DrMAD: Distilling Reverse-Mode Automatic Differentiation for Optimizing Hyperparameters of Deep Neural Networks

Jie Fu[1], Hongyin Luo[2], Jiashi Feng[3] and Tat-Seng Chua[4]

[1]Graduate School for Integrative Sciences and Engineering, National University of Singapore
[2]Department of Computer Science and Technology, Tsinghua University
[3]Department of Electrical and Computer Engineering, National University of Singapore
[4]School of Computing, National University of Singapore

Abstract

The performance of deep neural networks is well known to be sensitive to the setting of their hyperparameters. Recent advances in reverse-mode automatic differentiation allow for optimizing hyperparameters with gradients. The standard way of computing these gradients involves a forward and backward pass of computations. However, the backward pass usually needs to consume unaffordable memory to store all the intermediate variables to *exactly* reverse a training procedure. In this work we propose a new method, DrMAD, to distill the knowledge of the forward pass into an shortcut path, through which we *approximately* reverse the training trajectory. Experiments on MNIST dataset show that DrMAD reduces memory consumption by 4 orders of magnitude for optimizing hyperparameters without sacrificing its effectiveness. To the best of our knowledge, DrMAD is the first research attempt to automatically tune *hundreds of thousands* of hyperparameters of deep neural networks in practice.

1 Introduction

Modern machine learning algorithms are rarely hyperparameter-free: hyperparameters determining the learning rate or how important the $L2$-norm penalties are during training. Recent results [21, 5, 28] show that the performance of large-size deep models is sensitive to the setting of their hyperparameters. Tuning hyperparameters of deep neural networks is thus now recognized as a crucial step in the process of applying machine learning algorithms to achieve best

performance and drive industrial applications [24]. For decades, the de-facto standard for hyperparameter tuning in machine learning has been a simple grid search [24]. Recently, it has been shown that hyperparameter optimization, a principled and efficient way of automatically tuning hyperparameters, can reach or surpass human expert-level hyperparameter settings for deep neural networks to achieve new state-of-the-art performance in a variety of benchmark datasets [24, 25, 28].

The common choice for hyperparameter optimization is gradient-free Bayesian optimization [6]. Bayesian optimization builds on a probability model[1] for $P(\text{validation\_loss}|\text{hyperparameter})$ that is obtained by updating a prior from a history H of {hyperparameter, validation_loss} pairs. This probability model is then used to optimize the validation loss after complete training of the model's elementary[2] parameters. Although those techniques have been shown to achieve good performance with a variety of models on benchmark datasets [24], they can hardly scale up to handle more than 20 hyperparameters[3] [18, 24]. Due to this inability, hyperparameters are often considered nuisances, indulging researchers to develop machine learning algorithms with fewer of them. We argue that being able to richly hyperparameterize our models is more than a pedantic trick. For example we can set a separate $L2$-norm penalty for each layer[4], which has been shown to improve the performance of deep models on several benchmark datasets [25].

On the other hand, automatic differentiation (AD), as a mechanical transformation of a objective function, can calculate gradients with respect to hyperparameters (thus called hypergradients) accurately [2, 18]. Although hypergradients enable us to optimize thousands of hyperparameters, all the prior attempts [3, 2, 18] insist on *exactly* tracing the training trajectory backwards, which are unfeasible for real-world data and deep models from a memory perspective. Suppose we are to train a neural networks on MNIST with 60,000 training samples, the mini-batch size is 100, the epoch number is 200, and every elementary parameter vector takes up 0.1 GB. In order to trace the training trajectory backwards, the naïve solution has to store all the intermediate variables (e.g. weights) at every iteration thus costing memory up to $60000/100 \times 200 \times 0.1$ GB $= 12$ TB. The improved method proposed in [18] needs at least 60 GB memory, which still cannot utilize the power of modern GPUs[5] even for this small-scale dataset, let alone ImageNet dataset consisting of millions of training samples.

