Experiments With Scalable Gradient-based Hyperparameter Optimization for Deep Neural Networks

by

Michael St. Jules

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Stephen Vavasis

Waterloo, Ontario, Canada, 2017

© Michael St. Jules 2017

I hereby declare that I am the sole author of this report. This is a true copy of the report, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

Gradient-based hyperparameter optimization algorithms have the potential to scale to numbers of individual hyperparameters proportional to the number of elementary parameters, unlike other current approaches. Some candidate completions of DrMAD, one such algorithm that updates the hyperparameters after fully training the parameters of the model, are explored, with experiments tuning per-parameter L2 regularization coefficients on CIFAR10 with the DenseNet architecture. Experiments with DenseNets on CIFAR10 are also conducted with an adaptive method, which updates the hyperparameters during the training of elementary parameters, tuning per-parameter learning rates and L2 regularization. The experiments do not establish the utility of either method, but the adaptive method shows some promise, with further experiments required.

Table of Contents

List of Figures v

1 Introduction 1

1.1 Overview . . . 2

1.2 Background and Related Work . . . 2

1.2.1 Neural Networks and Image Classification Tasks . . . 2

1.2.2 Hyperparameter Optimization . . . 6

2 Algorithm Details 14

2.1 Elementary Parameter Paths . . . 14

2.2 Hyperparameter Sharing . . . 15

2.3 Symmetry . . . 16

2.4 Learning Algorithms: SGD and Variants . . . 17

3 Experiments 20

3.1 Experiments on CPU . . . 20

3.2 Experiments on GPU . . . 27

3.2.1 DrMAD . . . 30

3.2.2 Adaptive Method . . . 35

4 Discussion and Directions for Future Work 41

References 42

List of Figures

1.1 Computational graph for computing the hypergradients in [48] 11

2.1 Parameter paths projected onto two random coordinates for a small CNN. The green dot is at the initial value, and the red one is at the final value. 15

2.2 Paths as functions of the iteration, for a random coordinate each, for a small CNN. The green dot is at the initial value, and the red one is at the final value. . 16

2.3 Distance of the current parameters in the second meta-iteration from the final parameters in the first meta-iteration, for 5 epochs each. The parameters are reset with the same initial values for the second run, but the seed for dropout is not the same. 18

3.1 Norms of hypergradients following the DrMAD hypergradient signs 22

3.2 Cosines of the angles between DrMAD and exact hypergradients and the between the vectors of their signs, calculated as $\frac{d\boldsymbol{\theta}_{DrMAD} \cdot d\boldsymbol{\theta}_{exact}}{\|d\boldsymbol{\theta}_{DrMAD}\| \|d\boldsymbol{\theta}_{exact}\|}$, following the DrMAD hypergradient signs 23

3.3 Error rates following the DrMAD hypergradient signs 24

3.4 Norms of the accumulated hypergradients for 20,000 elementary iterations, which are accumulated starting with value 0 at $t = T - 1 = 20,000 - 1$ on the right and following the reverse iterations with decreasing t towards the left 25

3.5 Norms of the accumulated hypergradients for 200,000 elementary iterations, which are accumulated starting with value 0 at $t = T - 1 = 200,000 - 1$ on the right and following the reverse iterations with decreasing t towards the left. . . . 26

3.6 Comparisons of exact reverse path values with and without exact arithmetic on meta-iteration 11 (but updating with the signs of DrMAD hypergradients) for 2000 elementary iterations. 28

3.7 DenseNet benchmark errors. The final test error was **11.4%**. 30
3.8 Errors per meta-iteration with DrMAD with initial L2 decay 10^{-4}. 31
3.9 Errors during training with DrMAD with initial L2 decay 10^{-4}. 31
3.10 Per-layer average log L2 decay hyperparameters with DrMAD, starting at -4. . . 32
3.11 Errors during training with DrMAD with initial L2 decay 10^{-2}. 32
3.12 Per-layer average log L2 decay hyperparameters with DrMAD, starting at -2. . . 33
3.13 Errors during training with DrMAD with initial L2 decay 10^{-7}. 33
3.14 Per-layer average log L2 decay hyperparameters with DrMAD, starting at -7. . . 34
3.15 Errors with the adaptive method, first experiment 36
3.16 Per-layer average log L2 decay hyperparameters with the adaptive method, first experiment 36
3.17 Per-layer average log learning rates with the adaptive method, first experiment . . 37
3.18 Errors with the adaptive method, second experiment 38
3.19 Per-layer average log L2 decay hyperparameters with the adaptive method, second experiment 39
3.20 Per-layer average log learning rates with the adaptive method, second experiment 40

Chapter 1

Introduction

Machine learning is a field in the intersection of computer science and statistics with the broad goal of *learning* from data, through algorithms, where 'learning' here means fitting a particular model to the data, for the purpose of e.g. classifying or predicting the value of some function of the data, representing the data or generating more examples from the distribution from which the data came (or, more precisely, from the distribution fitted to the data).

In deep learning, a subfield of machine learning, artificial neural networks composed of several layers of affine functions and componentwise nonlinearities with hundreds, thousands or even millions of parameters to be learned, are used for such tasks. The objective function to be optimized is continuous and usually differentiable almost everywhere, but never globally convex. Deep learning has achieved considerable success in several tasks, and this project focuses on image classification in particular.

Unfortunately, the model or even the training algorithm often makes use of *hyperparameters*, i.e. parameters that aren't themselves learned during training. These include both discrete and continuous hyperparameters. Examples of discrete hyperparameters are the training algorithm itself, the data preprocessing, the number of training iterations or stopping criteria as well as details of the architecture of the neural network including the types and number of layers, the types of nonlinearities, etc.. Continuous hyperparameters are typically the learning rates and momentum decays as well as coefficients of norm penalties on the weights in the training objective (usually L2) which help regularize the model and improve generalization. All of these hyperparameters together can be cumbersome to tune by hand based on the validation error, indicating a need for *hyperparameter optimization* methods.

1.1 Overview

In section 1.2, and subsection 1.2.1 specifically, the necessary background on neural networks, deep learning and image classification is covered. In subsection 1.2.2, different approaches to hyperparameter optimization are discussed, with focus on gradient-based approaches in subsection 1.2.2, on which this project builds.

In chapter 2, some preliminary experiments and issues with the the approaches followed in this project are explored. Specifically, in section 2.1, the paths taken by elementary parameters during training are explored, demonstrating obstacles to improvements of the DrMAD [22] reverse path approximation. In section 2.2, hyperparameter sharing is discussed. In section 2.3, issues of symmetry in the weights of a neural networks and ensuring symmetry breaks consistently between hyperparameter updates are explained. In section 2.4, the choices of learning algorithms in this project, for both elementary parameters and hyperparameters, are justified.

In sections 3.1 and 3.2 of chapter 3, the results of experiments on CPU and GPU are presented and discussed, with focus on DrMAD [22] and exact hypergradients [48] in the CPU experiments, and DrMAD and an adaptive method in the GPU experiments.

Finally, the project is concluded and directions for future work are mentioned in 4.

1.2 Background and Related Work

1.2.1 Neural Networks and Image Classification Tasks

Feedforward neural networks, or more specifically, *multilayer perceptrons*, are machine learning models, which as functions of their input, are the alternating composition of affine transformations, i.e. $\mathbf{x} \in \mathbb{R}^{d_1} \mapsto \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^{d_2}$ where $\mathbf{W}$ is a $d_2 \times d_1$ matrix and $\mathbf{b} \in \mathbb{R}^{d_2}$, with nonlinear functions, called *activation functions*, applied to each *component* or *coordinate* or *unit*. $\mathbf{W}$ and $\mathbf{b}$ are usually parameters of the model to be 'learned' or estimated, while typical activation functions are:

- the *sigmoid function* $\sigma : \mathbb{R} \to (0, 1)$ defined by $\sigma(x) = \frac{1}{1+e^{-x}}$,
- the *hyperbolic tangent function* $\tanh : \mathbb{R} \to (-1, 1)$ defined by $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$, or
- the *rectified linear unit* $\mathrm{ReLU} : \mathbb{R} \to \mathbb{R}^+$ defined by $\mathrm{ReLU}(x) = \max\{0, x\}$.

The weights $\mathbf{W}$ and biases $\mathbf{b}$ in all of the layers to be trained are together denoted by $\mathbf{w}$ and called the *parameters* or *elementary parameters* (as in [48]). $\boldsymbol{\theta}$ denotes the *hyperparameters*, parameters whose values are fixed during training.

In classification with C classes, often denoted $0, 1, \ldots, C-1$, on input $\mathbf{x}$, the model must output a vector in $\mathbb{R}^C$, with each component positive and together summing to 1 (i.e. strictly inside the interior of the C-simplex). This is usually enforced by applying the *softmax function* to the C components at the end of the neural network, so that for $\mathbf{z} = (z_j)_j \in \mathbb{R}^C$, and $k = 0, 1, \ldots, C-1$, $\sigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$. The *training loss* function $L(\mathbf{w}, \boldsymbol{\theta})$ is then typically the *cross-entropy*, a smooth approximation of the classification error, plus some *regularization* function $R(\mathbf{w}, \boldsymbol{\theta})$ with *hyperparameters* $\boldsymbol{\theta}$:

$$L(\mathbf{w}, \boldsymbol{\theta}) = -\tfrac{1}{N} \sum_{i=1}^{N} \sum_{c=0}^{C-1} y_{ic} \log(\hat{y}_c(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta})) + R(\mathbf{w}, \boldsymbol{\theta}),$$

where $y_{ic} = 1$ if $\mathbf{x}_i$ is in class c and 0, otherwise; and $\hat{\mathbf{y}}(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta}) = (\hat{y}_c(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta}))_{c=0}^{C-1} \in \mathbb{R}^C$ is the output of the network on input data $\mathbf{x}_i$ with parameters $\mathbf{w}$ and hyperparameters $\boldsymbol{\theta}$, so that $\hat{y}_c(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta})$ represents the predicted probability that $\mathbf{x}_i$ belongs to class c.

A typical choice for $R(\mathbf{w}, \boldsymbol{\theta})$ is *L2 weight decay* or *L2 regularization* or an *L2 penalty* $R(\mathbf{w}, \boldsymbol{\theta}) = \lambda \|\mathbf{w}\|^2$, for $\lambda > 0$ a component of $\boldsymbol{\theta}$. More generally, each component w_j of $\mathbf{w} = (w_j)_j$ may receive its own regularization hyperparameter, i.e. $R(\mathbf{w}, \boldsymbol{\theta}) = \sum_j \lambda_j w_j^2$, where the λ_j are again components of $\boldsymbol{\theta}$. The hyperparameters $\boldsymbol{\theta}$ are fixed, and then the (elementary) parameters $\mathbf{w}$ are trained on the dataset.

In this project, the datasets used are

- MNIST [41], consisting of 60,000 training and 10,000 test 32×32 grayscale images of handwritten digits, 0 to 9, so 10 classes, although only subsets of size 20,000 and 5,000 of the training and test subsets were used; and
- CIFAR10 [37], consisting of 50,000 training and 10,000 test 32×32 RGB images with 10 classes.

