right) \\ \boldsymbol{B}_t^{\boldsymbol{\alpha}} = -\beta_t\alpha_t\boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} - \delta_t^{\otimes}\left(\beta_t\boldsymbol{v}_{t-1} + \dfrac{\partial\mathcal{L}_{train}}{\partial\boldsymbol{\theta}_{t-1}} + \mu_t\boldsymbol{\theta}_{t-1}\right) \\ \boldsymbol{C}_t^{\boldsymbol{\alpha}} = \beta_t\boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} + \left(\mu_t\mathbf{1} + \dfrac{\partial^2\mathcal{L}_{train}}{\partial\boldsymbol{\theta}_{t-1}^2}\right)\boldsymbol{Z}_{t-1}^{\boldsymbol{\alpha}} \end{cases} \tag{5}$$

where $\mathbf{1}$ is a $D \times D$ identity matrix, and $\delta_t^{\otimes}(\boldsymbol{q})$ turns a vector $\boldsymbol{q}$ of size D into a matrix of size $D \times T$, whose t-th column is set to $\boldsymbol{q}$ and other columns to $\mathbf{0}$s. While the matrices in Eq 5 are updated online, the hyperparameters aren't. This is because in the non-greedy setting we don't have access to the hypergradients until we have computed $\boldsymbol{Z}_T^{\boldsymbol{\alpha}}$. A similar technique can be applied to momentum and weight decay to get $\boldsymbol{Z}_T^{\boldsymbol{\beta}}$ and $\boldsymbol{Z}_T^{\boldsymbol{\mu}}$ (see Appendix A). All hypergradient derivations in this paper were checked with finite differences. Note that we focus on learning the hyperparameters of SGD with momentum because it is the most common optimizer in the deep learning literature. However, just like reverse-mode differentiation, forward-mode differentiation can be applied to any differentiable hyperparameter, albeit with appropriate $\boldsymbol{A_t}$ and $\boldsymbol{B_t}$ matrices.

4 Non-greedy Differentiation Over Long Horizons

4.1 Hyperparameter sharing: trading off noise reduction with bias increase

The main challenge of doing non-greedy meta-learning over long horizons is gradient degradation. In HPO this arises because a small change in $\mathcal{L}_{train}(\boldsymbol{\theta}_t)$ can cascade forward into a completely different $\boldsymbol{\theta_T}$, resulting in large fluctuations of the hypergradients. This noise makes the generalization loss hard to minimize (Eq 1). We find that the two main causes for this noise are the ordering of the training minibatches, and the weight initialization $\boldsymbol{\theta}_0$. Ideally, the hyperparameters we learn should be agnostic to both of these factors, and so we would like to average out their effect on hypergradients. The most obvious way to do that is to compute hypergradients for each time step t across several random seeds, where each seed corresponds to a different dataset ordering and weight initialization. We could then obtain an average hypergradient value at each inner step t, which we consider as optimal in our analysis.

The issue with this approach is its computational and memory cost. Indeed, each calculation of the hypergradients requires solving and differentiating through the entire inner problem, and has thus been considered intractable in the literature [15]. Instead, we propose to use the long horizon to our advantage and *average out hypergradients across the inner loop (Eq 2) rather than the outer loop (Eq 1)*. More specifically, we sum hypergradients from W neighbouring time steps in the inner loop (Eq 3), which is exactly equivalent to sharing one hyperparameter over all these steps. The average is then obtained trivially by dividing by W. For instance, when learning the learning rate schedule $\boldsymbol{\alpha}$ for $H = 10^4$ inner steps, we can learn $\boldsymbol{\alpha} \in \mathbb{R}^{10}$ where each learning rate is used for a window of $W = 10^3$ contiguous gradient steps.

While averaging hypergradients reduces noise when contiguous hypergradients are sampled from similar distributions, using a single average hypergradient for W contiguous steps introduces a bias, since it prevents each step to learn its own hyperparameter. We flesh out the details of this trade-off in Theorem 1.0.

Theorem 1.0 Consider that each time step $t \in \{1, 2, ..., T\}$ has a corresponding hyperparameter λ_t and non-greedy hypergradient $g_t = \partial L_{val}(\boldsymbol{\theta}_T)/\partial \lambda_t$. Each g_t is viewed as a random variable due to the sampling process of the weight initialization $\boldsymbol{\theta}_0$ and the inner loop minibatch selection. Let $\boldsymbol{g} = [g_1, g_2, \ldots, g_T]$ be sufficiently well approximated by a Gaussian distribution $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with mean $\boldsymbol{\mu} = [\mu_1, \mu_2, ..., \mu_T]$ and covariance matrix $\boldsymbol{\Sigma}$, where $\boldsymbol{\mu}$ corresponds to the optimal hypergradients. Assume that the values of $\boldsymbol{\mu}$ can be written as the ϵ-Lipschitz function $\mu_{t+1} = \mu_t + \epsilon_t$, where $\epsilon_t \in [-\epsilon, \epsilon]$. Finally, let $c \in [-1, 1]$ denote the maximum correlation between the values of $\boldsymbol{g}$. Then, we show that the mean squared error of the hypergradients with respect to $\boldsymbol{\mu}$ when sharing W contiguous hyperparameters has an upper bound:

$$\mathrm{MSE}_W \leq \frac{(1 + c(W-1))}{W}\mathrm{MSE}_1 + \epsilon^2 \frac{(W^2-1)}{12} \tag{6}$$

$$\text{where} \quad \mathrm{MSE}_1 = \frac{1}{T}\sum_t \boldsymbol{\Sigma}_{tt} \tag{7}$$

and so for sufficiently small ϵ and c we have with certainty, $MSE_W < MSE_1$ for some $W > 1$, where MSE_1 is the hypergradient error without any sharing (See Appendix B for proof).

Discussion Theorem 1.0 demonstrates how sharing W contiguous hyperparameters has two effects: 1) it reduces the hypergradient noise by a factor $\mathcal{O}(W/(1 + cW))$ due to the averaging of noisy hypergradients, and 2) it increase the error up to an amount $\mathcal{O}(\epsilon^2 W^2)$ due to an induced bias. Intuitively, averaging contiguous hypergradients maximally reduces noise when they aren't correlated ($c = 0$), and minimally increases bias when they are drawn from distributions of similar means ($\epsilon = 0$). In the simpler case where hypergradients are iid and have the same variance at each step, namely $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{1})$, the expressions above become $\mathrm{MSE}_1 = \sigma^2$ and $\mathrm{MSE}_W \leq \mathrm{MSE}_1/W + \epsilon^2(W^2-1)/12$. Note that in all cases, the upper bound on MSE_W has a single minimum W^* corresponding to the optimal trade-off between noise reduction and bias increase.

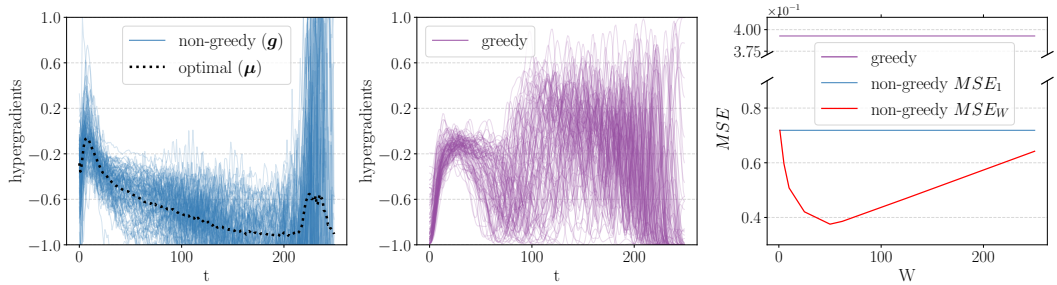

Figure 2: Hypergradients of $\boldsymbol{\alpha}$ when calculated over 100 different seeds in the non-greedy (left) and greedy (middle) setting. The mean squared error is also shown (right), and is calculated with respect to the optimal hypergradient ($\boldsymbol{\mu}$), i.e. the average of non-greedy hypergradients (dotted line in left figure). We can see that sharing hyperparameters over W steps lowers the MSE even for the small value of $T = 250$ used here. The best trade-off between noise reduction and bias increase is $W = 50$.