The high memory cost roots in pursuing exact reverse of the training procedure. In this work, we propose DrMAD, to reduce the memory cost and make the hyperparameter optimization feasible in practice. Compared with exact reverse methods, DrMAD chooses to reverse training dynamics in an *approximate*

[1] Gaussian processes are the most popular choice as the regressor. Except in [4], where they use a random forest and in [26] where they use a Bayesian neural network.

[2] Following the tradition in [18], we use *elementary* to unambiguously denote the *traditional* parameters updated by back-propagation, e.g. weights and biases in a neural network.

[3] Here we mean *effective* hyperparameters, as has been shown in [30], Bayesian optimization can handle high-dimensional inputs only if the number of effective hyperparameters is small.

[4] The winning deep model for ILSVRC2015 has 152 layers [15].

[5] Till the writing of this paper, the most advanced GPU is equipped with 24GB memory.

manner. Doing so allows us to reduce the memory consumption of tuning hyperparameters by a factor of 50,000 at least. On the MNIST dataset, our method only needs 0.2 GB memory thus enabling the use of GPUs. More importantly, the memory consumption is independent of the problem size as long as the deep model has converged[6]. In addition, DrMAD only incurs negligible performance drop. Section 2 and Section 3 describe this problem and our solution in detail respectively, which is the main technical contribution of this paper.

In short, we make the following contributions in this work:

- We give an algorithm that approximately reverses stochastic gradient descent to compute gradients w.r.t. hyperparameters. Our method can reduce memory consumption by orders of magnitude without sacrificing its effectiveness compared with the previous attempts [18, 9] based on exact arithmetic.
- We show that DrMAD can be further accelerated when combined with checkpointing, a standard technique in reverse-mode automatic differentiation.
- We give some suggestions on how to prevent overfitting faced by high-dimensional hyperparameter optimization.

2 Foundations and Related Work

We first review the general framework of automatic hyperparameter tuning and previous work on automatic differentiation for hyperparameters of machine learning models.

2.1 Automatic Hyperparameter Tuning

Typically, a deep learning algorithm $\mathcal{A}_{w,\lambda}$ has a vector of elementary parameters $\boldsymbol{w} = \{w_1, ..., w_m\} \in \boldsymbol{W}$, where $\boldsymbol{W} = W_1 \times ... \times W_m$ define the parameter space, and a vector of hyperparameters $\boldsymbol{\lambda} = \{\lambda_1, ..., \lambda_n\} \in \boldsymbol{\Lambda}$, where $\boldsymbol{\Lambda} = \Lambda_1 \times ... \times \Lambda_n$ define the hyperparameter space. We further use $l_{train} = \mathcal{L}(\mathcal{A}_{w,\lambda}, \mathcal{X}_{train})$ to denote the training loss, and $l_{valid} = \mathcal{L}(\mathcal{A}_{w,\lambda}, \mathcal{X}_{train}, \mathcal{X}_{valid})$ to denote the validation loss that $\mathcal{A}_{w,\lambda}$ achieves on validation data $\mathcal{X}_{valid}$ when trained on training data $\mathcal{X}_{train}$. An automatic hyperparameter tuning algorithm then tries to find $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ that minimizes l_{valid} in an efficient and principled way.

2.2 Automatic Differentiation

Most training of machine learning models is driven by the evaluation of derivatives, which are usually handled by four approaches [2]: numerical differentia-

[6]We only require that the deep model converges, but not necessarily achieves highest performance. This can be easy to be met in practice, as will be discussed in detail in Section 3.1.