The preferred algorithm for learning the parameters is (mini-batch) *stochastic gradient descent* (SGD) [58] with momentum or Nesterov momentum [53]; several adaptive methods have been developed and are popular, and while they may reduce the training loss more quickly, their test errors are often worse. [76] SGD involves replacing at each step $L(\mathbf{w}, \boldsymbol{\theta})$ with

$$L(\mathbf{w}, \boldsymbol{\theta}, t) = -\tfrac{1}{|B_t|} \sum_{i:\mathbf{x}_i \in B_t} \sum_{c=0}^{C-1} y_{ic} \log(\hat{y}_c(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta})) + R(\mathbf{w}, \boldsymbol{\theta}),$$

where B_t is a small random sample (often of size 32, 64 or 128) from the dataset, called a *batch* or *mini-batch*, and used in step t. The purpose is to satisfy, for each t,

$$\mathbb{E}[L(\mathbf{w}, \boldsymbol{\theta}, t)] = L(\mathbf{w}, \boldsymbol{\theta}) \,,$$

and

$$\mathbb{E}[\nabla_{\mathbf{w}} L(\mathbf{w}, \boldsymbol{\theta}, t)] = \nabla_{\mathbf{w}} L(\mathbf{w}, \boldsymbol{\theta}) \,.$$

In practice, however, the dataset is shuffled randomly once before training, split into batches of a fixed size (the *batch size*), and these batches are used one at a time, until all are used. This is then repeated in the same order starting again from the first batch. An *epoch* then consists of consecutive iterations that cover all of the training data, starting from the first batch and ending with the last.

SGD starts with parameters initialized at $\mathbf{w}_0$ and velocity $\mathbf{v}_{-1} = \mathbf{0}$, and ends with velocity $\mathbf{v}_{T-1}$ and parameters $\mathbf{w}_T = \mathbf{w}_{T-1} + \alpha_{T-1}\mathbf{v}_{T-1}$. See Algorithm 1 below. The α_t are called the *learning rates* or *step sizes*, and the γ_t are the *momentum decay rates* or just *momentum decays* or *momentum*.

Algorithm 1 Stochastic gradient descent with momentum (in the notation of [48], with some modifications)

1: **input:** initial $\mathbf{w}_0$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
2: initialize $\mathbf{v}_{-1} = \mathbf{0}$
3: **for** $t = 0$ **to** $T - 1$ **do**
4: $\quad \mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
5: $\quad \mathbf{v}_t = \gamma_t \mathbf{v}_{t-1} - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
6: $\quad \mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
7: **end for**
8: **output** trained parameters $\mathbf{w}_T$

SGD with Nesterov momentum is instead defined by the following step [7]

$$\begin{aligned}\mathbf{v}_{t+1} &= \gamma_t \mathbf{v}_t - \alpha_t \nabla_{\mathbf{w}} L(\mathbf{w}_t + \gamma_t \mathbf{v}_t, \boldsymbol{\theta}, t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_{t+1} + \mathbf{v}_t\end{aligned}$$

but is reformulated in [7] as

$$\begin{aligned}\mathbf{v}_{t+1} &= \gamma_t \mathbf{v}_t - \alpha_t \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_{t+1} - \gamma_t \mathbf{v}_t + (1 + \gamma_{t+1})\mathbf{v}_{t+1}\end{aligned}$$

Note that the learning rate is incorporated into the velocities here, but not in regular SGD with momentum.

CNNs, ResNet and DenseNet

Convolutional neural networks (CNNs) [40] have made neural networks competitive on image tasks, with [38] achieving a breakthrough on ImageNet [60], an image classification task with millions of images and over ten thousand classes. What characterizes the CNN is the convolutional layer, which uses considerable amounts of weight sharing. The input to a 2D convolutional layer for 2D images has 3 dimensions, 2 for the 2 dimensions of the image, and the third dimension for the number of *input channels* or *input features*. For example, if the first layer of the neural network is a convolution, then the input channels are often the 3 RGB values at each pixel; otherwise the inputs are simply outputs of previous layers. The weights in a convolutional layer are formed by *kernels* or *filters*, 3D arrays with depth equal to the number of input features into the layer, and width and height usually set to some small constant, typically 3. To compute the resulting feature map from this kernel and the input to the layer, the dot product is taken between the kernel and the inputs at each spatial location, using the full depth. That is, letting $\mathbf{W} = (\mathbf{w}_{ij})_{i,j=-1}^{1}$ be a 3×3 kernel of the same depth of the input $\mathbf{Z} = (\mathbf{z}_{ij})_{i=0,j=0}^{M-1,N-1}$ of spatial dimensions $M \times N$. Then the output with this kernel at location $i, j, 1 \leq i \leq M-2, 1 \leq j \leq N-2$ is the *convolution*

$$\sum_{i',j'=-1}^{1} \mathbf{w}_{i'j'}^{\top} \mathbf{z}_{i+i',j+j'} \ .$$

In other words, this is the dot product of $\mathbf{W}$ with the block of the same dimensions centered at i, j, with the full depth. The coordinates $i + i', j + j', i', j' = -1, 0, 1$ together form the *receptive field* for the output at coordinates i, j. Just $9 = 3 \times 3$ weights are used to produce $(M - 3 + 1) \times (N - 3 + 1)$ outputs. This describes the situation with the *stride* equal to 1, but with a greater stride s, only at each $(s - 1)$th position will the kernel be convolved with the corresponding block.

These outputs together form one *feature map*, and each kernel in a convolutional layer will produce a separate feature map, which will all be stacked depth-wise.

Other common layers in CNNs include:

- Pooling layers, which are like convolutional layers, but rather than computing convolutions and summing over the depth, the values are computed separately for each feature map (or depth value), and the computed value is typically the maximum (*maxpooling*) or the average in the receptive field at that depth. Furthermore, the size of the pool is typically 2×2, and stride 2 is used, so this means dividing each input feature map into non-overlapping 2×2 squares, each from which a single value is computed.

- Batch normalization layers [33], which normalize the units for each batch during training, subtracting from the value of each unit its batch mean and then dividing by its batch standard deviation; this is typically followed by rescaling and shifting with trainable parameters. For prediction, the mean and variance of the units over the full training set is used.

- Dropout layers [29, 70], which set the value of each input unit to 0 independently with some probability p, for regularization, and dividing by $1 - p$ to preserve the expected value. This is disabled for prediction.

Note that all of these layers just defined are shift invariant (excluding the edges for convolutions and pooling, and when the shift is a multiple of the stride).

DenseNets [30] established the state-of-the-art performance on the image classification tasks SVHN [54], CIFAR10 and CIFAR100 [37], and better scalable performance on ImageNet [60], beating deep *residual networks* (ResNet) and some variants thereof [28, 79]. Residual networks are a modification of CNNs with *skip connections*, writing the output $\mathbf{y}'$ of a layer or block as the sum of the output of a block of transformations H on its input $\mathbf{y}$ and the input itself, to allow information from previous layers to pass through more easily:

$$\mathbf{y}' = H(\mathbf{y}; \mathbf{w}, \boldsymbol{\theta}) + \mathbf{y} .$$

H_l typically consists of convolutions, batch normalization and ReLU, but also sometimes dropout. $\mathbf{y}$ is padded with 0s on the edges before convolutions to preserve the dimensionality. H does not depend directly on all of the parameters in $\mathbf{w}$ or hyperparameters in $\boldsymbol{\theta}$.

Densely connected neural networks, or *DenseNets*, are similar, but rather than taking a sum, the outputs of all (or some consecutive) previous blocks, $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{l-1}$ are concatenated and accumulated, and then the transformations H_l are applied:

$$\mathbf{y}_l = H_l([\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{l-1}]; \mathbf{w}, \boldsymbol{\theta}) .$$

This allows for more efficient use of parameters.

Because DenseNets have established state-of-the-art performance, experiments in this project are conducted with them, but on smaller networks than the most competitive.

1.2.2 Hyperparameter Optimization

To "tune" the hyperparameters $\boldsymbol{\theta}$, the dataset is split before training into *training set* and *validation set*, and then the elementary parameters are trained using the data from the training set. $\boldsymbol{\theta}$

is then selected as to minimize the *validation error* or some other *validation loss* with the final weights after training with given fixed values of $\boldsymbol{\theta}$. In order to measure the performance of the final network in an unbiased way, a third set of data is split off from the rest at the beginning, too, and the elementary parameters $\mathbf{w}$ and hyperparameters $\boldsymbol{\theta}$ (and any other values in the network, like batch normalization means and standard deviations) are never modified in response to this data.

The error is the percentage of incorrectl classed data, where the class of $\mathbf{x}_i$ is chosen as the class corresponding to the (or a) maximum component of the output of the neural network on $\mathbf{x}_i$, $\hat{\mathbf{y}}(\mathbf{x}_i; \mathbf{w}) = (\hat{y}_c(\mathbf{x}_i; \mathbf{w}))_{c=0}^{C-1}$, so the validation error is simply the error on the validation set, and similarly for the training error. In this project, the validation loss is also the cross-entropy, but on the validation set and without the regularization term $R(\mathbf{w}, \boldsymbol{\theta})$.

Several approaches to hyperparameter optimization have been developed. One of the simplest is grid search, which is an exhaustive search on a grid of values. More efficient is random search [8], because the *effective dimension* of the problem may be significantly lower than the number of hyperparameters, so time won't be wasted checking all values for a hyperparameter of little importance.

A large class of algorithms often used for hyperparameter optimization are *sequential model-based global optimization* algorithms and more narrowly, *Bayesian optimization* algorithms. See [9, 10, 31, 66, 67] for reviews and popular variants. These involve modelling the validation loss as a function of hyperparameters and choosing next points based on measures like the *expected improvement*, taking the full history of hyperparameter configurations into account, but as such, normally, each hyperparameter iteration would require at least $N \times D$ operations and the storing of this much data, where N is the number of hyperparameter configurations tried so far, and D is the number of hyperparameters, just to be able to look at them all. The time complexity is not inherently concerning if N doesn't grow too large and everything fits on GPU, and even GPU transfers may be cheap compared to the training of the neural network, but in practice, most of these methods don't scale well to large numbers of hyperparameters, since the number of evaluations required to cover the hyperparameter space grows exponentially in the number of hyperparameters, *the curse of dimensionality*. [66] Some specific attempts have been made in this regard. Random embeddings [75] have been proposed for scaling Bayesian optimization to billions of dimensions, but where the hyperparameters have low *effective dimensionality*, i.e. the hyperparameters can be reduced to a small set that determines them all. Some papers have mentioned the possibility of using hyperparameter gradient information for Bayesian optimization [48, 77, 78], but as of writing, it's not clear anyone has actually attempted this, as obtaining sufficiently accurate hyperparameter gradient information remains an open problem, one which this project explores.

Recently, Hyperband [43, 59], evolutionary algorithms [52], particle swarms [45, 46] and reinforcement learning [3, 80] algorithms have been developed for hyperparameter optimization. These all require the training of multiple networks (preferably in parallel) before deciding on the next candidate hyperparameters to train. In [43, 52, 59, 80], the networks aren't trained until convergence at each round, which means the "winning" hyperparameters are those that reach the lowest loss within the allotted number of iterations, so that more quickly trained networks become preferred. Hyperband is a refinement of the SuccessiveHalving algorithm, "uniformly allocate a budget to a set of hyperparameter configurations, evaluate the performance of all configurations, throw out the worst half, and repeat until one configurations remains," that allocates resources more efficiently and explores more hyperparameters of the SuccessiveHalving algorithm itself. [43]

Gradient-based Approaches

What sets gradient-based approaches apart from those above are the potential to scale to huge number of effective (but continuous) dimensions, in comparison to Bayesian optimization, while an individual setting of the hyperparameters informs where to search next through the hypergradient, unlike, in evolutionary, particle swarm and reinforcement learning algorithms, as well as Hyperband, which require the performances of the network with several different hyperparameters to be compared to choose the next ones.

In these approaches, the final parameters $\mathbf{w}$ after training with hyperparameters $\boldsymbol{\theta}$ implicitly depend on $\boldsymbol{\theta}$ (and other values, like learning rate, momentum decays, randomness, etc.). Writing $\mathbf{w}(\boldsymbol{\theta})$, the validation loss, say $f(\mathbf{w})$, then also depends on the hyperparameters this way, and the *hypergradient* $\nabla_{\boldsymbol{\theta}} f(\mathbf{w}(\boldsymbol{\theta}))$ is computed or estimated. $\boldsymbol{\theta}$ is then updated using this gradient, through some variant of gradient descent. A *meta-iteration* refers to the step from the previous value of $\boldsymbol{\theta}$ to its next after an update, with all of the elementary training in it.