4.2 The FDS algorithm

As it is presented in section 3.3, forward mode differentiation would still have a memory cost that scales as $\mathcal{O}(DT)$ since we are learning one hyperparameter per step. However, addressing gradient degradation by sharing hyperparameters also conveniently reduces that memory cost by a factor W down to $\mathcal{O}(DN)$, where $N = T/W$ is the number of unique hyperparameter values we learn. This is because we can safely average hypergradients without calculating them individually, by reusing the same column of $\boldsymbol{Z}$ for W contiguous steps. This is shown in Algorithm 1, when learning a schedule of $N^{\boldsymbol{\alpha}}$ learning rates with FDS. Note that none of the matrices we store in memory scale as T. We also never need to store $\mathcal{H}$ or $A^{\boldsymbol{\alpha}}$ in memory since we calculate the Hessian matrix product $\mathcal{H}Z^{\boldsymbol{\alpha}}$ directly. Most importantly, note that hyperparameters aren't updated greedily or online, but are updated once per outer step, which corresponds to differentiating through the entire unrolled inner loop optimization and getting the exact hypergradients $\partial\mathcal{L}_{val}(\boldsymbol{\theta}_T)/\partial\boldsymbol{\alpha}$.

The main cost of Algorithm 1 comes from calculating $\mathcal{H}Z^{\boldsymbol{\alpha}}$. We find that a first-order approximation which sets the Hessian to zero is too crude for large horizons. We can easily adapt the work of [32] to FDS by truncating alternating inner steps, but we found also found this to degrade hypergradients too much. One could also use functional forms for more complex schedules to be learned in terms of fewer hyperparameters, but this typically results in including stronger assumptions about the shape of each hyperparameter schedule, which can easily cloud the true performance of HPO algorithms. In practice, we use a similar form to Algorithm 1 to learn the learning rate, momentum and weight decay jointly.

Algorithm 1 Simplified FDS algorithm when learning N^α learning rates, each used for W contiguous time steps

Initialize: $N^\alpha, W = T/N^\alpha, \boldsymbol{\alpha} = \mathbf{0}^{N^\alpha}$

#outer loop
for o in $1, 2, \ldots$ **do**
  Initialize: $\mathcal{D}_{train}, \mathcal{D}_{val}, \boldsymbol{\theta}_0 \in \mathbb{R}^D$,
  $Z^{\boldsymbol{\alpha}} = \mathbf{0}^{D\times N^\alpha}, C^{\boldsymbol{\alpha}} = \mathbf{0}^{D\times N^\alpha}$

  #inner loop
  for t in $1, 2, \ldots, T$ **do**
    $\boldsymbol{x}_{train}, \boldsymbol{y}_{train} \sim \mathcal{D}_{train}$
    $\boldsymbol{g}_{train} = \partial\mathcal{L}_{train}(\boldsymbol{x}_{train}, \boldsymbol{y}_{train})/\partial\boldsymbol{\theta}$

    #hyperparameter sharing
    $i = \lfloor t/W \rfloor$
    $\mathcal{H}Z^{\boldsymbol{\alpha}}_{[0:i]} = \partial(\boldsymbol{g}_{train} Z^{\boldsymbol{\alpha}}_{[0:i]})/\partial\boldsymbol{\theta}$
    $Z^{\boldsymbol{\alpha}}_{[0:i]} = A^{\boldsymbol{\alpha}} Z^{\boldsymbol{\alpha}}_{[0:i]} + B^{\boldsymbol{\alpha}}_{[0:i]}$

    update $C^{\boldsymbol{\alpha}}$ (Eq 5)
    $\boldsymbol{\theta}_{t+1} = \Phi(\boldsymbol{\theta}_t)$

  end for

  $\boldsymbol{g}_{val} = \partial\mathcal{L}_{val}(\mathcal{D}_{val})/\partial\boldsymbol{\theta}$
  $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - 0.1 \times \boldsymbol{g}_{val} Z^{\boldsymbol{\alpha}}/W$

end for

5 Experiments

Our experiments show how FDS mitigates gradient degradation and outperforms competing HPO methods for tasks with a long horizon. In sections 5.1 and 5.2 we consider small datasets (MNIST and SVHN) where reverse-mode differentiation is tractable, so that the effect of hyperparameter sharing can be measured. We then showcase FDS on larger datasets where only forward-mode differentiation is tractable in sec-

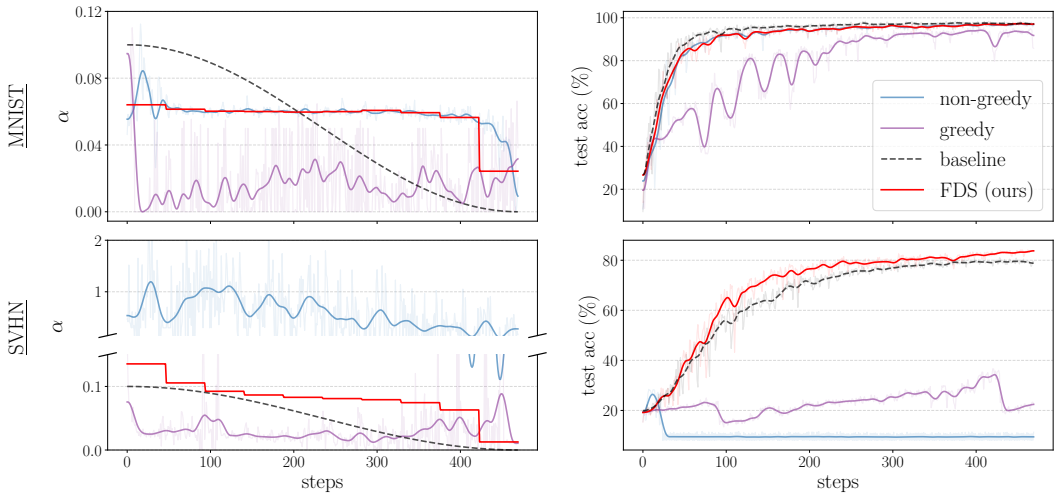

Figure 3: The learning rate schedule $\boldsymbol{\alpha}$ learned for the MNIST and SVHN datasets using a LeNet architecture over one epoch. We observe that on real-world datasets like SVHN, both greedy and non-greedy hyperparameter optimizations fail to learn decent learning rate schedules when using one hypergradient per inner step. However, sharing learning rates over contiguous steps stabilizes non-greedy hypergradients and allows us to find learning rates that can even outperform reasonable off-the-shelf schedules in this setting.

tions 5.3 and 5.4. All experiments are carried out on a single GTX 2080 GPU. More implementation details can be found in Appendix C.

5.1 The effect of hyperparameter sharing on hypergradient noise

We consider a LeNet network trained on an inner loop of $T = 250$ gradient steps, and calculate the hypergradients of the learning rate $\boldsymbol{\alpha}$ across 100 seeds. Each seed is evaluated at the same value of $\boldsymbol{\alpha}$, but corresponds to a different training dataset ordering and weight initialization. We can calculate the hypergradients greedily ($H = 1$) and non-greedily ($H = T$). The optimal hypergradients are considered to be the average of the non-greedy seeds as per section 4.1, which allows an MSE to be calculated for each method. These results are shown in Figure 2 for a learning rate schedule initialized to small values. We observe that greediness is a poor approximation to the optimal hypergradients, and that averaging contiguous hypergradients in the non-greedy case can significantly reduce the MSE even when averaging over only $W = 50$ steps.

The simplicity of this problem allows us to calculate $\mathbf{\Sigma}$, c and ϵ as defined in Theorem 1.0, to verify that our upper bound holds and that its shape as a function of W is realistic. In the setting shown in Figure 2 we find $\epsilon = \max |\mu_{t+1} - \mu_t| \sim 0.08$ and $(1/T)\sum_t \mathbf{\Sigma}_{tt} \sim 0.25$. The value of the maximum correlation between any two steps, c, can be quite high which makes the upper bound loose, but in practice . In practice however, the shape of the upper bound in Theorem 1.0 as a function of W (see Appendix B figure) closely matches the measured mean squared error as plotted in Figure 2. Note that the values of ϵ, $\mathbf{\Sigma}$, and c can vary depending on the value of $\boldsymbol{\alpha}$. As illustrated in Theorem 1.0, we find that settings that have a smaller values of ϵ_t benefit from a larger W and reduce the MSE more (see Appendix D for more examples).