	Forward Pass			Backward Pass	
v_{-1}	$= x_1$	$= 3$	$\bar{x}_1$	$= \bar{v}_{-1}$	$= 6$
v_0	$= x_2$	$= 6$	$\bar{x}_2$	$= \bar{v}_0$	$= 0.02$
v_1	$= \ln(v_0)$	$= \ln(6)$	$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= 6$
v_2	$= v_{-1}^2$	$= 3^2$	$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_1 \frac{\partial v_1}{\partial v_0}$	$= 0.02$
v_3	$= \cos(v_0)$	$\cos(3)$	$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= -0.14$
v_4	$= v_1 + v_2$	$= 1.79 + 9$	$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= 1$
			$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= 1$
v_5	$= v_4 + v_3$	$= 10.79 - 0.98$	$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= 1$
			$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= 1$
y	$= v_5$	$= 9.80$	$\bar{v}_5$	$= \bar{y}$	$= 1$

Table 1: Reverse-mode automatic differentiation example, with $y = f(x_1, x_2) = \ln(x_2) + x_1^2 + \cos(x_1)$ at $(x_1, x_2) = (3, 6)$. Setting $\bar{y} = 1$, $\partial y/\partial x_1$ and $\partial y/\partial x_2$ are computed in one backward pass.

tion; manually calculating the derivatives and coding them; symbolic differentiation by computer algebra; and automatic differentiation.

Manual differentiation can avoid approximation errors and instability associated with numerical differentiation, but is labor intensive and prone to errors [14]. Although symbolic differentiation could address weakness of both numerical and manual methods, it has the problem of "expression swell" and not being efficient at run-time[14]. AD has been underused [2], if not unknown, by the machine learning community despite its extensive use in other fields, such as real-parameter optimization [29] and probabilistic inference [19]. AD systematically applies the chain rule of calculus at the elementary operator level [14]. It also guarantees the accuracy of evaluation of derivatives with a small constant factor of computational overhead and ideal asymptotic efficiency [2].

AD has two modes: forward and reverse[7] [14]. Here we only consider the reverse-mode automatic differentiation (RMAD) [2]. RMAD is a generalization of the back-propagation [12] used in the deep learning community[8]. RMAD allows the gradient of a scalar loss with respect to its parameters to be computed in a single backward pass after a forward pass [2]. Table 1 shows an example of RMAD for $y = f(x_1, x_2) = \ln(x_2) + x_1^2 + \cos(x_1)$.

2.3 Gradient-Based Methods for Hyperparameters

Although our method could work in principle for any continuous hyperparameter, in this paper we focus on studying the tuning of regularization hyperparameters, which only appear in the penalty term. We consider stochastic gradient descent (SGD), as it is the only affordable way of optimizing large-size neural

[7]Do not confuse these two modes with the forward and backward passes used in reverse-mode automatic differentiation, as will be described in detail shortly.

[8]One of the most popular deep learning libraries, Theano [1], can be described as a limited version of RMAD and a heavily optimized version of symbolic differentiation [2].

networks [12]. To make the definitions in Section 2.1 more concrete and concise, we denote the training objective function as:

$$l_{train} = \mathcal{L}(\boldsymbol{w}|\boldsymbol{\lambda}, \mathcal{X}_{train}) = C(\boldsymbol{w}|\boldsymbol{\lambda}, \mathcal{X}_{train}) + P(\boldsymbol{w}, \boldsymbol{\lambda}) = C_{train} + P(\boldsymbol{w}, \boldsymbol{\lambda}), \quad (1)$$

where $\boldsymbol{w}$ is the vector of elementary parameters (including weights and biases), $\mathcal{X}_{train}$ is the training dataset, $\boldsymbol{\lambda}$ is the vector of hyperparameters, $C(\cdot)$ is the cost function on either training (denoted by C_{train}) or validation (denoted by C_{valid}) data, and $P(\cdot)$ is the penalty term.

The elementary parameters updating formula is: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta_{\boldsymbol{w}} \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}_t|\boldsymbol{\lambda}, \mathcal{X}_{train})$, where the subscript t denotes the count of iteration (i.e. one forward and backward pass over one mini-batch), and $\eta_{\boldsymbol{w}}$ is the learning rate for elementary parameters.