Several methods use *implicit differentiation*, based on the assumption that the final elementary parameters are approximate critical points, starting with [6, 39]. These require the inversion of the Hessian of the training loss (or some approximation thereof, like the diagonals or the Moore-Penrose pseudoinverse). The more recent [56] *approximately* inverts the Hessian on some input using the iterative conjugate-gradient method, which, although feasible through repeated Hessian-vector products [55] with access to the exact full gradient of the training loss, has not yet been extended to the stochastic setting. In particular, inversion is not a linear operation, so the expected value of the inverse of the stochastic estimate of the Hessian (multiplied by a vector) may be biased away from the inverse of the Hessian (multiplied by a vector). In the paper, the author also proves convergence under certain assumptions, including, of course, invertibility

of the Hessian of the training loss at the optimal values. Note however, in practice, the Hessian at the end of training is often degenerate, with negative eigenvalues and eigenvalues close to 0 [13, 61, 62, 63]; this may be an important obstacle for these implicit differentiation-based algorithms generally.

Iterative differentiation, developed in [16], entails differentiating the validation loss with respect to the hyperparameters by using the chain rule through all of the steps of gradient descent. [48] extended this to the stochastic setting and avoids storing the whole trajectory or even checkpoints (from which steps can be recomputed) by reversing the SGD with momentum path exactly, drastically improving the memory efficiency. This project continues along these lines.

Below in Algorithm 2, and in the notation of 1, $d\mathbf{v}_t = \dfrac{\partial f(\mathbf{w}_T)}{\partial \mathbf{v}_t}, d\mathbf{w}_t = \dfrac{\partial f(\mathbf{w}_T)}{\partial \mathbf{w}_t}$, where $\mathbf{w}_T$ the final parameters, is a function of $\mathbf{v}_t, \mathbf{w}_t, \boldsymbol{\theta}$ and so on.

Algorithm 2 Reverse-mode differentiation of SGD [48]

1: **input:** $\mathbf{w}_T$, $\mathbf{v}_{T-1}$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
2: initialize $d\mathbf{v}_T = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = 0$, $d\gamma_t = 0$
3: initialize $d\mathbf{w}_T = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
4: **for** $t = T - 1$ **counting down to** 0 **do**
5: $\quad d\alpha_t = d\mathbf{w}_{t+1}^{\mathsf{T}} \mathbf{v}_t$
6: $\quad \mathbf{w}_t = \mathbf{w}_{t+1} - \alpha_t \mathbf{v}_t$
7: $\quad \mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
8: $\quad \mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$
(lines 6–8: exactly reverse gradient descent operations)
9: $\quad d\mathbf{v}_t = \gamma_{t+1} d\mathbf{v}_{t+1} + \alpha_t d\mathbf{w}_{t+1}$
10: $\quad d\gamma_t = d\mathbf{v}_t^{\mathsf{T}}(\mathbf{v}_{t-1} + \mathbf{g}_t)$
11: $\quad d\mathbf{w}_t = d\mathbf{w}_{t+1} - (1 - \gamma_t) d\mathbf{v}_t \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
12: $\quad d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v}_t \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
13: **end for**
14: **output** gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_0$, $\mathbf{v}_{-1}$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

Note the γ_{t+1} in line 9, but γ_t in the other lines. The $1 - \gamma_t$ is for the dependence of the loss function through $\mathbf{g}_t$, as $\dfrac{\partial \mathbf{v}_t}{\partial \mathbf{g}_t} = -(1 - \gamma_t)$. Line 9 is as it appears above because the loss function depends on $\mathbf{v}_t$ only through paths through $\mathbf{w}_{t+1}$ and $\mathbf{v}_{t+1}$, so

$$
\begin{aligned}
d\mathbf{v}_t &= \frac{\partial f(\mathbf{w}_T)}{\partial \mathbf{v}_t} && (1.1)\\
&= \frac{\partial f(\mathbf{w}_T)}{\partial \mathbf{v}_{t+1}}\frac{\partial \mathbf{v}_{t+1}}{\partial \mathbf{v}_t} + \frac{\partial f(\mathbf{w}_T)}{\partial \mathbf{w}_{t+1}}\frac{\partial \mathbf{w}_{t+1}}{\partial \mathbf{v}_t} && (1.2)\\
&= d\mathbf{v}_{t+1}\gamma_{t+1} + d\mathbf{w}_{t+1}\alpha_t && (1.3)\\
&= \gamma_{t+1} d\mathbf{v}_{t+1} + \alpha_t d\mathbf{w}_{t+1} && (1.4)
\end{aligned}
$$

In [48], the authors avoid referring to two different values of γ in a single reverse iteration by using $d\mathbf{v}_t = d\mathbf{v}_t^0 + \alpha_t d\mathbf{w}_{t+1}$ in place of $d\mathbf{v}_t = \gamma_{t+1} d\mathbf{v}_{t+1} + \alpha_t d\mathbf{w}_{t+1}$, and having $d\mathbf{v}_{t-1}^0 = \gamma_t d\mathbf{v}_{t+1}$ at the end of the iteration.

The Hessian-vector products in lines 11 and 12 can be computed with reverse-mode automatic differentiation (backpropagation) with at most a constant factor more memory and time than a gradient step using the L-operator, as a simple consequence of the linearity of differentiation. [55] That is, for $\mathbf{x}, \mathbf{v} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$,

$$
\mathbf{v}^\mathsf{T} \nabla_{\mathbf{y}} \nabla_{\mathbf{x}} L(\mathbf{x}, \mathbf{y}) = \nabla_{\mathbf{y}} \Big(\mathbf{v}^\mathsf{T} \nabla_{\mathbf{x}} L(\mathbf{x}, \mathbf{y}) \Big),
$$

and, the special case $\mathbf{x} = \mathbf{y} \in \mathbb{R}^n$ gives

$$
\mathbf{v}^\mathsf{T} \nabla_{\mathbf{x}} \nabla_{\mathbf{x}} L(\mathbf{x}) = \nabla_{\mathbf{x}} \Big(\mathbf{v}^\mathsf{T} \nabla_{\mathbf{x}} L(\mathbf{x}) \Big).
$$

The velocity in the algorithm is used to reverse the path; plain SGD without momentum (e.g. $\gamma_t = 0$ for all t) would not allow this, because from $g_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ and α_t alone, $\mathbf{w}_{t-1}$ cannot be recovered. A major concern raised in [48] about naively reversing the paths with finite precision arithmetic on a computer is the loss of information about $\mathbf{v}_{t-1}$ when multiplied by γ_t to obtain $\mathbf{v}_t$ during each elementary iteration. In particular, $-\log_2(\gamma_t)$ bits of $\mathbf{v}_{t-1}$ are lost on average directly because of this multiplication; in [48], these lost bits are stored, so that previous iterations may be recovered exactly. Unfortunately, these extra bits accumulate without bound as the number of elementary iterations increases, making their algorithm unsuitable for use on GPUs. This motivates [22]:

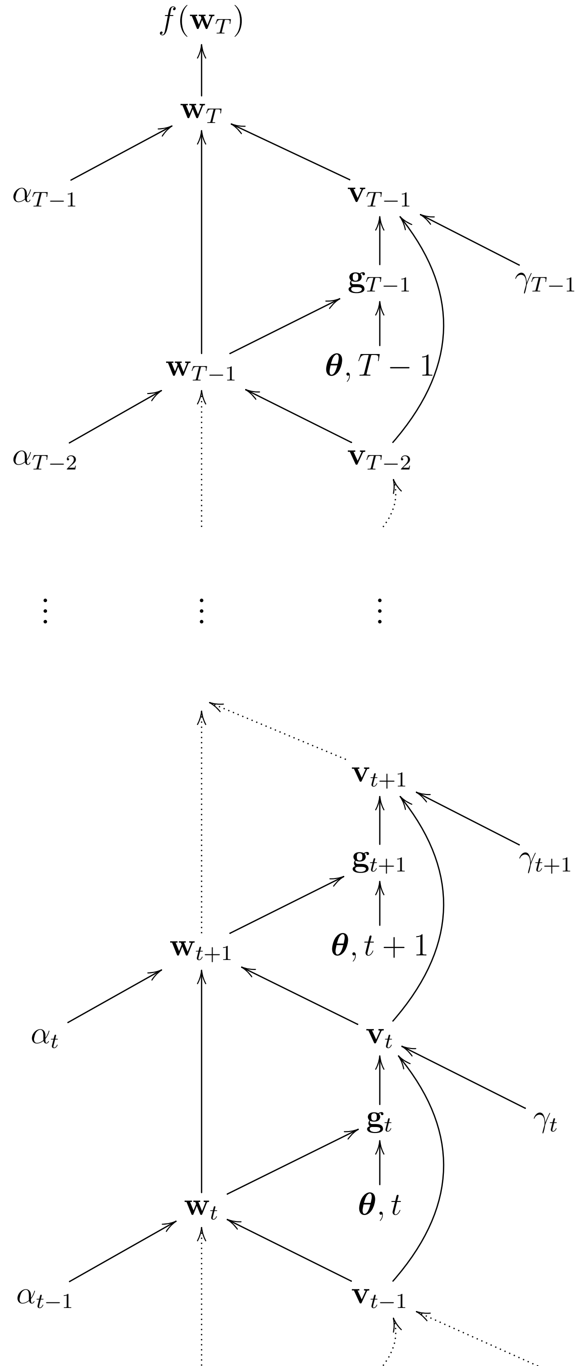


Figure 1.1: Computational graph for computing the hypergradients in [48]

Algorithm 3 DrMAD [22]

1: **input:** $\mathbf{w}_T$, $\mathbf{v}_{T-1}$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
2: initialize $d\mathbf{v}_T = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$
3: initialize $d\mathbf{w}_T = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
4: **for** $t = T - 1$ **counting down to** 0 **do**
5: $\quad \mathbf{w}_t = \frac{t}{T}\mathbf{w}_T + (1 - \frac{t}{T})\mathbf{w}_0$ ▷ approximately reverse path
6: $\quad d\mathbf{v}_t = \gamma_{t+1} d\mathbf{v}_{t+1} + \alpha_t d\mathbf{w}_{t+1}$
7: $\quad d\mathbf{w}_t = d\mathbf{w}_{t+1} - (1 - \gamma_t) d\mathbf{v}_t \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
8: $\quad d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v}_t \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
9: **end for**
10: **output** gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_0$ and $\boldsymbol{\theta}$

The reverse path in Algorithm 3 which is spaced uniformly on the straight line between $\mathbf{w}_0$ and $\mathbf{w}_T$, was inspired in DrMAD [22] by [27], in which the authors observed that the loss function generally decreased monotonically along this path. Of course, this is a crude approximation, which ignores even the general fact that the distances between successive steps tend to decrease with each step. This has important consequences for the practical use of DrMAD.

One further advantage of DrMAD suggested in [22], beyond being suitable for use on GPU, is that the path computation does not depend on precisely which learning algorithm is used, e.g. SGD, SGD with momentum, SGD with Nesterov momentum [53], adaptive methods like the popular AdaGrad [18], RMSProp [74] and Adam [35]. The exact algorithm of [48], on the other hand, would need to be rederived for each such method. One reason why DrMAD may not need to be readapted is that if the final parameters $\mathbf{w}$ obtained by one training algorithm are also obtainable using SGD with momentum (SGDm), then the exact path followed doesn't matter, and it can be treated as if the SGDm path was followed instead. Similarly, rather than approximating the SGD+momentum path, it may suffice to come up with a plausible SGDm path, e.g. a path that is close to one that could have been taken by SGDm with different randomness. It's unlikely the DrMAD path itself is one such path, however, as section 2.1 suggests.