5.2 The effect of hyperparameter sharing on HPO

The section above considered hypergradient noise around a fixed hyperparameter setting. In this section, we consider how that noise and its mitigation translate to meta-learning hyperparameters over several outer steps.

In Figure 3, we initialize $\boldsymbol{\alpha} = \mathbf{0}$ and train a LeNet network over $T \sim 500$ gradient steps on MNIST and SVHN. We compare the maximally greedy setting ($H = 1$), the non-greedy setting ($H = T$), and FDS (also $H = T$ but with sharing of contiguous hyperparameters). In all cases we take 50

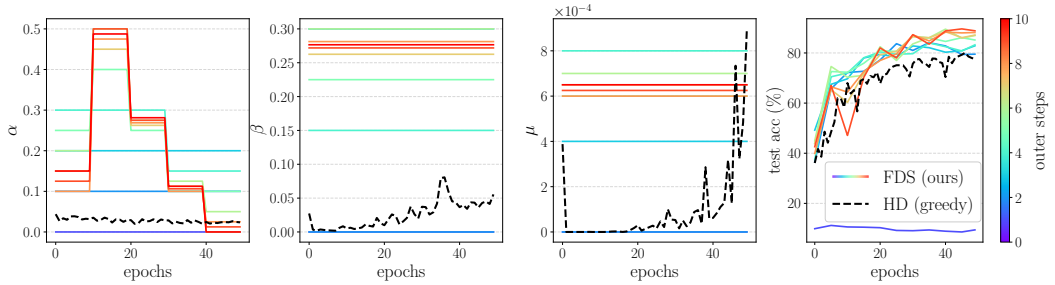

Figure 4: FDS applied to the SGD optimizer to learn (from left to right) the learning rate schedule $\boldsymbol{\alpha}$, the momentum $\boldsymbol{\beta}$, and weight decay $\boldsymbol{\mu}$ for a WRN-16-1 on CIFAR-10. For each outer step (color) we solve CIFAR-10 from scratch for 50 epochs, and update all hyperparameters such that the final weights minimize some validation loss. We use hyperparameter sharing over $W = 10$, 50 and 50 epochs for $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\mu}$ respectively. All hyperparameters are initialized to zero and converge within just 10 outer steps to values that significantly outperform Hypergradient Descent (HD) [14], the online greedy alternative. We also outperform state-of-the-art black-box alternatives (see Section 5.4).

outer steps per hyperparameter. For the greedy setting, we make greediness more transparent by not using tricks like online learning as in Hypergradient Descent [14] where α_t is set to α_{t-1} before its hypergradients are calculated. As previously observed by [15], greedy optimization leads to poor solutions with learning rates that are always too small. While the non-greedy setting without sharing works well for simple datasets like MNIST, it fails for real-world datasets like SVHN, converging to much higher learning rates than reasonable. This is due to gradient degradation, whose negative effect can compound during outer optimization, as a single large learning rate can increase σ for neighbouring steps. As we share hyperparameters in FDS, we reduce and stabilize the outer optimization. This allows us to learn a much more sensible learning rate schedule, which can even outperform a reasonable cosine annealing off-the-shelf schedule on SVHN.

5.3 FDS on CIFAR-10

We demonstrate that our algorithm can be used to learn the learning rate, momentum and weight decay over 50 epochs of CIFAR-10 ($H \sim 10^4$) for a Wide ResNet of 16 layers. We choose not to use larger architectures or more epochs to save compute time and because hyperparameters matter most for fewer epochs. Note also that we are not interested in finding the model with the best performance, but rather in finding the best hyperparameters given a model. For the learning rate, we choose W such that the ratio of T/W is similar to the optimal one found in 5.1, and we set $W = T$ for the momentum and weight decay since only a single value is commonly used for these hyperparameters. The schedules learned are shown in Figure 4, which demonstrates that FDS converges in just 10 outer steps to hyperparameters that are very different to online greedy differentiation [14], and correspond to significantly better test accuracy performances. Note that having the maximum learning rate be large and occur half way during training is reminiscent of the one-cycle schedule [49].

5.4 FDS compared to other HPO methods

A common theme in meta-learning research has been the lack of appropriate baselines, with researchers often finding that random search (RS) can outperform complex search algorithms, for instance in NAS [50] or automatic augmentation [51]. In this section we compare FDS to several competing HPO methods on CIFAR-10, in both the performance of the hyperparameters found and the time it takes to find them. We consider Random Search (RS), Bayesian Optimization (BO), Hypergradient Descent (HD) [14], HyperBand (HB) [12] and the combination of BO and HB, namely BOHB [13]. The latter is typically regarded as state-of-the-art in HPO. We use the official HpBandster [52] implementation of these algorithms, except for HD which we re-implemented ourselves.

The performance of HPO methods is often clouded by very small search ranges. For instance, in [13] the learning rate is searched over the range $[10^{-6}, 10^{-1}]$. In the case of DARTS [37],

expanding the search space to include many poor architectures has helped diagnose issues with its search algorithm [53]. For these reasons, we assume only weak priors and consider large search ranges: $\alpha \in [-1, 1]$, $\beta \in [-1.5, 1.5]$, and $\mu \in [-4 \times 10^{-3}, 4 \times 10^{-3}]$, which includes many poor hyperparameter values. In FDS we can specify search ranges by using a fixed update size $\boldsymbol{\gamma}$ to the hyperparameters, which decays by 2 every time the hypergradient flips sign, which is common in sign based optimizers [54–57]. We found that black-box HPO methods did not scale well to more than ~ 10 hyperparameters for such large search ranges, and so considered learning 7 learning rates, 1 momentum and 1 weight decay over 50 epochs. Note that increasing these numbers doesn't affect the accuracy of FDS but significantly reduces that of RS, BO, HB and BOHB. The performance over time of the hyperparameters found by each method is shown in Figure 1. As is common in HPO, we plot regret (in test accuracy) with respect to the best hyperparameters known in this setting, which we obtained from an expensive grid search around the most common hyperparameters used in the literature (more details in Appendix F). To squeeze out the optimal performance out of FDS in this table, we use the same trick as used in HB and BOHB, namely using smaller budgets for some runs, in particular early ones. We can see that our method matches the performance of the hand-tuned baseline in just three hours, slightly outperforming the hyperparameters that BOHB converges to, and doing so nearly 20 times faster. We retain the convergence speed of online greedy differentiation [14] while outperforming its performance by $\sim 10\%$ test accuracy.

6 Discussion

While FDS is significantly better than modern HPO methods in the case of learning hyperparameters over long horizons, it is worth pointing out some of its limitations. First, black-box methods are slower but can readily tackle non-differentiable hyperparameters, while FDS would need to use relaxation techniques to differentiate through, say, discrete variables. Note however that forward-mode differentiation can solve for any differentiable hyperparameter, contrary to some other approaches like implicit differentiation [34]. Then, the computational and memory cost scale linearly with the number of hyperparameters. For instance, for a WideResNet of 16 layers we are limited in memory to $\sim 10^3$ hyperparameters on a single 12 GB GPU. This is still one or two order of magnitude more than most black-box methods, but it falls short of reverse-mode-based greedy methods. Finally, forward-mode differentiation doesn't come in an automatic-differentiation package yet, like reverse differentiation, and so some pen and paper work is still required to derive hypergradients. The focus of our future work will be on the many meta-learning applications that rely on greedy optimization, such as differentiable architecture search, to improve their performance by making them none greedy.

7 Conclusion

This work makes an important step towards gradient-based HPO for long horizons by introducing FDS, which enables non-greediness through the mitigation of gradient degradation. More specifically, we theorize and demonstrate that sharing hyperparameters over contiguous time steps is a simple and efficient way to reduce the error in hypergradients, which naturally lands itself to forward-mode differentiation. We quantify the performance improvement this brings compared to the ubiquitous greedy approach. Finally, we show that FDS outperform greedy gradient-based alternatives in the quality of hyperparameters found, while being significantly faster than all state-of-the-art black-box methods. We hope that our work encourages the community to reconsider gradient-based HPO in terms of non-greediness, and pave the way towards a universal hyperparameter solver.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[2] A. Brock, J. Donahue, and K. Simonyan, "Large scale GAN training for high fidelity natural image synthesis," in *International Conference on Learning Representations*, 2019.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations* (Y. Bengio and Y. LeCun, eds.), 2015.