The gradients of hyperparameters (hypergradients) are computed on the validation data $\mathcal{X}_{valid}$ without considering the penalty term [10, 18, 17]:

$$\nabla_{\boldsymbol{\lambda}} C_{valid} = \nabla_{\boldsymbol{w}} C_{valid} \frac{\partial \boldsymbol{w}_t}{\partial \boldsymbol{\lambda}} = \nabla_{\boldsymbol{w}} C_{valid} \frac{\partial^2 l_{train}}{\partial \boldsymbol{\lambda} \partial \boldsymbol{w}}, \quad (2)$$

where $C_{valid} = C(\boldsymbol{w}|\mathcal{X}_{valid})$ is the validation cost.

The hyperparameters are updated at every iteration in [17, 10]. In [10], given the elementary optimization has converged, the hyperparameters are updated as:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \eta_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{w}} C_{valid} (\nabla^2_{\boldsymbol{w}} l_{train})^{-1} \frac{\partial^2 l_{train}}{\partial \boldsymbol{\lambda} \partial \boldsymbol{w}}, \quad (3)$$

where $\eta_{\boldsymbol{\lambda}}$ is the learning rate for hyperparameters. The authors in [17] propose to update hyperparameters by simply approximating the Hessian in Eq. 3 as $\nabla^2_{\boldsymbol{w}} l_{train} = I$:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \eta_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{w}} C_{valid} \frac{\partial^2 l_{train}}{\partial \boldsymbol{\lambda} \partial \boldsymbol{w}}. \quad (4)$$

However, updating hyperparameters at every iteration might result in unstable hypergradients, because it only considers the influence of the regularization hyperparameters on the current elementary parameter update. Consequently, this approach can hardly scale up to handle more than 20 hyperparameters as shown in [17].

In this paper, we follow the direction of [18, 3, 9], in which RMAD is used to compute hypergradients, taking into account the effects of the hyperparameters on the entire learning trajectory. Specifically, different from Eq. 2 in [17, 10] only considering $\frac{\partial \boldsymbol{w}_t}{\partial \boldsymbol{\lambda}}$, we consider the term $\frac{\partial \boldsymbol{w}_T}{\partial \boldsymbol{\lambda}}$ (here T represents the final iteration number till convergence) similar to [18, 3, 9]: $\boldsymbol{w}_T = \sum_{0<t<T} \triangle \boldsymbol{w}_{t,t+1}(\boldsymbol{w}_t(\boldsymbol{\lambda}_k), \boldsymbol{\lambda}_k, \mathcal{X}_t, \eta_{\boldsymbol{\lambda}}) + \boldsymbol{w}_0$, where the subscript k in $\boldsymbol{\lambda}_k$ stands for the counter of meta-iterations used for hyperparameter optimization (i.e. the number of entire training of elementary parameters), $\mathcal{X}_t$ is the mini-batch of training data used in iteration t, and $\boldsymbol{w}_0$ is the initial parameter vector. Update of hyperparameter in this paper and also [18, 3, 9] is:

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \eta_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{w}} C_{valid} \frac{d}{d\boldsymbol{\lambda}} (\sum_{0<t<T} \triangle \boldsymbol{w}_{t,t+1}(\boldsymbol{w}_t(\boldsymbol{\lambda}_k), \boldsymbol{\lambda}_k, \mathcal{X}_t, \eta_{\boldsymbol{\lambda}}) + \boldsymbol{w}_0). \quad (5)$$

Unfortunately, RMAD requires all the intermediate variables obtained in the forward pass be maintained in memory for the backward pass [14]. Conventional RMAD with exact arithmetic stores the entire training trajectory $\{\boldsymbol{w}_0, ..., \boldsymbol{w}_T\}$ in memory, which is totally impractical for even small-size tasks.