Adaptive hypergradient methods have also been developed in the same spirit as [48], but which update the hyperparameters during the training of the elementary parameters and effectively only use a single reverse iteration per elementary iteration (or for several elementary iterations), so reversing the velocity is not necessary. These are [47], which tunes regularization hyperparameters, specifically L2 regularization and Gaussian noise added to the units, and [1, 2, 5], which tune learning rates, albeit based on the training loss rather than the validation loss, but this is a simple modification acknowledged in [5]. The adaptive learning rate methods, with hypergradients of the *validation loss*, on first consideration seem promising, because the popular

adaptive methods AdaGrad, RMSProp and Adam have been shown to lead to worse generalization performance than plain SGD with momentum; [76] and hypergradient methods deal directly with the validation loss, a measure of generalization. In [22], it is claimed that [47] does not scale well to large numbers of hyperparameters, but experiments in this project, while not conclusive, seem to suggest otherwise.

Recently, [20, 21] derived general forward and reverse mode iterative differentiation methods for arbitrary learning algorithms whose iterations mapping $\mathbf{w}_t \to \mathbf{w}_{t+1}$ (and other variables, like the velocity) are differentiable. However, their reverse mode algorithm, essentially a generalization of [16, 48] involved storing all intermediate steps along the path, which is not feasible on GPU (nor would be checkpointing and recomputing), while their forward mode algorithm scales as $O(Tmd)$ in time and $O(md)$ in space, where T is the number of elementary iterations, m is the number of elementary parameters and d is the number of hyperparameters. This is because the algorithm stores and updates the matrix $\frac{d\mathbf{w}_t}{d\boldsymbol{\theta}}$, or more generally, $\frac{d\mathbf{s}_t}{d\boldsymbol{\theta}}$, where $\mathbf{s}_t$ contains $\mathbf{w}_t, \mathbf{v}_t$ and any other variables used and updated at iteration t. That is, writing the elementary update step as $\mathbf{s}_{t+1} = \Phi_t(\mathbf{s}_t, \boldsymbol{\theta})$, the chain rule gives

$$\frac{d\mathbf{s}_{t+1}}{d\boldsymbol{\theta}} = \frac{\partial \Phi_t(\mathbf{s}_t,\boldsymbol{\theta})}{\partial \mathbf{s}_t}\frac{d\mathbf{s}_t}{d\boldsymbol{\theta}} + \frac{\partial \Phi_t(\mathbf{s}_t,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\ ,$$

and the hypergradient is $\nabla_{\mathbf{w}} f(\mathbf{w}_{t+1})\frac{d\mathbf{w}_{t+1}}{d\boldsymbol{\theta}}$ (which may be computed only at the end).

This may be acceptable for a very small number of hyperparameters, because otherwise the equivalent of the entire neural network once for each hyperparameter needs to be stored. Interestingly, with the forward mode version, the hypergradient can be computed at each elementary iteration during the forward pass, treating the current parameters as the final parameters although it's not clear what specific advantage this would have, because the hypergradient is with respect to the entire training trajectory from intialization, so it seems the elementary parameters should be reinitialized after updating the hyperparameters. [51] also updates the learning rates in an online way by approximating the hypergradient for the whole trajectory. A priori, the adaptive methods above, which only use information from the last step, seem better suited for this.

Chapter 2

Algorithm Details

In this chapter, some preliminary experiments and issues with the the approaches followed in this project are explored. Specifically, in section 2.1, the paths taken by elementary parameters during training are explored, demonstrating obstacles to improvements of the DrMAD [22] reverse path approximation. In section 2.2, hyperparameter sharing is discussed. In section 2.3, issues of symmetry in the weights of a neural networks and ensuring symmetry breaks consistently between hyperparameter updates are explained. In section 2.4, the choices of learning algorithms in this project, for both elementary parameters and hyperparameters, are justified.

2.1 Elementary Parameter Paths

In [27], the authors experimentally demonstrated that for neural networks on benchmark datasets with modern architecture, the loss function decreased monotonically on the straight line from the initial parameters to the final parameters. [44] visualized the projections of the paths taken by the weights of a CNN on MNIST [41] onto their first two principal components, demonstrating that the paths taken are far from straight. This was already observed for multilayer perceptrons in [24, 25]. Furthermore, while the paths projected onto their first principal components appear smooth, projecting onto random coordinates/components reveals more erratic behaviour, on convolutional neural networks and especially standard multilayer perceptrons. See figures 2.1 and 2.2, which were produced for this project with a small CNN, modified slightly from [23] on CIFAR10 [37], with architecture conv-maxpool-conv-maxpool-dropout-FC-dropout-FC, with FC for fully connected and with dropout probability 0.5. The learning rate was 0.005; the momentum decay, 0.95; and the L2 penalty coefficient 0.02. From the figures, it's apparent that saving a

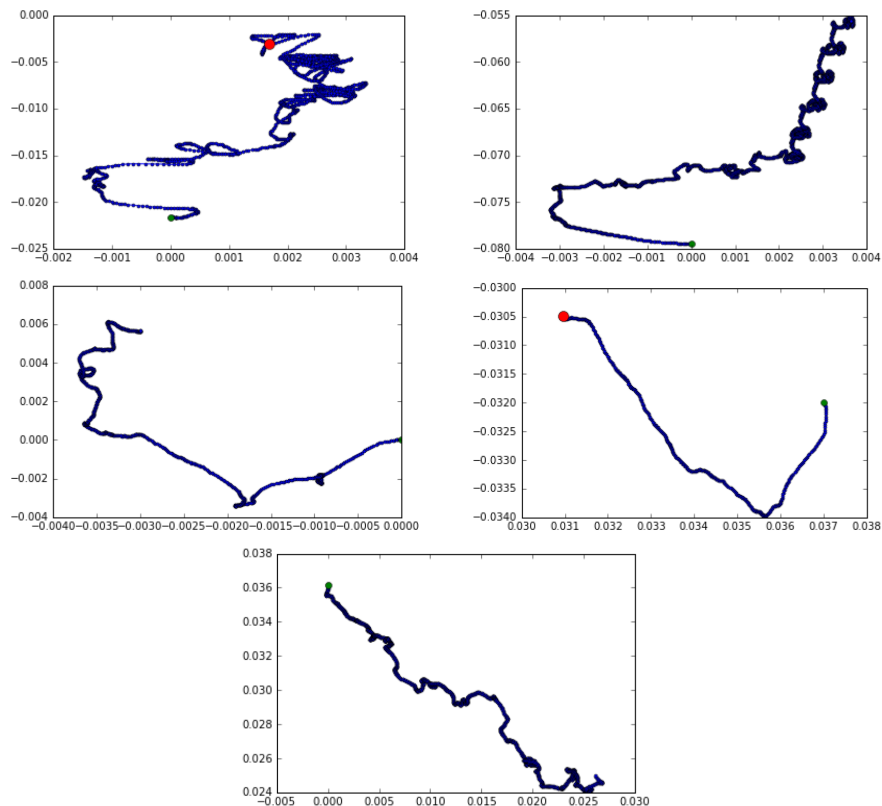


Figure 2.1: Parameter paths projected onto two random coordinates for a small CNN. The green dot is at the initial value, and the red one is at the final value.

few intermediate points and attempting to fit an interpolating spline exactly through them could produce a path that overshoots past sharp turns, while smooth approximating curves often involve hyperparameters for their degree of smoothing, and these would be cumbersome to tune. Even before this, saving the parameters at multiple points during training may infeasible or require the use of smaller models to fit on GPU.

2.2 Hyperparameter Sharing

Hyperparameters often correspond to particular (or groups of) weights, so if a particular weight is shared, i.e. used for multiple inputs to the layer, as is the case for convolutional, scale and bias layer weights (the last two by default in Lasagne [14], and used for batch normalization), then

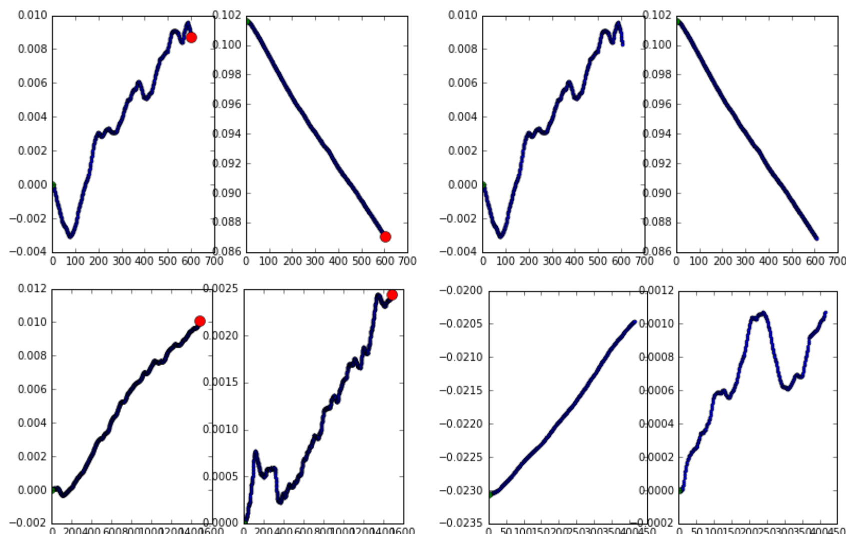


Figure 2.2: Paths as functions of the iteration, for a random coordinate each, for a small CNN. The green dot is at the initial value, and the red one is at the final value.

a hyperparameter, e.g. a learning rate or an L2 penalty coefficient, corresponding to it is also ‘shared’ in the same way. In particular, in computing the hypergradient with respect to such a weight, there will be contributions from all uses of the weight. This may reduce the variance of this hyperparameter’s component of the hypergradient, and make the hyperparameter less prone to overfitting to the validation loss. On the other hand, fully connected/dense layer weights are not shared this way, so it may be preferable to force their corresponding hyperparameters to be shared by unit. This is used in the adaptive method on GPU, but due to time constraints, not DrMAD on GPU.

Scaling hyperparameter optimization to large numbers of hyperparameters with minimal extra sharing is one of the main goals of this project.

2.3 Symmetry

Because neural networks can often have their weights permuted and still compute the same function, as is typically the case when convolutional layers with multiple kernels are used (the kernels can be permuted, and the permutations have to propagate upwards towards the output), it cannot be expected that a particular parameter will end up with even similar values under multiple training runs. [44] illustrated exactly this with random initialization. As such, a *separate* hyperparam-

eter (e.g. L2 regularizer or learning rate) for each weight may not make sense, because the final training parameter corresponding to a hyperparamter may be more similar to a different equivalent parameter in the next training run, or nowhere near any of the previous equivalent weights. However, fixing the random initialization and the sequence of training batches encourages symmetry to be broken the same way. Including hyperparameter velocity could potentially average over the impacts of this poor tying of parameters and hyperparameters when it occurs, while still allowing individually tied parameters and hyperparameters when it doesn't, but because of the huge number of hyperparameters and small number of meta-iterations, this averaging is may not accomplish much.

Unfortunately, in the GPU experiments, I forgot to set a seed for the dropout layers and to reinitialize the batch normalization layers' aggregate means and standard deviations that are computed as *exponential moving averages* [14], which does not affect the elementary gradients directly because only the current batch statistics are used for them, but may have affected the hypergradients and the evaluations on the training, validation and test sets. However, as *exponential* moving averages, and because the epochs in these experiments contained 400 to 703 batches (depending on the training-validation set split) and several epochs were used, it's unlikely this effect from batch normalization is noticeable. Dropout, however, may have introduced enough randomness to break symmetry differently, and experiments suggest this. See figure 2.3.

Another potential source of nondeterminism unaccounted for is apparently the cuDNN GPU backend [12] used by Theano for its convolutions [73].