[6] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations*, 2017.

[7] M. Li, E. Yumer, and D. Ramanan, "Budgeted training: Rethinking deep neural network training under resource constraints," in *International Conference on Learning Representations*, 2020.

[8] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[9] X. Dong, M. Tan, A. W. Yu, D. Peng, B. Gabrys, and Q. V. Le, "Autohas: Differentiable hyper-parameter and architecture search," 2020.

[10] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2019.

[11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on Machine Learning* (F. Bach and D. Blei, eds.), Proceedings of Machine Learning Research, (Lille, France), PMLR, 07–09 Jul 2015.

[12] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, pp. 185:1–185:52, 2017.

[13] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning*, pp. 1436–1445, 2018.

[14] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online learning rate adaptation with hypergradient descent," in *International Conference on Learning Representations*, 2018.

[15] Y. Wu, M. Ren, R. Liao, and R. Grosse., "Understanding short-horizon bias in stochastic meta-optimization," in *International Conference on Learning Representations*, 2018.

[16] L. Franceschi, M. Donini, P. Frasconi, and M. Pontil, "Forward and reverse gradient-based hyperparameter optimization," in *International conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), Proceedings of Machine Learning Research, (International Convention Centre, Sydney, Australia), PMLR, 06–11 Aug 2017.

[17] M. Donini, L. Franceschi, M. Pontil, O. Majumder, and P. Frasconi, "Scheduling the Learning Rate via Hypergradients: New Insights and a New Algorithm," *arXiv e-prints*, Oct. 2019.

[18] M. Feurer and F. Hutter, *Chapter 1: Hyperparameter Optimization*. Cham: Springer International Publishing, 2019.

[19] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable bayesian optimization using deep neural networks," in *International conference on Machine Learning*, 2015.

[20] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations*, 2017.

[21] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, *et al.*, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[22] Y. Bengio, "Gradient-based optimization of hyperparameters," *Neural Comput.*, Aug. 2000.

[23] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, "Meta-learning in neural networks: A survey," *arXiv preprint arXiv:2004.05439*, 2020.

[24] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135, PMLR, 2017.

[25] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil, "Bilevel programming for hyperparameter optimization and meta-learning," in *International conference on Machine Learning*, 2018.

[26] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, 1990.

[27] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, 1989.

[28] J. Domke, "Generic methods for optimization-based modeling," in *International Conference on Artificial Intelligence and Statistics* (N. D. Lawrence and M. Girolami, eds.), Proceedings of Machine Learning Research, (La Palma, Canary Islands), PMLR, 21–23 Apr 2012.

[29] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based Hyperparameter Optimization through Reversible Learning," *arXiv e-prints*, Feb. 2015.

[30] F. Pedregosa, "Hyperparameter optimization with approximate gradient," in *International conference on Machine Learning*, 2016.

[31] J. Fu, H. Luo, J. Feng, K. H. Low, and T.-S. Chua, "Drmad: Distilling reverse-mode automatic differentiation for optimizing hyperparameters of deep neural networks," in *IJCAI*, 2016.

[32] A. Shaban, C.-A. Cheng, N. Hatch, and B. Boots, "Truncated back-propagation for bilevel optimization," in *AISTATS*, 2019.

[33] A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine, "Meta-learning with implicit gradients," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019.

[34] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing Millions of Hyperparameters by Implicit Differentiation," *arXiv e-prints*, Nov. 2019.

[35] L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein, "Understanding and correcting pathologies in the training of learned optimizers," vol. 97, pp. 4556–4565, 09–15 Jun 2019.

[36] J. Luketina, M. Berglund, K. Greff, and T. Raiko, "Scalable gradient-based tuning of continuous regularization hyperparameters," in *International conference on Machine Learning*, 2016.

[37] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations*, 2019.

[38] T. Wang, J. Zhu, A. Torralba, and A. A. Efros, "Dataset distillation," *CoRR*, vol. abs/1811.10959, 2018.

[39] M. Ren, W. Zeng, B. Yang, and R. Urtasun, "Learning to reweight examples for robust deep learning," in *International conference on Machine Learning*, 2018.

[40] Y. Bengio, P. Frasconi, and P. Simard, "The problem of learning long-term dependencies in recurrent networks," in *IEEE International Conference on Neural Networks*, 1993.

[41] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, 1994.

[42] P. Parmas, C. E. Rasmussen, J. Peters, and K. Doya, "PIPPS: Flexible model-based policy search robust to the curse of chaos," vol. 80 of *Proceedings of Machine Learning Research*, (Stockholmsmässan, Stockholm Sweden), pp. 4065–4074, PMLR, 10–15 Jul 2018.

[43] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.

[44] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on Machine Learning*, 2013.

[45] S. Flennerhag, A. A. Rusu, R. Pascanu, F. Visin, H. Yin, and R. Hadsell, "Meta-learning with warped gradient descent," in *International Conference on Learning Representations*, 2020.

[46] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *BMVC*, 2016.

[47] H. Stackelberg, *The Theory Of Market Economy*. Oxford University Press, 1952.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019.

[49] L. N. Smith and N. Topin, "Super-convergence: Very fast training of residual networks using large learning rates," *CoRR*, vol. abs/1708.07120, 2017.

[50] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *UAI*, 2019.

[51] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, "Randaugment: Practical automated data augmentation with a reduced search space," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 3008–3017, 2020.

[52] A. Klein. https://automl.github.io/HpBandSter/build/html/quickstart.html, 2019.

[53] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," in *International Conference on Learning Representations*, 2020.

[54] "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp. 295–307, 1988.

[55] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the rprop algorithm," in *IEEE International Conference on Neural Networks*, pp. 586–591 vol.1, 1993.

[56] J. Bernstein, Y. Wang, K. Azizzadenesheli, and A. Anandkumar, "SIGNSGD: compressed optimisation for non-convex problems," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018* (J. G. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 559–568, PMLR, 2018.

[57] M. Safaryan and P. Richtárik, "On Stochastic Sign Descent Methods," *arXiv e-prints*, p. arXiv:1905.12938, May 2019.

8 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]
   (c) Did you discuss any potential negative societal impacts of your work? [No] We don't believe there are any
   (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? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] In Appendices
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] In Appendices

(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]

Appendices

A: Forward-mode Hypergradient Derivations

Recall that we are interested in calculating

$$\boldsymbol{Z}_t = \boldsymbol{A}_t \boldsymbol{Z}_{t-1} + \boldsymbol{B}_t$$

recursively during the inner loop, where

$$\boldsymbol{Z}_t = \frac{d\boldsymbol{\theta}_t}{d\boldsymbol{\lambda}} \qquad \boldsymbol{A}_t = \left.\frac{\partial \boldsymbol{\theta}_t}{\partial \boldsymbol{\theta}_{t-1}}\right|_{\boldsymbol{\lambda}} \qquad \boldsymbol{B}_t = \left.\frac{\partial \boldsymbol{\theta}_t}{\partial \boldsymbol{\lambda}}\right|_{\boldsymbol{\theta}_{t-1}}$$

so that we can calculate the hypergradients on the final step using

$$\frac{d\mathcal{L}_{val}}{d\boldsymbol{\lambda}} = \frac{\partial \mathcal{L}_{val}}{\partial \boldsymbol{\theta}_T} \boldsymbol{Z}_T$$

Each type of hyperparameter needs its own matrix $\boldsymbol{Z}_t$, and therefore its own matrices $\boldsymbol{A}_t$, and $\boldsymbol{B}_t$. Consider first the derivation of these matrices for the learning rate, namely $\boldsymbol{\lambda} = \boldsymbol{\alpha}$. Recall that the update rule of SGD with momentum and weight decay after substituting the velocity $\boldsymbol{v}_t$ in is

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha_t \left(\beta_t \boldsymbol{v}_{t-1} + \frac{\partial \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}} + \mu_t \boldsymbol{\theta}_{t-1} \right)$$

and therefore it follows directly that

$$\boldsymbol{A}_t^{\boldsymbol{\alpha}} = \mathbf{1} - \alpha_t \left(\frac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} + \mu_t \mathbf{1} \right)$$