An "information buffer" method is proposed in [18] to recompute the learning trajectory on the fly during the reverse pass of RMAD rather than storing it in memory by making use of the SGD momentum mechanism. Certain amount of auxiliary bits as information buffer, which depend on the specific learning dynamics, to handle finite precision arithmetic is needed in this case. Although the proposed method in [18] is shown to be able to tune hundreds of thousands of hyperparameters and reduce the memory consumption by a factor of 200 (in the most ideal setting), it can only work on small-size datasets[9].

3 Approximating RMAD with a Shortcut

In this paper, we raise a crucial yet rarely investigated question: do we really need to *exactly* trace the whole training procedure backwards, starting from the trained parameter values and working back to the initial random parameters? If all that we care about are *decent* enough hyperparameters, the answer might be no. Driven by this question, we demonstrate how to establish a shortcut by distilling the knowledge of the forward pass of RMAD.

3.1 Distilling Knowledge from the Forward Pass

Trying to trace backwards exactly can be wasteful as this approach does not take into account the highly structured nature of deep models training dynamics. It has been argued in [13] that if we knew the direction defined by the final learned weights after convergence, a single coarse line search could do a good job of training a neural network. The results in [13] are consistent with recent empirical and theoretical work arguing that local minima are not a significant problem for training deep neural networks [7, 8].

It is also well known that when doing elementary parameter optimization, second-order methods such as Newton's method are more efficient with fewer iterations compared to the naïve gradient descent, but they cannot easily applied to high-dimensional models due to heavy computations with large matrices. In practice, it is usually approximated by a diagonal or block-diagonal approximation [22]. Motivated by these approximation techniques, we make an aggressive approximation for RMAD here – discard all the intermediate variables altogether. In other words, we choose a shortcut, which simply approximates the

[9] In the experimental part of [18], they only use 10,000 training samples of MNIST dataset and the number of iterations is 100, which are far from the real-world settings.

forward pass learning history used in Eq. 5 as a series of parameter vectors $\boldsymbol{w} = (1-\beta)\boldsymbol{w}_0 + \beta\boldsymbol{w}_T$ for varying values of $0 < \beta < 1$, which can be generated on the fly almost without storing anything.

More concretely, DrMAD works by first obtaining the final trained elementary parameter values using SGD algorithms. Alg. 1 demonstrates the procedure formally. No that we could use any SGD variants here and do not put constraints on the momentum term. But the previous most similar work [18] is highly dependent on the momentum setting in order to save memory.

Then Alg. 2 shows how to compute the gradients of hyperparameters by DrMAD , where in step 4 we approximate the learning dynamics. These hypergradients are used to update hyperparameters using Eq. 5. In addition to reduction in memory, compared to [18], Alg. 2 also reduces the computational operations as a byproduct, because it does not to recompute the elementary parameters exactly.

Algorithm 1 Stochastic gradient descent (SGD).

1: **inputs:** initial $\boldsymbol{w}_1$, fixed learning rate α, fixed decay $\boldsymbol{\gamma}$, hyperparameters $\boldsymbol{\lambda}$, train loss function l_{train}
2: initialize $\boldsymbol{v}_1 = 0$
3: **for** $t = 1$ to $T-1$ **do**
4: $\quad \boldsymbol{g}_t = \nabla_{\boldsymbol{w}} l_{train}$ //evaluate gradient
5: $\quad \boldsymbol{v}_{t+1} = \gamma \boldsymbol{v}_t - (1-\gamma)\boldsymbol{g}_t$
6: $\quad \boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha_t \boldsymbol{v}_t$ //update position
7: **end for**
8: **output:** trained parameters $\boldsymbol{w}_T$

Algorithm 2 Distilling reverse-mode automatic differentiation (DrMAD) of SGD.