These are not issues in the adaptive version, since the elementary weights were never reinitialized. In [47], however, they retrained their network a second time with the hyperparameters fixed at their final values, and observed improved performance in the second run. This, of course, should not be done with the learning rates.

2.4 Learning Algorithms: SGD and Variants

Adaptive methods like AdaGrad [18], RMSProp [74] and Adam [35] have been shown to lead to worse generalization [76] despite their widespread use, so SGD with momentum or Nesterov momentum for the elementary iterations are used in the experiments in this project. They are also simpler to reverse exactly, although only regular momentum is.

For the hyperparameters, the vector of the signs of the hypergradient is used, with a constant learning rate, and with momentum on GPU with DrMAD. Following the signs has been shown to approximate the adaptive methods AdaGrad, RMSProp and Adam. [4]

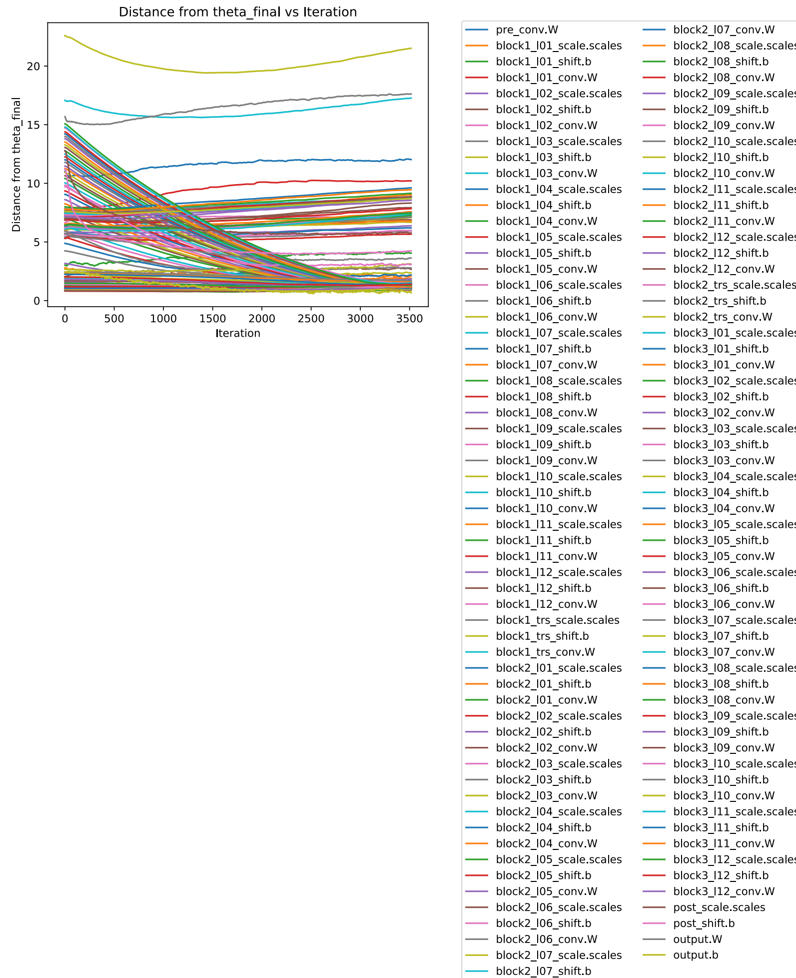


Figure 2.3: Distance of the current parameters in the second meta-iteration from the final parameters in the first meta-iteration, for 5 epochs each. The parameters are reset with the same initial values for the second run, but the seed for dropout is not the same.

The learning rate for the adaptive version and the regularization were parametrized and tuned on a logarithmic scale, as in [48] and as considered in [6, 39], but unlike in the original DrMAD GPU version [23], [47] and [5].

Parameter and hyperparameter clipping is also used in some cases to prevent overflow and nan errors. In the DrMAD algorithm, the elementary parameters are clipped between -10^3 and 10^3 (or projected onto the hypercube with those values at the corners, as in *projected* gradient descent and variants thereof), while in the adaptive version, the log learning rate hyperparameters are clipped to be below 0.

Chapter 3

Experiments

In this chapter, more extensive experiments are performed. In sections 3.1 and 3.2, the results of experiments on CPU and GPU are presented and discussed, with focus on DrMAD [22] and exact hypergradients [48] in the CPU experiments, and DrMAD and an adaptive method in the GPU experiments.

3.1 Experiments on CPU

Experiments are performed to assess the quality of the DrMAD hypergradients, by comparing them in norm and angle with the exact hypergradient, as reported in figures 3.1 and 3.2. The cosine of the angle is calculated as the dot product of the two divided by their norms, $\frac{d\boldsymbol{\theta}_{DrMAD} \cdot d\boldsymbol{\theta}_{exact}}{\|d\boldsymbol{\theta}_{DrMAD}\| \|d\boldsymbol{\theta}_{e}xact\|}$ to determine whether or not the DrMAD directions are descent directions at all, since otherwise they might increase the validation loss. Interestingly, the test error clearly tended to decrease despite often poor agreement between the DrMAD and exact hypergradients (and the vectors of their signs), as reported in 3.3. Further experiments examine how the norm of the exact hypergradients scale during their computation, as in figures 3.4 and 3.5. The values computed during the exact hypergradient computations are compared with and without exact arithmetic in figure 3.6.

The setup of the experiments comes from [23] based on those from [49], written in Python's numpy library [57] with Autograd [50]. These are performed with a small multilayer perceptron with fully connected layers of $50, 50, 50, 10$ units each. The dataset used was MNIST [41], consisting of 60,000 training and 10,000 test 32×32 grayscale images of handwritten digits, 0 to 9, so 10 classes, although only subsets of size 20,000 and 5,000 of the training and test subsets

were used. As in the original experiments, rather than tuning the L2 regularization directly, hyperparameters $\mathbf{s}$ are tuned, after rewriting the parameters as $\mathbf{w} = e^{\mathbf{s}} \cdot \mathbf{z}$, where the multiplication and exponentiation are component-wise, and $\mathbf{s}$ and $\mathbf{z}$ have the same form as $\mathbf{w}$. Then, where g is the cross-entropy on the training set, the elementary loss function is written

$$g(e^{\mathbf{s}} \cdot \mathbf{z}, t) + e^{\lambda}\|\mathbf{z}\|^2 = g(\mathbf{w}, t) + e^{\lambda}\|\mathbf{z}\|^2 = g(\mathbf{w}, t) + e^{\lambda}\|e^{-\mathbf{s}} \cdot \mathbf{w}\|^2$$

and the elementary training was performed with respect to $\mathbf{z}$ rather than $\mathbf{w}$. Had training been performed with respect to $\mathbf{w}$, this would be equivalent to tuning L2 regularization hyperparameters, by the RHS of the loss function above, but since the training is performed in $\mathbf{z}$, the initial scale and regularization are tied and tuned together.

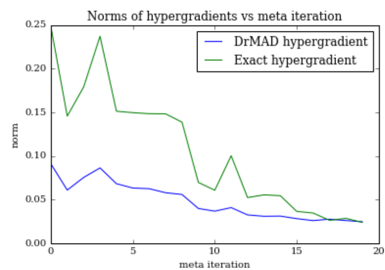
For networks with batch normalization, like DenseNet [30], this rescaling may be redundant, so in the GPU experiments, L2 hyperparameters are tuned directly and scaling is ignored.

The hyperparameters are constrained to be unit-wise, i.e. a hyperparameter is used for each component of the output of a layer, rather than each weight in a layer or simply for each layer. The (pseudo)randomness is determined by a seed, in order to reproduce it exactly in subsequent meta-iterations. The hypergradient steps were in all cases in the direction of the signs of the DrMAD hypergradient.

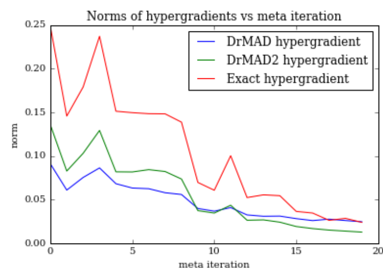
It is worth noting that in some earlier experiments with a single-layer perceptron, too small a validation set, e.g. 19,000 training data and 1000 validation data from the 20,000, actually lead to the hyperparameters overfitting within 200 iterations, i.e. with the validation loss lower than the training loss (without its regularization penalty) and the training loss lower than the test loss. This no longer occured with a 15,000 – 5000 training–validation data split.

In experiments with 2000 reverse path iterations, whether the number of forward iterations was 2000 or 20,000, it was found that the DrMAD hypergradient $d\boldsymbol{\theta}$ in step 8 exploded, being several orders of magnitude greater than the hypergradient in norm.

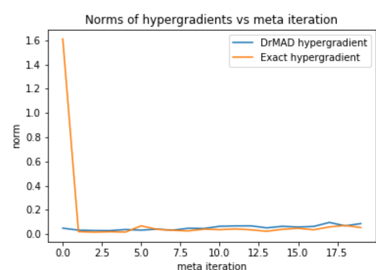
On the other hand, roughly 200 reverse iterations resulted in DrMAD hypergradients of approximately the same order of magnitude as the exact hypergradients for 200 to 200,000 forward iterations, which might suggest that most of the changes to the parameters occur early on in the forward pass, e.g. getting the parameters to roughly the correct scale, and that later iterations make small (but nonetheless important) adjustments therafter; see figure 3.1. However, after the first few meta-iterations, with larger numbers of elementary iterations, there were sometimes negative dot products between the DrMAD and exact hypergradient, as well as between the vectors of signs of the exact and DrMAD hypergradients; see figure 3.2. Despite this issue, the test error continued to decrease; see figure 3.3. Of course, the test errors reported are nowhere near competitive on MNIST; competitive errors are already well below 1%. [41]



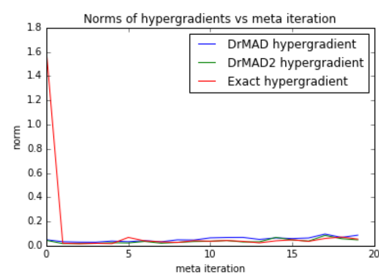
(a) 200 elementary iters per meta



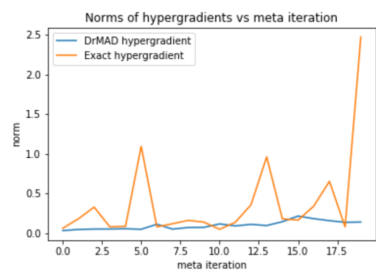
(b) 200 with DrMAD2



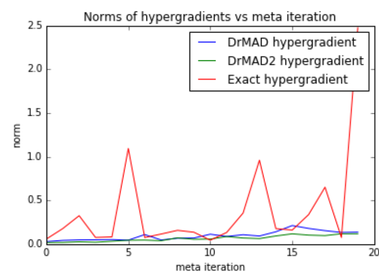
(c) 2000 elementary iters per meta



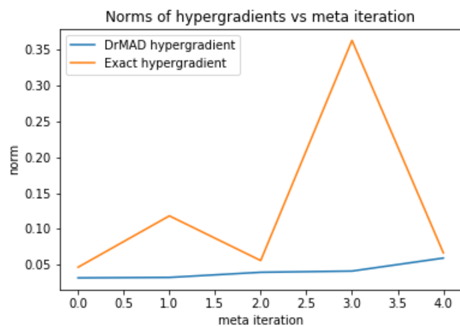
(d) 2000 with DrMAD2



(e) 20,000 elementary iters per meta

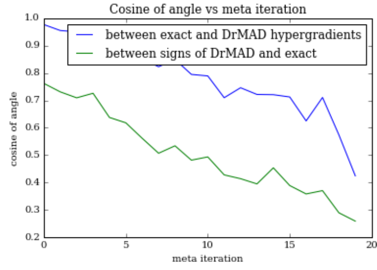


(f) 20,000 with DrMAD2

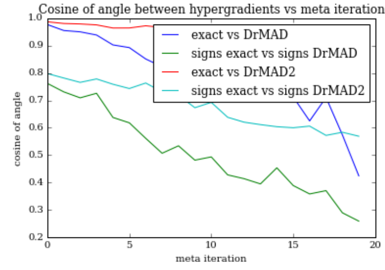


(g) 200,000 elementary iters per meta

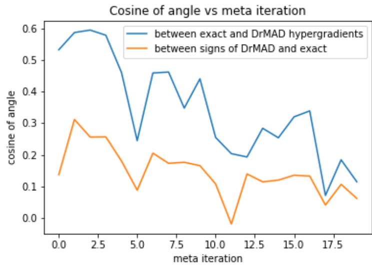
Figure 3.1: Norms of hypergradients following the DrMAD hypergradient signs