The calculation of $\boldsymbol{B}_t^{\boldsymbol{\alpha}}$ is a bit more involved in our work because when using momentum, $\boldsymbol{v}_{t-1}$ is now itself a function of $\boldsymbol{\alpha}$. First we write

$$\begin{aligned}
\boldsymbol{B}_t^{\boldsymbol{\alpha}} &= -\beta_t \left(\frac{\partial \alpha_t}{\partial \boldsymbol{\alpha}} \boldsymbol{v}_{t-1} + \alpha_t \frac{\partial \boldsymbol{v}_{t-1}}{\partial \boldsymbol{\alpha}} \right) - \frac{\partial \alpha_t}{\partial \boldsymbol{\alpha}} \left(\frac{\partial \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}} + \mu_t \boldsymbol{\theta}_{t-1} \right) \\
&= -\beta_t \alpha_t \frac{\partial \boldsymbol{v}_{t-1}}{\partial \boldsymbol{\alpha}} - \delta_t^{\otimes} \left(\beta_t \boldsymbol{v}_{t-1} + \frac{\partial \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}} + \mu_t \boldsymbol{\theta}_{t-1} \right)
\end{aligned}$$

Now since

$$\boldsymbol{v}_t = \beta_t \boldsymbol{v}_{t-1} + \frac{\partial \mathcal{L}_{train}}{\boldsymbol{\theta}_{t-1}} + \mu_t \boldsymbol{\theta}_{t-1}$$

we can write the partial derivative of the velocity as an another recursive rule:

$$\begin{aligned}
\boldsymbol{C}_t^{\boldsymbol{\alpha}} &= \frac{\partial \boldsymbol{v}_t}{\partial \boldsymbol{\alpha}} \\
&= \beta_t \boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} + \frac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\alpha} \partial \boldsymbol{\theta}_{t-1}} + \mu_t \frac{\partial \boldsymbol{\theta}_{t-1}}{\partial \boldsymbol{\alpha}} \\
&= \beta_t \boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} + \left(\mu_t \mathbf{1} + \frac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} \right) \frac{\partial \boldsymbol{\theta}_{t-1}}{\partial \boldsymbol{\alpha}}
\end{aligned}$$

And putting all together recovers the system:

$$\begin{cases} \boldsymbol{A}_t^{\boldsymbol{\alpha}} = \mathbf{1} - \alpha_t \left(\dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} + \mu_t \mathbf{1} \right) \\ \boldsymbol{B}_t^{\boldsymbol{\alpha}} = -\beta_t \alpha_t \boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} - \delta_t^{\otimes} \left(\beta_t \boldsymbol{v}_{t-1} + \dfrac{\partial \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}} + \mu_t \boldsymbol{\theta}_{t-1} \right) \\ \boldsymbol{C}_t^{\boldsymbol{\alpha}} = \beta_t \boldsymbol{C}_{t-1}^{\boldsymbol{\alpha}} + \left(\mu_t \mathbf{1} + \dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} \right) \boldsymbol{Z}_{t-1}^{\boldsymbol{\alpha}} \end{cases}$$

For learning the momentum and weight decay, a very similar approach yields

$$\begin{cases} \boldsymbol{A}_t^{\boldsymbol{\beta}} = \mathbf{1} - \alpha_t \left(\dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} + \mu_t \mathbf{1} \right) \\ \boldsymbol{B}_t^{\boldsymbol{\beta}} = -\beta_t \alpha_t \boldsymbol{C}_{t-1}^{\boldsymbol{\beta}} - \delta_t^{\otimes} (\alpha_t \boldsymbol{v}_{t-1}) \\ \boldsymbol{C}_t^{\boldsymbol{\beta}} = \delta_t^{\otimes} (\boldsymbol{v}_t) + \beta_t \boldsymbol{C}_{t-1}^{\boldsymbol{\beta}} + \left(\mu_t \mathbf{1} + \dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} \right) \boldsymbol{Z}_{t-1}^{\boldsymbol{\beta}} \end{cases}$$

and

$$\begin{cases} \boldsymbol{A}_t^{\boldsymbol{\mu}} = \mathbf{1} - \alpha_t \left(\dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} + \mu_t \mathbf{1} \right) \\ \boldsymbol{B}_t^{\boldsymbol{\mu}} = -\beta_t \alpha_t \boldsymbol{C}_{t-1}^{\boldsymbol{\mu}} - \delta_t^{\otimes} (\alpha_t \boldsymbol{\theta}_{t-1}) \\ \boldsymbol{C}_t^{\boldsymbol{\mu}} = \delta_t^{\otimes} (\boldsymbol{\theta}_{t-1}) + \beta_t \boldsymbol{C}_{t-1}^{\boldsymbol{\mu}} + \left(\mu_t \mathbf{1} + \dfrac{\partial^2 \mathcal{L}_{train}}{\partial \boldsymbol{\theta}_{t-1}^2} \right) \boldsymbol{Z}_{t-1}^{\boldsymbol{\mu}} \end{cases}$$

B: Theorem 1: Proof

Preamble Consider that each time step $t \in \{1, 2, ..., T\}$ has a corresponding hyperparameter λ_t and hypergradient $g_t = \partial L_{val}(\boldsymbol{\theta}_T)/\partial \lambda_t$. Each g_t is viewed as a random variable due to the sampling process of the weight initialization $\boldsymbol{\theta}_0$ and the inner loop minibatch selection. Assume that $\boldsymbol{g} = [g_1, g_2, \ldots, g_T]$ is sufficiently well approximated by a Gaussian distribution, where $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with mean $\boldsymbol{\mu} = [\mu_1, \mu_2, ..., \mu_T]$ and covariance matrix $\boldsymbol{\Sigma}$. Assume that the values of $\boldsymbol{\mu}$ can be written as the ϵ-Lipschitz function $\mu_{t+1} = \mu_t + \epsilon_t$, where $\epsilon_t \in [-\epsilon, \epsilon]$. Note that in general, the gradients at different time steps may be correlated. Let the magnitude of the correlation be bounded by $c \in [-1, 1]$:

$$\frac{|\Sigma_{tt'}|}{\sqrt{\Sigma_{tt}\Sigma_{t't'}}} \leq c \qquad \forall \ t \neq t' \tag{8}$$

Let W define the size of a non-overlapping window over which hypergradients are averaged. This produces K windows, where each window $k \in \{1, 2, ..., K\}$ contains the time steps $S^{(k)}$ i.e. $S^{(1)} = \{1, 2, \ldots W\}$, $S^{(2)} = \{W+1, W+2, \ldots 2W\}$, etc. For simplicity of analysis [1] we assume the chosen window sizes are divisors of T such that $K = T/W$. Sharing hyperparameters over W contiguous time steps amounts to using the average hypergradient $\bar{g}^{(k)}$ for each step in that window, where

$$\bar{g}^{(k)} \coloneqq \frac{1}{W} \sum_{t \in S^{(k)}} g_t \tag{9}$$

We can now consider the mean squared error across all time steps when averaging contiguous hypergradients in non-overlapping windows of size W:

$$\mathrm{MSE}_W = \frac{1}{K} \sum_k \frac{1}{W} \sum_{t \in S^{(k)}} \mathbb{E}\left[\left(\bar{g}^{(k)} - \mu_t\right)^2\right] \tag{10}$$

where all expectations in our proof are over $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Note the case MSE_1 gives the standard case where no averaging occurs ($K = T$).