1: **inputs:** initial $\boldsymbol{w}_0$, learned $\boldsymbol{w}_T$, training loss l_{train}, validation loss l_{valid}
2: initialize $d\boldsymbol{\lambda} = 0$, $\gamma = 0.1$, $\beta_t = 0.01$ where $0 < \beta_t < 1$, $d\boldsymbol{w} = \nabla_{\boldsymbol{w}} l_{valid}$
3: **for** $t = T-1$, **counting down to 1** **do**
4: $\quad \boldsymbol{w}_{t-1} = (1-\beta_t)\boldsymbol{w}_0 + \beta_t \boldsymbol{w}_T$ //approximate $\boldsymbol{w}_{t-1}$ on the fly
5: $\quad d\boldsymbol{v} = d\boldsymbol{v} + \alpha d\boldsymbol{w}$
6: $\quad d\boldsymbol{\lambda} = d\boldsymbol{\lambda} - (1-\gamma) d\boldsymbol{v} \nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{w}} l_{valid}$
7: **end for**
8: **output:** gradient of l_{valid} w.r.t. $\boldsymbol{\lambda}$

Actually, DrMAD can be seen as a form of knowledge distillation [16]. It has been show that the knowledge acquired by a large ensemble of large-size models, the "teacher", can be *distilled* into a single small model, the "student" [16]. For example we can first form a teacher by training an ensemble of five 10-layer deep neural networks on a dataset. After this, a student with 5-layer could

achieve almost the same accuracy as the teacher with much less parameters by approximating the teacher's behavior rather than being trained on the original dataset from scratch.

In our situation, the most valuable knowledge is the initial random and final trained weight vectors once the deep model has converged. Therefore, the forward pass of SGD training provides the final weight vector after convergence, which is defined as a teacher; The reverse pass given by shortcut is a student here.

We should emphasize that our convergence requirement is reasonable, as it has been demonstrated that modern deep neural networks are relatively easy to converge in practice [7, 13].

DrMAD is also in essence similar to transfer learning based hyperparameter optimization methods [28]. In transfer Bayesian hyperparameter optimization, one can explore a wide range of hyperparameters on a auxiliary dataset, and then transfer this knowledge to quickly find the suitable hyperparameters on another dataset with less optimization time [28]. DrMAD is different from transfer Bayesian optimization in that it always runs on the same dataset.

Obviously, obtaining derivatives from the shortcut may never reveal more information about hyperparameters than calculating derivatives from the exact trajectories. However, when (memory and computational) costs are taken into account, the shortcut may convey more information per unit cost. In fact, our approach explicitly separates the optimization of the elementary parameters and hyperparameters, which serves as a trade-off between accuracy and computational expense. In the experimental part, we will show that the accuracy performance of DrMAD is slightly worse than the RMAD with exact arithmetic. Furthermore, by separating the optimization of the elementary parameters and hyperparameters, we can use existing GPU-based deep learning libraries, such as Theano [1], to speed up the forward pass, because currently none of the automatic differentiation libraries, such as Autograd[10], support GPUs.

3.2 Approximate Checkpointing with DrMAD

The computations of hypergradients are dependent on their hyperparameters through thousands of iterations of SGD. Furthermore, within each iteration of SGD, it involves forward- and then back-propagations through a deep neural network. Overall, the stacking of all the above operations would result in vanishing gradient problems [11]. Even worse, when optimizing elementary parameters, we typically have thousands of iterations to generate gradients; while for hyperparameters, due to the limited computing budget, it cannot get access to more than 50 meta-iterations in practice, which might not be enough to drive the hyperparameter search effectively.

Checkpointing, as a standard way to save memory in RMD, only stores the intermediate variables on a fraction of the training steps, and recomputes the missing steps of the forward training procedure as needed during the backward

[10] https://github.com/HIPS/autograd

pass [18]. However, this would still need too much memory to be useful for large-size neural networks [18]. Furthermore it can hardly trace back from the trained elementary parameters to the initial random ones, thus giving rise to unstable hyperparameter optimization process.

In contrast, when combining checkpointing with DrMAD, we can more safely decrease the number of backward iterations. Checkpointing serves as a hypergradient damper here. Empirically, we observe that DrMAD can achieve almost the same test error on MNIST dataset after retaining only 10% of the backward iterations.