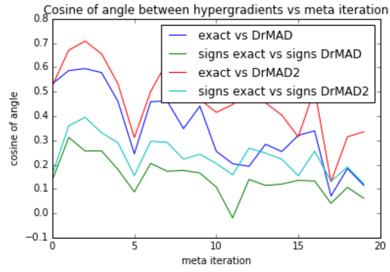
(a) 200 elementary iters per meta



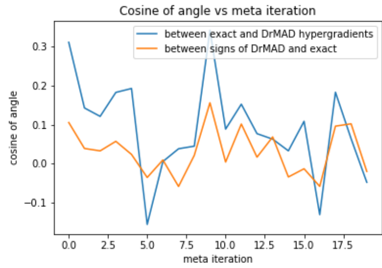
(b) 200 with DrMAD2



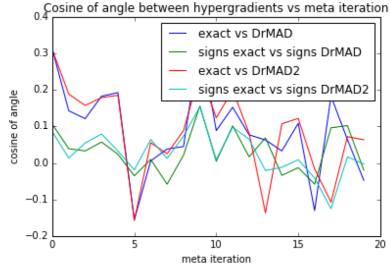
(c) 2000 elementary iters per meta



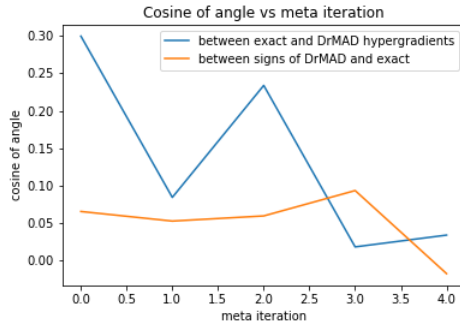
(d) 2000 with DrMAD2



(e) 20,000 elementary iters per meta

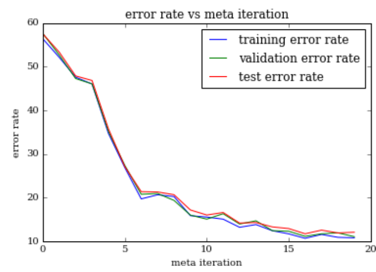


(f) 20,000 with DrMAD2

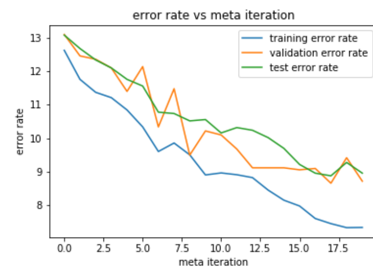


(g) 200,000 elementary iters per meta

Figure 3.2: Cosines of the angles between DrMAD and exact hypergradients and the between the vectors of their signs, calculated as $\frac{d\boldsymbol{\theta}_{DrMAD} \cdot d\boldsymbol{\theta}_{exact}}{\|d\boldsymbol{\theta}_{DrMAD}\| \|d\boldsymbol{\theta}_{e}xact\|}$, following the DrMAD hypergradient signs

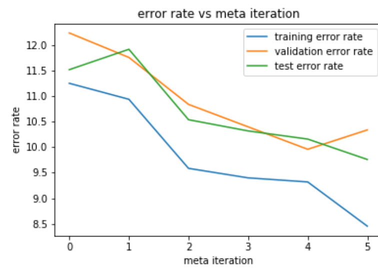


(a) 200 elementary iters per meta



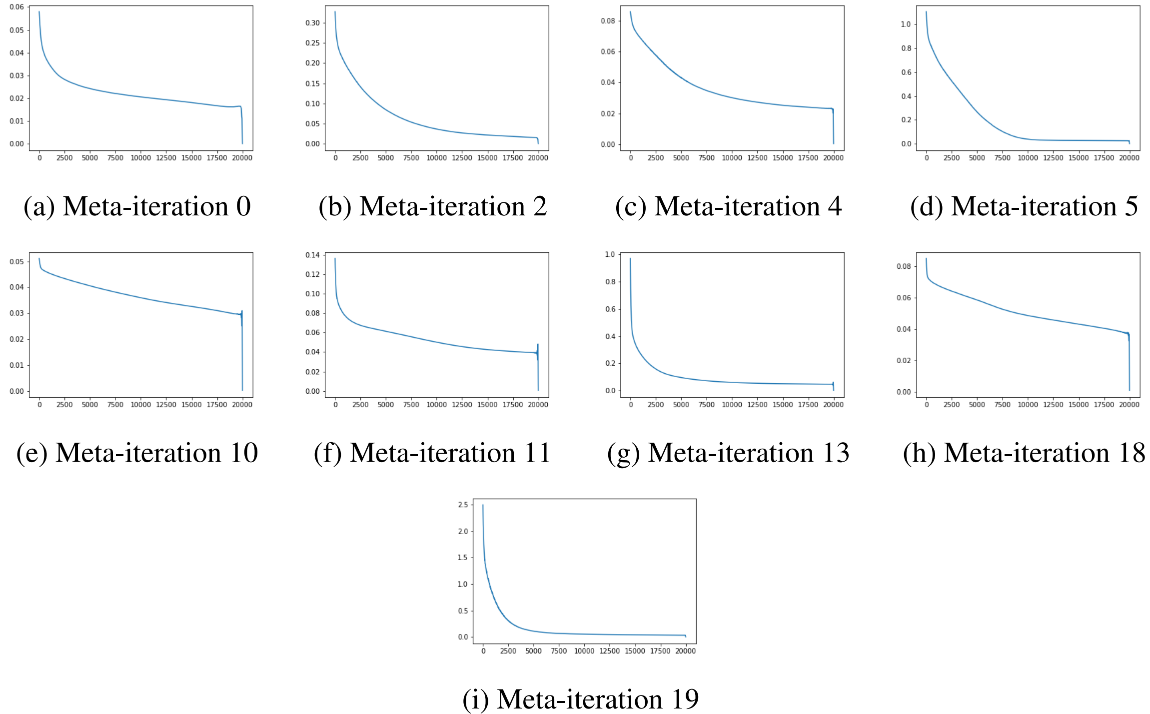
(b) 2000 elementary iters per meta

(c) 20,000 elementary iters per meta



(d) 200,000 elementary iters per meta

Figure 3.3: Error rates following the DrMAD hypergradient signs



(a) Meta-iteration 0 (b) Meta-iteration 2 (c) Meta-iteration 4 (d) Meta-iteration 5

(e) Meta-iteration 10 (f) Meta-iteration 11 (g) Meta-iteration 13 (h) Meta-iteration 18

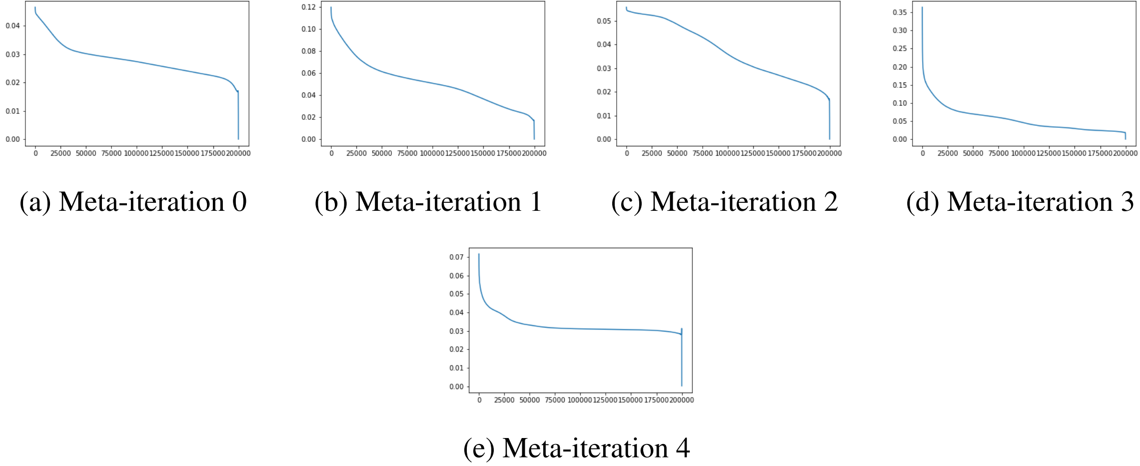
(i) Meta-iteration 19

Figure 3.4: Norms of the accumulated hypergradients for 20,000 elementary iterations, which are accumulated starting with value 0 at $t = T - 1 = 20,000 - 1$ on the right and following the reverse iterations with decreasing t towards the left

The DrMAD hypergradient explosion with 2000 or more but not 200 reverse iterations is likely because the accumulated hypergradients of step 8 of DrMAD and step 12 of the exact hypergradient algorithm of [48] quickly increase in norm near $t = 0$. See figures 3.4 and 3.5.

In this project, including in the experiments on GPU, the number of reverse iterations for DrMAD is calculated as follows:

Let $\mathbf{w}_t$ denote the t-th iterate (from the forward step). Then, with $\beta = 1/T_{reverse}$, where $T_{reverse}$ is the number of reverse steps, and assuming the approximation of $\mathbf{w}_t$ by $t\beta\mathbf{w}_T + (1 - t\beta)\mathbf{w}_0$ is good for small t:



(a) Meta-iteration 0 (b) Meta-iteration 1 (c) Meta-iteration 2 (d) Meta-iteration 3

(e) Meta-iteration 4

Figure 3.5: Norms of the accumulated hypergradients for 200,000 elementary iterations, which are accumulated starting with value 0 at $t = T - 1 = 200,000 - 1$ on the right and following the reverse iterations with decreasing t towards the left.

$$
\begin{aligned}
\frac{\|\mathbf{w}_t - \mathbf{w}_0\|}{\|\mathbf{w}_T - \mathbf{w}_0\|} &\approx \frac{\|t\beta \mathbf{w}_T + (1 - t\beta)\mathbf{w}_0 - \mathbf{w}_0\|}{\|\mathbf{w}_T - \mathbf{w}_0\|} \\
&= \frac{\|t\beta \mathbf{w}_T - t\beta \mathbf{w}_0\|}{\|\mathbf{w}_T - \mathbf{w}_0\|} \\
&= t\beta
\end{aligned}
$$

so that $\beta \approx \frac{\|\mathbf{w}_t - \mathbf{w}_0\|}{t\|\mathbf{w}_T - \mathbf{w}_0\|}$.

Rather than taking β based on a single value of t, a weighted average of the first 20 was used, with most weight given to the earlier values. In the CPU experiments, β was further multiplied by 10 (equivalently, the number of reverse iterations were approximately divided by 10), to get roughly 200 reverse iterations for 2000, 20,000 and 200,000 forward iterations. This unfortunately introduces a hyperhyperparameter. In the DrMAD GPU experiments, β was not multiplied by any factor this way.

Furthermore, if a learning rate schedule is set for the forward pass and fewer reverse iterations are used, what learning rates to use in the reverse pass iterations isn't immediately clear. Generally, one may want the reverse path iterates to use learning rates similar to what was used in the forward pass when the forward iterates were closest.