Theorem Then

$$\mathrm{MSE}_1 = \frac{1}{T} \sum_t \boldsymbol{\Sigma}_{tt} \tag{11}$$

$$\mathrm{MSE}_W \leq \frac{(1 + c(W-1))}{W} \mathrm{MSE}_1 + \epsilon^2 \frac{(W^2 - 1)}{12} \tag{12}$$

Proof The case for MSE_1 follows trivially from the definition of variance:

$$\mathrm{MSE}_1 = \frac{1}{T} \sum_t \mathbb{E}[(g_t - \mu_t)^2] = \frac{1}{T} \sum_t \boldsymbol{\Sigma}_{tt} \tag{13}$$

and so the mean squared error is the average of the variances. We now focus on the $W > 1$ case. Consider a window enumerated by k, and the vector of gradients within that window $\boldsymbol{g}^{(k)} = (g_t | t \in S^{(k)})$. Under the Gaussian assumption, that vector is Gaussian distributed with covariance $\boldsymbol{\Sigma}^{(k)}$, which is a block of the covariance matrix $\boldsymbol{\Sigma}$ corresponding to the variables in $\boldsymbol{g}^{(k)}$. Let $\bar{\mu}^{(k)} = (1/W) \sum_{t \in S^{(k)}} \mu_t$ be the average of means in window k. We consider the mean squared error from window k as :

[1] This assumption is unnecessary and can be relaxed but would result in a more cumbersome theorem statement, as the final window of size $< W$ would need to be considered.

$$\mathrm{MSE}^{(k)} = \frac{1}{W} \sum_{t \in S^{(k)}} \mathbb{E}\left[\left(\bar{g}^{(k)} - \mu_t\right)^2\right] \tag{14}$$

$$= \frac{1}{W} \sum_{t \in S^{(k)}} \mathbb{E}\left[\left(\bar{g}^{(k)} - \bar{\mu}^{(k)}\right)^2 + \left(\mu_t - \bar{\mu}^{(k)}\right)^2 - 2\left(\bar{g}^{(k)} - \bar{\mu}^{(k)}\right)\left(\mu_t - \bar{\mu}^{(k)}\right)\right] \tag{15}$$

$$= \mathbb{E}\left[\left(\bar{g}^{(k)} - \bar{\mu}^{(k)}\right)^2\right] + \frac{1}{W} \sum_{t \in S^{(k)}} \left(\mu_t - \bar{\mu}^{(k)}\right)^2 \tag{16}$$

Now $\bar{g}^{(k)} = (1/W)\mathbf{1}^T \boldsymbol{g}^{(k)}$, and $\bar{\mu}^{(k)} = (1/W)\mathbf{1}^T \boldsymbol{\mu}^{(k)}$, and so $\bar{g}^{(k)} - \bar{\mu}^{(k)} = (1/W)\mathbf{1}^T(\boldsymbol{g}^{(k)} - \boldsymbol{\mu}^{(k)})$. Hence

$$\mathbb{E}\left[\left(\bar{g}^{(k)} - \bar{\mu}^{(k)}\right)^2\right] = \frac{1}{W^2}\mathbb{E}\left[\mathbf{1}^T\left(\boldsymbol{g}^{(k)} - \boldsymbol{\mu}^{(k)}\right)\left(\boldsymbol{g}^{(k)} - \boldsymbol{\mu}^{(k)}\right)^T \mathbf{1}\right] \tag{17}$$

$$= \frac{1}{W^2}\mathbf{1}^T \boldsymbol{\Sigma}^{(k)} \mathbf{1} \tag{18}$$

Now let $\boldsymbol{D}$ be the diagonal matrix of variances, i.e. $\boldsymbol{D}_{ii} = \boldsymbol{\Sigma}_{ii}^{(k)} \ \forall i$ and $\boldsymbol{D}_{ij} = 0 \ \forall i \neq j$. We use the correlation bound (8), which can be written as $|\boldsymbol{\Sigma}_{ij}^{(k)}| < c\left[\boldsymbol{D}^{\frac{1}{2}}\mathbf{1}\mathbf{1}^T\boldsymbol{D}^{\frac{1}{2}}\right]_{ij} \ \forall i \neq j$, and this allows us to write an upper bound for the expression above:

$$\mathbb{E}\left[(\bar{g}^{(k)} - \bar{\mu}^{(k)})^2\right] \leq \frac{1}{W^2}\mathbf{1}^T[(1-c)\boldsymbol{D} + c\boldsymbol{D}^{\frac{1}{2}}\mathbf{1}\mathbf{1}^T\boldsymbol{D}^{\frac{1}{2}}]\mathbf{1} \tag{19}$$

$$= \frac{(1-c)}{W^2}\sum_i \boldsymbol{D}_{ii} + c\left[\frac{1}{W}\mathbf{1}^T\boldsymbol{D}^{\frac{1}{2}}\mathbf{1}\right]\left[\frac{1}{W}\mathbf{1}^T\boldsymbol{D}^{\frac{1}{2}}\mathbf{1}\right] \tag{20}$$

$$= \frac{(1-c)}{W^2}\sum_i \boldsymbol{D}_{ii} + c\left[\frac{1}{W}\sum_i \sqrt{\boldsymbol{D}_{ii}}\right]^2 \tag{21}$$

$$= \frac{(1-c)}{W^2}\sum_i \boldsymbol{\Sigma}_{ii}^{(k)} + c\left[\frac{1}{W}\sum_i \sqrt{\boldsymbol{\Sigma}_{ii}^{(k)}}\right]^2 \tag{22}$$

This expression can be simplified further using Jensen's inequality for square roots:

$$\mathbb{E}\left[(\bar{g}_k - \bar{\mu}_k)^2\right] \leq \frac{(1-c)}{W^2}\sum_i \boldsymbol{\Sigma}_{ii}^{(k)} + \frac{cW}{W^2}\sum_i \boldsymbol{\Sigma}_{ii}^{(k)} \tag{23}$$

$$= \frac{1 + c(W-1)}{W^2}\sum_i \boldsymbol{\Sigma}_{ii}^{(k)} \tag{24}$$

Now we return to the second part of (16). This second term can be bounded using the Lipschitz constraints. In particular for window size W, the maximum error is given when there is maximum deviation from the mean, which occurs when $\mu_t = \mu_{t-1} + \epsilon$. If we write the first mean in window k as $\mu_1^{(k)}$ we have $\mu_t = \mu_1^{(k)} + (t-1)\epsilon \ \forall t \in S^{(k)}$ and in that case $\bar{\mu}^{(k)} = \frac{1}{W}(\mu_1^{(k)} + (\mu_1^{(k)} + \epsilon) + (\mu_1^{(k)} + 2\epsilon) + \ldots + (\mu_1^{(k)} + (W-1)\epsilon) = \mu_1^{(k)} + \frac{(W-1)\epsilon}{2}$ and so $\mu_t - \bar{\mu}_k = (t-1)\epsilon + \frac{(W-1)\epsilon}{2}$. Note that this quantity is the same for all windows k. We can use it to write an upper bound as follows:

$$\frac{1}{W}\sum_{t\in S^{(k)}}\left(\mu_t-\bar{\mu}^{(k)}\right)^2 \leq \frac{1}{W}\sum_{j=1}^{W}\epsilon^2\left((j-1)-\frac{W-1}{2}\right)^2 \tag{25}$$

$$= \frac{\epsilon^2}{W}\sum_{j=0}^{W-1}\left(j-\frac{W-1}{2}\right)^2 \tag{26}$$

$$= \frac{\epsilon^2}{W}\sum_{j=0}^{W-1} j^2-(W-1)j+\frac{(W-1)^2}{4} \tag{27}$$

$$= \frac{\epsilon^2}{W}\left(\frac{W(W-1)(2W-1)}{6}-(W-1)\frac{W(W-1)}{2}+\frac{W(W-1)^2}{4}\right) \tag{28}$$

$$= \epsilon^2\frac{(W^2-1)}{12} \tag{29}$$

Hence combining (24) and (29) together into (16) we have

$$\mathrm{MSE}^{(k)} \leq \frac{1+c(W-1)}{W^2}\sum_i \mathbf{\Sigma}_{ii}^{(k)}+\epsilon^2\frac{(W^2-1)}{12} \tag{30}$$

and so incorporating it into (10) we get

$$\mathrm{MSE}_W = \frac{1}{K}\sum_{k=1}^{K}\mathrm{MSE}^{(k)} \tag{31}$$

$$\leq \frac{W}{T}\sum_{k=1}^{K}\left(\frac{(1+c(W-1))}{W^2}\sum_i \mathbf{\Sigma}_{ii}^{(k)}+\epsilon^2\frac{(W^2-1)}{12}\right) \tag{32}$$

$$= \frac{(1+c(W-1))}{WT}\sum_i \mathbf{\Sigma}_{ii}+\epsilon^2\frac{(W^2-1)}{12} \tag{33}$$

$$= \frac{(1+c(W-1))}{W}\mathrm{MSE}_1+\epsilon^2\frac{(W^2-1)}{12} \qquad \square$$

For sufficiently small ϵ and c we have with certainty, $MSE_W < MSE_1$ for some $W > 1$.