3.3 Prevent Overfitting

Tuning hundreds of thousands of hyperparameters with Eq. 5 increases the risk of overfitting the validaiton dataset. This is similar to optimizing too many elementary parameters which might result in overfitting the training dataset. One rule of thumb as suggested in [18] is to use the size of the validation dataset as a rough guide to determine how many hyperparameters should be optimized. For example, if one has 10,000 validation datapoints at hand, he may have the luxury to optimize 1,000 hyperparameters.

Analogous to dropout commonly used when optimizing elementary parameters in deep neural networks [27], one slightly more principled way for avoiding overfitting in hyperparameter optimization would be to randomly drop out some hyperparameters during training. We call this method as *hyper-drop*. In the simplest situation, each hyperparameter is retained for subsequent optimization process with a fixed probability p independent of each other. In other words, p of the hyperparameters are randomly chosen to be optimized by DrMAD, whereas the other $(1-p)$ hyperparameters are left unchanged. To do so, we need to modify the penalty term, $P(\boldsymbol{w}, \boldsymbol{\lambda})$, in Eq. 1. Specifically, we use $\tilde{\boldsymbol{\lambda}} = \boldsymbol{r} * \boldsymbol{\lambda}$ to replace the original $\boldsymbol{\lambda}$, where $*$ denotes an element-wise product. We also set $\boldsymbol{r}_n \sim \text{Bernoulli}(p)$, where the subscript n in $\boldsymbol{r}_n$ is the number of hyperparameters. However, like dropout [27], hyper-dropout might need more meta-iterations to converge.

We might also choose to share the same hyperparameters across several different elementary parameters, which is similar to weight-tying [20]. Doing so enables us to explicitly trade-off the flexibility given by high-dimensional hyperparameters and low-risk of overfitting.

We have not tested the above methods thoroughly, and will leave it for future work.

4 Experiments

We evaluted DrMAD for optimizing continuous regularization hyperparameters on MNIST dataset (50,000 for training, 10,000 as validation, and 10,000 for testing) using a multilayer perceptron (MLP) with tanh activation function. The MLP has one hidden layer with 500 neurons, and each neuron has its own

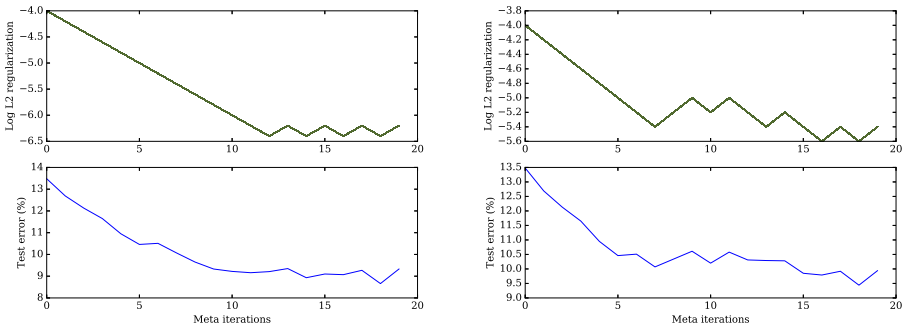

Figure 1: *Left:* Evolution of the average of hyperparameters and the corresponding test error obtain by RMAD. *Right:* Evolution of the average of hyperparameters and the corresponding test error obtain by DrMAD. We can observe that both the hyperparameter and test error curves of DrMAD is close to RMAD on MNIST dataset during hyperparameter optimization.

$L2$-norm penalty, thus 500 hyperparameters. The learning rate for elementary parameters is 0.01, the number of elementary iterations is 1000, the learning rate for hyperparameters is 0.1, the mini-batch size is 20, and the number of meta-iterations is 20. Data pre-processing only includes centering each feature.