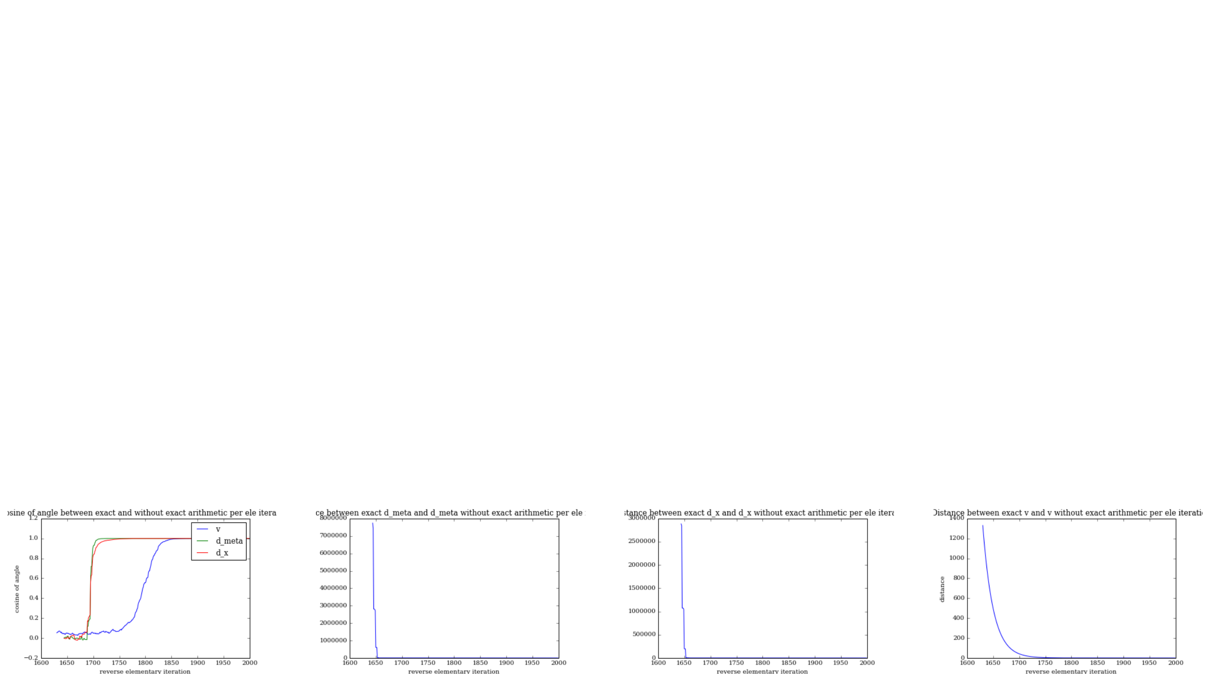
Also of note in figures 3.4 and 3.5 is a quick increase in norm in the exact hypergradients near $t = T - 1$. As such, one modification to DrMAD proposed in this project, called simply *DrMAD2*, takes the exact reverse path for the first 100 or so iterations (without exact arithmetic), and then follows a straight uniformly-spaced path between where this exact reversed path stopped and the initial parameters. For a small number of iterations (200-2000), it seemed to improve the agreement with the exact hypergradient, but for 20,000, this was no longer clear. See figure 3.3.

With $\gamma_t = \gamma = 0.95$ (the momentum decay rate used in these experiments), $-\log_2(0.95) \approx 0.074$ bits are lost on average per multiplication by γ in the forward pass. However, since the gradient $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ will be computed at an approximation of $\mathbf{w}_t$, further bits of $\mathbf{v}_t$ and $\mathbf{w}_t$ may be incorrect at each reverse iteration. The number lost should depend on the network, the loss, the current parameters and velocities. Making a guess that at most 3 times this number of bits would be lost per reverse iteration, so at most $-3\log_2(0.95)$ further bits could be incorrect on average per reverse iteration, and noting that float32 has 23 precision bits, and assuming we're prepared to lose 20 of these, $20/(-3\log_2(0.95)) \approx 90$ reverse iterations would be safe to perform without too much loss of accuracy. Higher momentum would mean more iterations, and lower would mean fewer. Disagreement between the exact method with exact arithmetic and the exact method without exact arithmetic wasn't really observable until about 100 reverse iterations have passed, from the right in plots in figure 3.6. Note, however, that numpy uses float64, not float32, by default, so the factor of 3 was probably too small. In figure 3.6:

- x = $\mathbf{w}_t$ is the current set of elementary parameters during the reverse iteration;
- v $= \mathbf{v}_t$ is the current velocities during the reverse iteration;
- d_meta $= d\boldsymbol{\theta}$ is the current accumulated hypergradient; and
- d_x $= d\mathbf{w}_t$ is the current accumulated hypergradient with respect to the current elementary parameters (eventually the initial parameters).

3.2 Experiments on GPU

The code used here is based on DrMAD's at [23] and the Lasagne DenseNet implementation [64], written with Lasagne [14] and Theano [72] in Python [57]. The experiments were conducted on NVIDIA's Tesla K80 GPU. The dataset used was CIFAR10 [37], consisting of 50,000 training and 10,000 test 32×32 RGB images with 10 classes. The errors and losses were evaluated on cycling batches of size 500-1000 from the training, validation and test sets. It took several hours

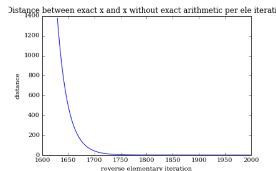


(a) Cosine of the angle between exact with and without exact arithmetic

(b) Distances between d_meta $= d\boldsymbol{\theta}$ with and without exact arithmetic

(c) Distances between d_x $= d\mathbf{w}_t$ with and without exact arithmetic

(d) Distances between v $= \mathbf{v}_t$ with and without exact arithmetic



(e) Distances between x $= \mathbf{w}_t$ with and without exact arithmetic

Figure 3.6: Comparisons of exact reverse path values with and without exact arithmetic on meta-iteration 11 (but updating with the signs of DrMAD hypergradients) for 2000 elementary iterations.

(in some cases nearly a full day) just to compile the Theano function for the Hessian-vector products with the DenseNet described next.

The benchmark to which the experiments are compared is the DenseNet implementation from [64] with dropout and no data augmentation (but normalization as preprocessing; the data augmentation described in some experiments consisted of shifting and flipping images). The network had depth 40, 16 feature maps in the first convolutional layer, a growth rate of 12 and 3 blocks. SGD with Nesterov momentum (as formulated in [7]), with a momentum decay of 0.9. Future experiments should use regular momentum, for which DrMAD is originally derived. [22] Two possibly important differences exist between the network here and those in the original DenseNet paper [30] and the Lasagne implementation [64]. The first is that the biases are not regularized in the experiments here, but they are in [30, 64]. This includes the biases from batch normalization, although in the experiments in this project, the scale weights from batch normalization do have the L2 penalty applied to them, since without it, in principle the scales can increase in scale arbitrary while weights after them decrease in scale, producing identical output, until overflow or underflow.[1] The second discrepancy involves the learning rates. The learning rate started at 0.1 for the first 30 epochs, then 0.01 for the next 30 until 60, and then 0.001 thereafter. This is the second difference: this learning rate schedule is erroneously taken from the ImageNet model in [30]; their CIFAR10 learning rate started at 0.1 and was divided by 10 at 50% and again at 75% of the number of epochs, i.e. at 150 and 225 epochs. The benchmark used here did not beat 10% test error, despite 7% and 6.5 % achieved in [30] and [64], respectively.

The final test error reported with this benchmark was **10.5 %**. The last few test errors were

$$10.5, 10.9, 12.5, 11.0, 9.0, 11.6, 12.7, 11.6, 12.9, 11.6, 11.5, 12.0, 11.1, 11.6, 10.5\ ,$$

and their mean is **11.4%**.

Again, in all experiments, the regularization hyperparameters were tuned on a logarithmic scale (base 10). In the adaptive method, the elementary learning rates were also tuned on a logarithmic scale (base e, but reported in base 10).

Steps were taken in the direction of the vector of signs of the hypergradient, rather than the direction of the hypergradient itself, as in [4] and experiments from [22, 48] at [23, 49].

[1] To illustrate, with x and y representing parameters, if x is penalized but y is not, then $xy = c$ has solutions with $x \neq 0$ and y unbounded, while if $xy = 1$, and both have an L2 penalty applied, this penalty $x^2 + y^2 = x^2 + 1/x^2$ is minimized at $x = y = 1$ and $x = y = -1$. Because x^2 and y^2 have hyperparameter coefficients which may change, and one of x or y could no longer be strongly penalized, clipping is also used.

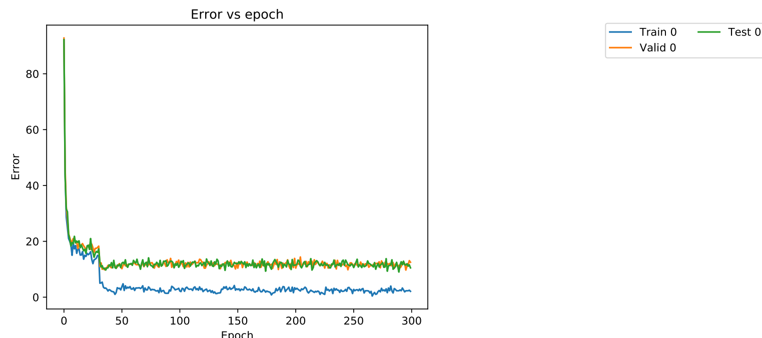


Figure 3.7: DenseNet benchmark errors. The final test error was **11.4%**.

3.2.1 DrMAD

The learning rate for the L2 regularization hyperparameters was set to 0.3333 so that hyperparameter updates could change the order of magnitude of a hyperparameter in at most 3 meta-iterations. Furthermore, rather than simply following the signs, momentum was used, with a rate of 0.7.

The longest running experiment, with each meta-iteration taking roughly a day, started with the L2 hyperparameters intialized at -4 (so coefficient 10^{-4}) , the value used in the Lasagne implementation of DenseNet [64], starting with 100 epochs and increasing the number by a factor of 1.4 each meta-iteration, until a maximum of 300 after which 300 was repeated. In the 9th meta-iteration (meta-iteration 8, starting from 0), the errors eventually became nan. There was no improvement in the test error rate through the use of DrMAD as implemented; the minimum test error occured at the end of the first meta-iteration, *before* any hyperparameter updates. See figures 3.8, 3.9, 3.10.

Bad initial hyperparameters were also tested, starting from 10 epochs and increasing by a factor of 1.4 until 300. An initial L2 regularization hyperarameter of -2 (coefficients 10^{-2}) lead to the prioritization of the minimization of the penalty, an approximately 0.9 error rate (equivalent to random guessing) and hypergradients of norm 0, so that the hyperparameters could not even be improved. See figures 3.11 and 3.12. On the other hand, an initialization at -7 (10^{-7}) lead to nan values in the last few iterations of the last (27th, labelled 26) epoch of the 4th meta-iteration (meta-iteration 3, starting at 0). This happened despite clipping each weight to be in the interval $[-10^3, 10^3]$. Many of the regularization hyperparameters even *decreased*. See figures 3.13 and 3.14.

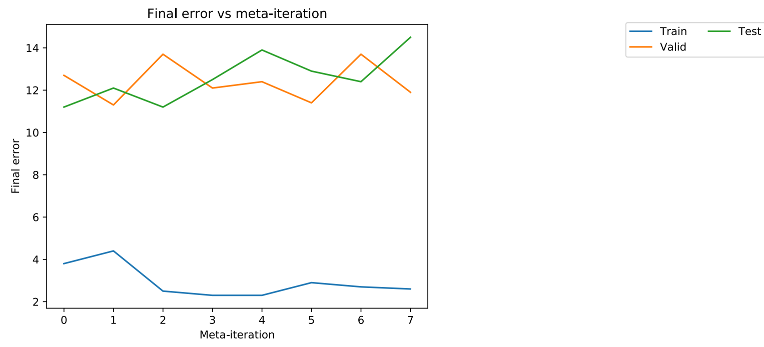


Figure 3.8: Errors per meta-iteration with DrMAD with initial L2 decay 10^{-4}.