Discussion . We assume that means $\boldsymbol{\mu}$ can be written as the ϵ-Lipschitz function $\mu_{t+1} = \mu_t + \epsilon_t$, where $\epsilon_t \in [-\epsilon, \epsilon]$. Generally speaking, contiguous hyperparameters have optimal values which are close, and therefore have close hypergradients during outer optimization. This assumption breaks if hyperparameters are initialized randomly, and so we initialize all of our hyperparameters to zero in our experiments. Ideally, we would solve for whole inner loop several times so that we can use the mean hypergradient $[\mu_0, \mu_1, ..., \mu_H]$ for each individual step, without doing any sharing. While we consider this to be the optimal hypergradients, this is too expensive in practice, and instead we consider averaging hypergradients from neighbouring inner steps. The result above indicates that when contiguous hypergradients are sufficiently de-correlated (small c), we can reduce the mean squared error by a factor W compared to not averaging. However, if means μ_t drift over time by an amount $\epsilon_t \leq \epsilon$ this introduces some bias which increases the error and eventually results in $MSE_W > MSE_1$.

Finally, it is worth considering the simpler scenario when each hypergradient is considered to be drawn independently, i.e. $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{1})$. In that case, $c = 0$ and the mean squared errors become:

$$\mathrm{MSE}_1 = \sigma^2 \tag{34}$$

$$\mathrm{MSE}_W \leq \frac{\mathrm{MSE}_1}{W}+\epsilon^2\frac{(W^2-1)}{12} \tag{35}$$

Visualizing the MSE for various μ_t profiles . Since the mean squared error depends on the specific shape of μ_t, we sample random μ_t profiles and show how their MSE evolves as a function of W. This illustrates how tight the upper bound is.

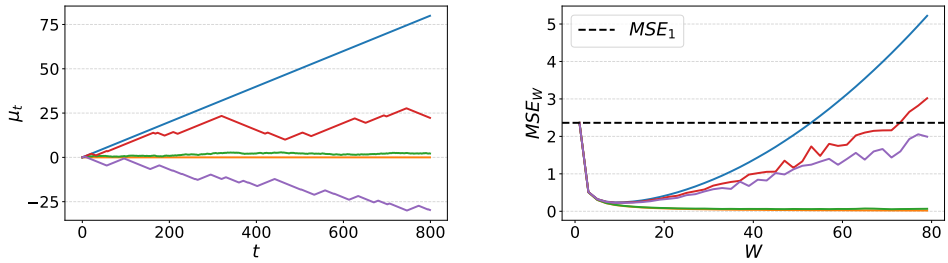

Figure 5: Several μ_t profiles and their corresponding mean squared error when sharing over W contiguous steps, as a function of W. The blue and yellow curve correspond to the maximal and minimal drifts scenarios respectively.

C: Implementation Details

We use a GeForce RTX 2080 Ti GPU for all experiments. We found that much of the literature on greedy methods uses the test set as the validation set, which creates a risk of meta-overfitting to the test set. Instead, we always carve out a validation set from our training set.

Figure 1 . Here we used very similar settings as Figure 4 for FDS, except we learned 7 learning rates to make the search space a bit more challenging. We use the HyperBanster HPO package for RS, BO, HB and BOHB. For HB and BOHB, the minimum budget argument is set to 1 epoch to allow for lots of fast evaluations, and the maximum budget is set to 50 epochs. We also use this technique to bring down our convergence time from ~ 10 hours to ~ 3 hours, namely we calculate hypergradients based on 10 epochs for some outer steps, rather than calculate all hypergradients on 50 epochs. Since HD needs the user to specify initial hyperparameter values, we random search over those for several consecutive HD runs.

Figure 2. We calculate the hypergradient with respect to some learning rate schedule over 100 seeds, where each seed corresponds to a different training set ordering and network initialization. The learning rate schedule is fixed, and initialized to be a cosine decay over the full 250 batches, starting at 0.01. The batch size is set to 128, and 1000 fixed images are used for the validation data.

Figure 3. Here we used a batch size of 128 for both datasets to allow 1 epoch worth of inner optimization in about 500 inner steps. Clipping was restricted to ± 3 to show the effect of noisy hypergradients more clearly. Since MNIST and SVHN are cheap datasets to run on a LeNet architecture, we can afford 50 outer steps and early stopping based on validation accuracy. All learning rates were initialized to zero.

Figure 4. We learn 5 values for the learning rates, 1 for the momentum and 1 for the weight decay, to make it comparable to the hyperparameters used in the literature for CIFAR-10. A batch size 256 is used, with 5% of the training set of each epoch set aside for validation. We found larger validation sizes not to be helpful. Hypergradient descent uses hyperparameters initialized at zero as well, and trains all hyperparameters online with an SGD outer optimizer with learning rate 0.2 and ± 1 clipping of the hypergradients. As described in appendix G, we used a sign based outer optimizer with adaptive step sizes rather than some hand-tuned outer learning rate schedule. We used initial values $\gamma_\alpha = 0.1, \gamma_\beta = 0.15$ and $\gamma_\mu = 4 \times 10^{-4}$ but the performance barely changed when these values were multiplied or divided by 2. Since we take 10 outer steps and initialize all hyperparameters at zero, this defines a search ranges: $\alpha \in [-1, 1]$, $\beta \in [-1.5, 1.5]$, and $\gamma \in [-4 \times 10^{-3}, 4 \times 10^{-3}]$. The Hessian matrix product is clipped to ± 10 to prevent one batch from having a dominating contribution to hypergradients.

D: Other hypergradient examples

Figure 2 depends on the value of $\boldsymbol{\alpha}$ at which hypergradients are calculated. For some learning rate schedules, contiguous hypergradients are sampled from closer distribution (ϵ small) and so sharing over larger windows is beneficial, as shown in the figure below.

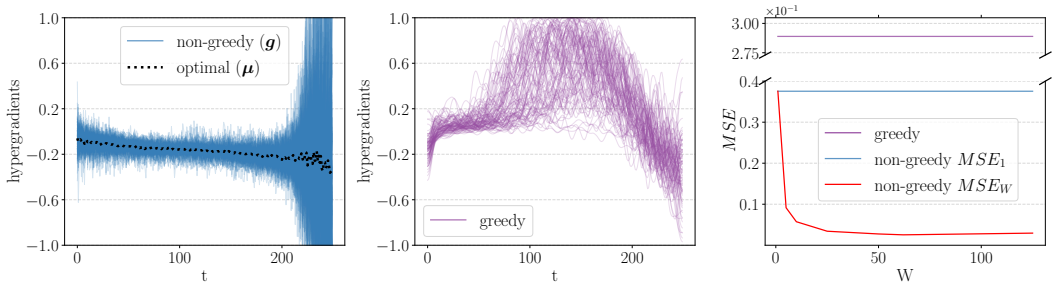

Figure 6: Similar to Figure 2 but for a smaller ϵ. We can see that averaging hypergradients helps even more.

E: Hypergradients

Here we provide the raw hypergradients corresponding to the outer optimization shown in Appendices: Figure 1. Note that the range of these hypergradients is made reasonable by the averaging of gradients coming from contiguous hyperparameters.

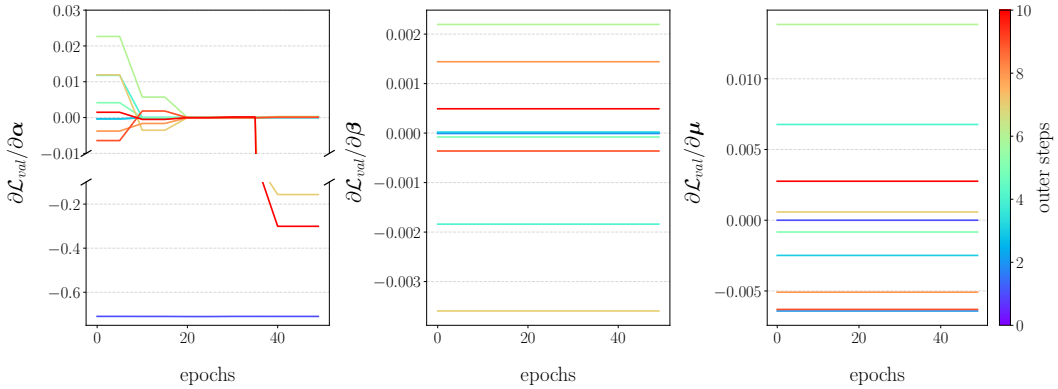

Figure 7: Hypergradients have a reasonable range but fail to always converge to zero when the validation performance stops improving.