It should be noted that we set the mini-batch size to a very small number and the number of ememtary iterations to a large number on purpose. Therefore, we can see if DrMAD can approxiate this highly zigzag and long learning trajectory.

Code for all experiments is modified from https://github.com/HIPS/hypergrad. We will open-source our modification soon.

Fig. 1 shows the evolution of hyperparameter values during optimization and the corresponding test error, using RMAD and DrMAD respectively.

Clearly, even the RMAD with exact arithmetic introduced in [18] can only achieve a test error of 8.7%. Because we put too much weight on penalty term $P(\boldsymbol{w}, \boldsymbol{\lambda})$ in Eq. 1 deliberately. Here we do not aim to provide state-of-the-art performance on MNIST, but to show that DrMAD can in principle optimize thousands of hyperparameters and provide similar performance as RMAD. Doing so inevitably incurs serious overfitting as expected.

5 Discussion and Conclusion

In this paper, we proposed a memory efficient procedure called distilling reverse-mode automatic differentiation (DrMAD) for gradient-based automatic optimization of continuous hyperparameters of deep neural networks. We showed how DrMAD allows the optimization of validation loss w.r.t. hundreds of thousands of hyperparameters in practice, which was previously impossible due to

the extremely huge memory consumption.

Due to the explicit decoupling of elementary and hyper- parameters, the shortcut trajectory would be different from the actual learning trajectory, thus loosing the ability to tune learning rate schedules suitable only for the original learning trajectory. However, the stability of hypergradients is sensitive to the learning rates schedule. Fortunately, there are many alternative learning rate schedulers (e.g. [23]) or we can tune the learning rates using Bayesian optimization [28].

Acknowledgments

Jie Fu would like to thank NVIDIA for GPU donation. Jie Fu would also like to thank Microsoft Azure for Research for providing the computational resources under the Windows Azure for Research Award program.

References

[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

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

[4] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms, 2013.

[5] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of The 30th International Conference on Machine Learning*, pages 115–123, 2013.

[6] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[7] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. *arXiv preprint arXiv:1412.0233*, 2014.

[8] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 2933–2941, 2014.

[9] Justin Domke. Generic methods for optimization-based modeling. In *International Conference on Artificial Intelligence and Statistics*, pages 318–326, 2012.

[10] Chuan-sheng Foo, Chuong B Do, and Andrew Y Ng. Efficient multiple hyperparameter learning for log-linear models. In *Advances in neural information processing systems*, pages 377–384, 2008.

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

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[13] Ian J Goodfellow and Oriol Vinyals. Qualitatively characterizing neural network optimization problems. *International Conference on Learning Representations*, 2014.

[14] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[16] Geoffrey E Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS 2014 Deep Learning Workshop*, 2014.

[17] Jelena Luketina, Mathias Berglund, and Tapani Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. *arXiv preprint arXiv:1511.06727*, 2015.

[18] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv preprint arXiv:1502.03492*, 2015.

[19] Radford M Neal. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2, 2011.

[20] Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang W Koh, Quoc V Le, and Andrew Y Ng. Tiled convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1279–1287, 2010.

[21] Nicolas Pinto, David Doukhan, James J DiCarlo, and David D Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS computational biology*, 5(11):e1000579, 2009.

[22] Tom Schaul and Yann LeCun. Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. *arXiv preprint arXiv:1301.3764*, 2013.

[23] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.

[24] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[25] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[26] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Patwary, Mostofa Ali, Ryan P Adams, et al. Scalable bayesian optimization using deep neural networks. *arXiv preprint arXiv:1502.05700*, 2015.

[27] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[28] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in Neural Information Processing Systems*, pages 2004–2012, 2013.

[29] Andrea Walther. Automatic differentiation of explicit runge-kutta methods for optimal control. *Computational Optimization and Applications*, 36(1):83–108, 2007.

[30] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, and Nando de Freitas. Bayesian optimization in a billion dimensions via random embeddings. *arXiv preprint arXiv:1301.1942*, 2013.