F: Baselines

The objective here is to select the best hyperparameter setting that a deep learning practitioner would reasonably be expected to use, based on the hyperparameters used by the community for the datasets at hand. For CIFAR-10, the most common hyperparameter setting is the following: α is initialized at $\alpha_0 = 0.2$ (for batch size 256, as used in our experiments) and decayed by a factor $\eta = 0.2$ at $30\%, 60\%$ and 80% of the run (`MultiStep` in Pytorch); the momentum β is constant at 0.9, and the weight decay μ is constant at 5×10^{-4}. We search for combinations of hyperparameters around this setting. More specifically, we search over all combinations of $\alpha_0 = \{0.05, 0.1, 0.2, 0.4, 0.6\}$, $\eta = \{0.1, 0.2, 0.4\}$, $\beta = \{0.45, 0.9, 0.99\}$, and $\mu = \{2.5 \times 10^{-4}, 5 \times 10^{-4}, 1 \times 10^{-3}\}$. This makes up a total of 135 hyperparameter settings, which we each run 3 times to get a mean and standard deviation. The distribution of those means are provided in Appendices: Figure 8, and the best hyperparameter setting is picked based on validation performance. This is the value we report in Table 1 under *Hand-tuned (best)*.

MNIST and SVHN hyperparameters matter less, and in particular we observed no gain from using momentum and weight decay. The most popular learning rate schedules used for these datasets seem to be the cosine annealing one. We evaluate this schedule for $\alpha_0 = \{0.05, 0.1, 0.2, 0.4, 0.6\}$ and select the best hyperparameters based on validation performance.

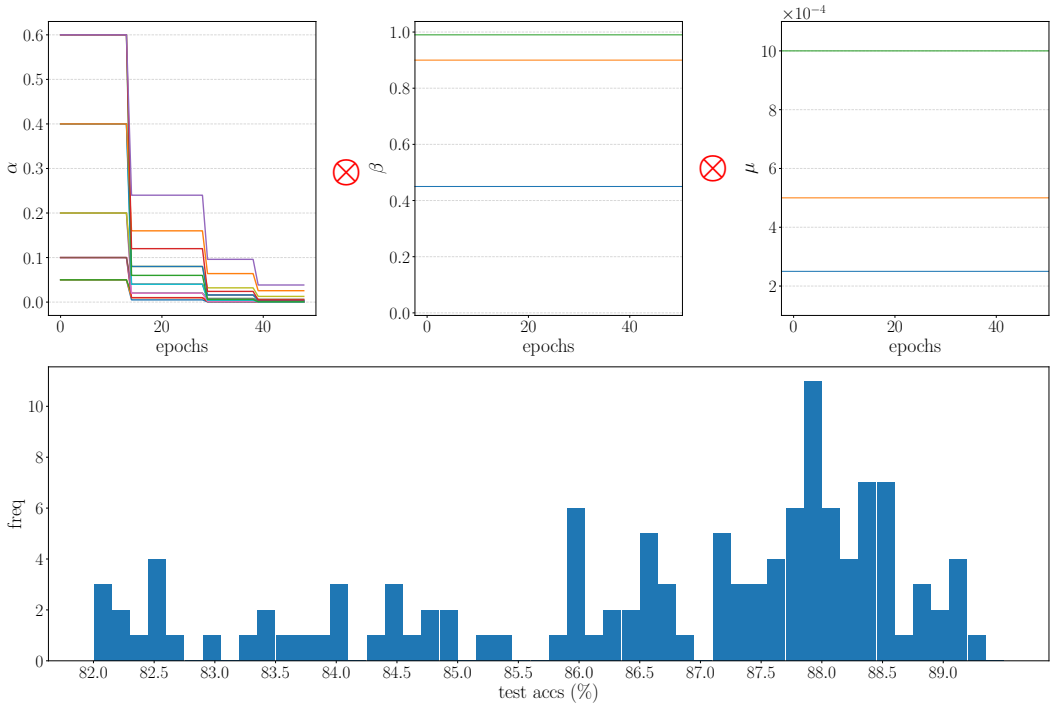

Figure 8: The combination of hyperparameters searched over for CIFAR-10 (top row) and the corresponding distribution of test accuracies (bottom row).

G: Using Hypergradient Signs

While hyperparameter sharing produces stable hypergradients with a reasonable range (see Appendix E), tuning the outer learning rate schedule can be tedious and unintuitive, and doesn't allow the user to specify a range to search hyperparameters over. A simple and fairly common learning rate schedule consists in decaying the learning rate (e.g. by a factor of 2) every time the gradient changes sign [54]. This frees us from having to tune the outer learning rate but fails to define a search range for The idea of using the sign of gradients to improve the efficiency of gradient descent dates back to the RPROP optimizer [55]. More recently, gradient signs have also been used to improve efficiency in the context of distributed learning [56], which has led to the discovery of robust convergence properties of sign-based SGD even in the case of biased gradients [57]. In this work, we observe that using the sign of hypergradients can also improve the efficiency of long-horizon meta-learning. We update hyperparameters with positive (negative) hypergradients by an amount $+\gamma$ ($-\gamma$), and γ is decayed by a factor of 2 every time the corresponding hypergradient changes sign across consecutive outer steps. This allows for convergence after hypergradients have changed signs a few times. Discarding hypergradient magnitudes makes it easier to tune the outer optimizer since we don't have to set an outer learning rate schedule, but most importantly it allows us to define the range of hyperparameter search more explicitly. This is especially useful to compare FDS to other HPO algorithms which also use a fixed search range (section 5.3).

Algorithm 2 FDS algorithm when using an outer optimizer based on the sign of hypergradients

Initialize: $N^\alpha, W = T/N^\alpha, \boldsymbol{\alpha} = \mathbf{0}^{N^\alpha}$

#outer loop
for o in $1, 2, ...$ **do**
  Initialize: $\mathcal{D}_{train}, \mathcal{D}_{val}, \boldsymbol{\theta}_0 \in \mathbb{R}^D$,
  $Z^{\boldsymbol{\alpha}} = \mathbf{0}^{D \times N^\alpha}, C^{\boldsymbol{\alpha}} = \mathbf{0}^{D \times N^\alpha}$

  #inner loop
  for t in $1, 2, ..., T$ **do**
    $\boldsymbol{x}_{train}, \boldsymbol{y}_{train} \sim \mathcal{D}_{train}$
    $\boldsymbol{g}_{train} = \partial \mathcal{L}_{train}(\boldsymbol{x}_{train}, \boldsymbol{y}_{train}) / \partial \boldsymbol{\theta}$

    #hyperparameter sharing
    $i = \lfloor t/W \rfloor$
    $\mathcal{H} Z^{\boldsymbol{\alpha}}_{[0:i]} = \partial(\boldsymbol{g}_{train} Z^{\boldsymbol{\alpha}}_{[0:i]}) / \partial \boldsymbol{\theta}$
    $Z^{\boldsymbol{\alpha}}_{[0:i]} = A^{\boldsymbol{\alpha}} Z^{\boldsymbol{\alpha}}_{[0:i]} + B^{\boldsymbol{\alpha}}_{[0:i]}$

    update $C^{\boldsymbol{\alpha}}$ (Eq 5)
    $\boldsymbol{\theta}_{t+1} = \Phi(\boldsymbol{\theta}_t)$

  end for

  $\boldsymbol{g}_{val} = \partial \mathcal{L}_{val}(\mathcal{D}_{val}) / \partial \boldsymbol{\theta}$
  $\boldsymbol{s}_o = \mathrm{sgn}(\boldsymbol{g}_{val} Z^{\boldsymbol{\alpha}} / W)$

  for n in $1, 2, ..., N_\alpha$ **do**
    if $\boldsymbol{s}_{o,[n]} \neq \boldsymbol{s}_{o-1,[n]}$ **then**
      $\boldsymbol{\gamma}^{\boldsymbol{\alpha}}_{[n]} \leftarrow \boldsymbol{\gamma}^{\boldsymbol{\alpha}}_{[n]} / 2$
    end if
  end for

  $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \boldsymbol{s}_o \odot \boldsymbol{\gamma}^{\boldsymbol{\alpha}}$

end for