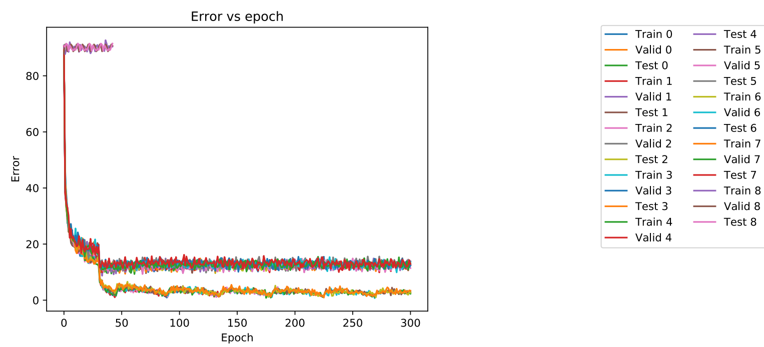


Figure 3.9: Errors during training with DrMAD with initial L2 decay 10^{-4}.

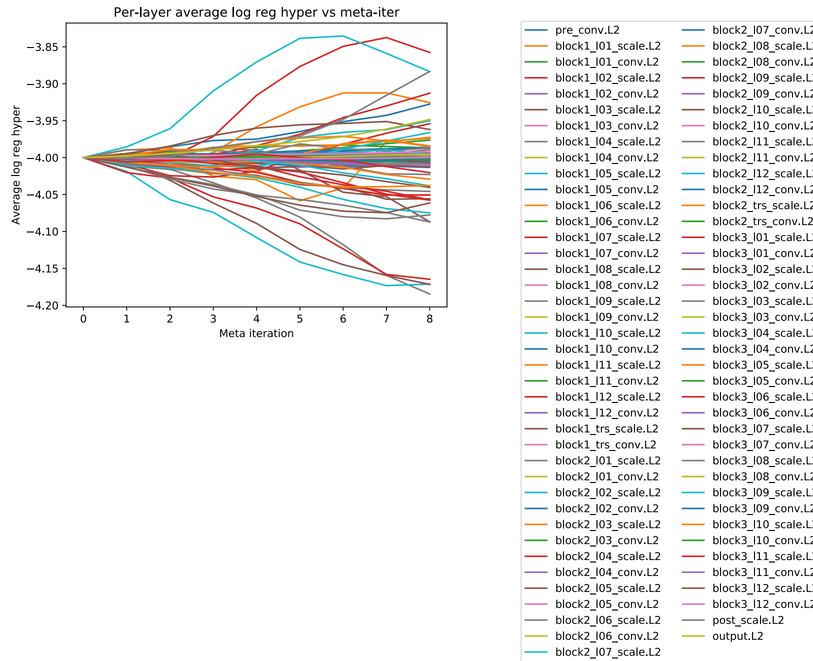


Figure 3.10: Per-layer average log L2 decay hyperparameters with DrMAD, starting at -4.

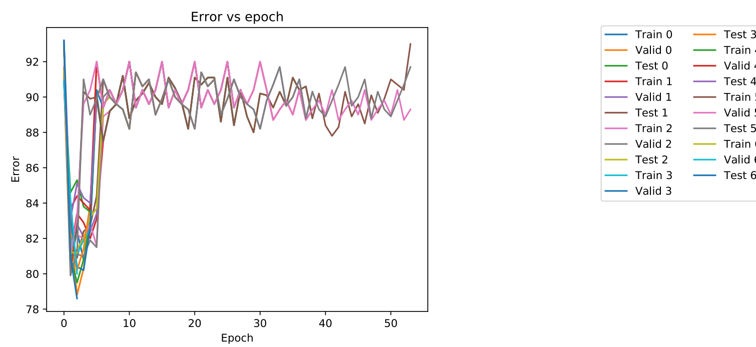


Figure 3.11: Errors during training with DrMAD with initial L2 decay 10^{-2}.

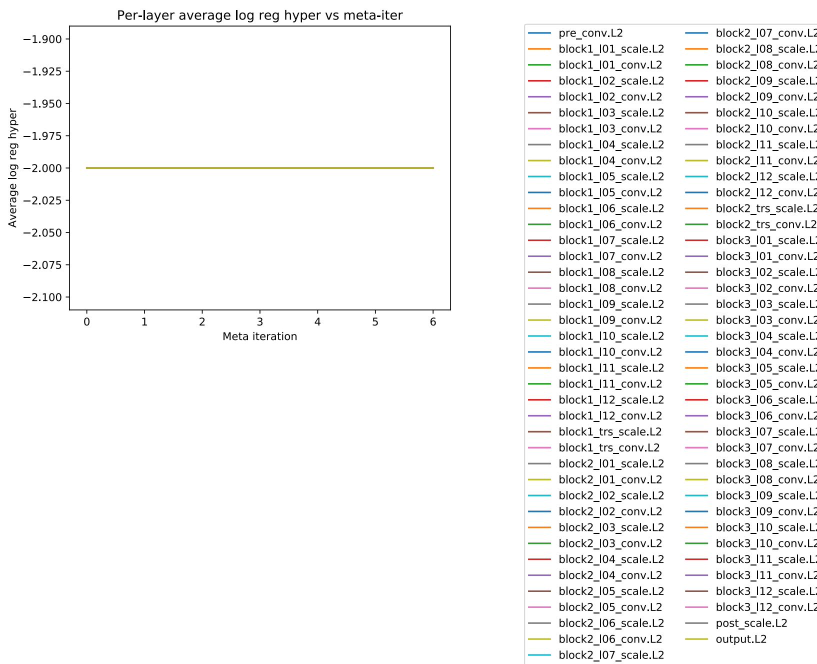


Figure 3.12: Per-layer average log L2 decay hyperparameters with DrMAD, starting at -2.

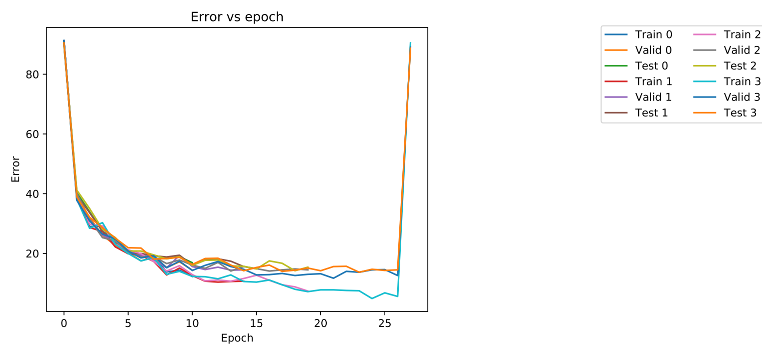


Figure 3.13: Errors during training with DrMAD with initial L2 decay 10^{-7}.

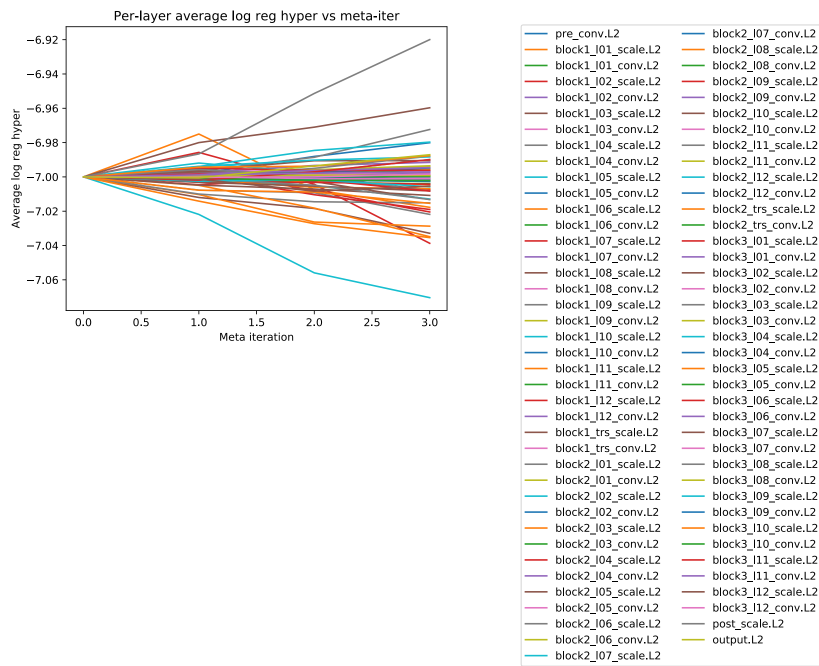


Figure 3.14: Per-layer average log L2 decay hyperparameters with DrMAD, starting at -7.

3.2.2 Adaptive Method

For the first experiment reported here with the adaptive method, the hypergradient was computed and the hyperparameters were updated every 100 elementary iterations, based on the last 5 of those iterations. In [47], instead of 100 and 5, they used 10 and 1. The purpose of this is to amortize the cost of the hypergradient computations, as they are slower than the SGD steps, even if only a constant times slower. No momentum was used for the hyperparameters. The initial elementary learning rate was again set to 0.1, and regular momentum was used with a decay of 0.9, fixed. The learning rate for the L2 regularization hyperparameters was set to 0.01 (in log base 10 scale), while the learning rate for the elementary log learning rates was set to 0.2 (in the natural logarithm scale, but the values are reported in base 10). The training and validation sets had sizes 40,000 and 10,000, respectively. The results are reported in figures 3.15, 3.16, 3.17.

Interestingly, the network started to overfit to the validation set, as the validation error was decreasing away from the test error and towards the training error after roughly the first 30 epochs. The final test error was **15.3%**, which is worse than the benchmark at **11.4%**. The last few test errors were

$$14.5, 15.2, 15.9, 15.1, 16.2, 15.0, 13.8, 15.9, 14.8, 14.2, 15.0, 14.8, 16.4, 14.4, 13.8, 15.3\ ,$$

and their average is approximately **15%**. One potential concern is that the learning rates decreased too quickly, preventing the parameters from escaping a region of poorer performance. A possible solution is to set lower higher lower bounds for the learning rates, e.g. use a more standard learning rate schedule (or, in this case, the benchmark schedule) as a lower bound with which to clip each individual learning rate; this would still allow the learning rates to increase when this can speed up training. In these experiments, however, it's likely the learning rates would have quickly hit and stayed at this lower bound, but this may be useful when the initial learning rate is set too low.

The experiment was repeated with a training-validation set split of 36,000 and 14,000 (with 300 epochs, this decreases the total number of elementary iterations), 131 iterations per hyperparameter update, with the hypergradient computed using the last 4, the elementary parameter momentum decay rate increased to 0.95, the L2 hyperparameter learning rate increased to 0.02 and the log learning rate learning rate decreased to 0.1. The results are reported in figures 3.18, 3.19, 3.20. The final test error was again **15.3%**. However, this was near the maximum among the last few test errors:

$$\begin{gathered}15.7, 13.42, 15.0, 12.14, 12.0, 14.7, 15.1, 14.7, 14.8, 13.7, 12.7,\\ 13.8, 13.6, 12.7, 9.7, 12.6, 11.3, 10.6, 12.8, 13.0, 12.3, 14.0, 12.3,\\ 12.0, 13.7, 13.0, 12.4, 12.8, 11.4, 12.4, 11.7, 15.0, 14.7, 15.4, 15.3\ .\end{gathered}$$

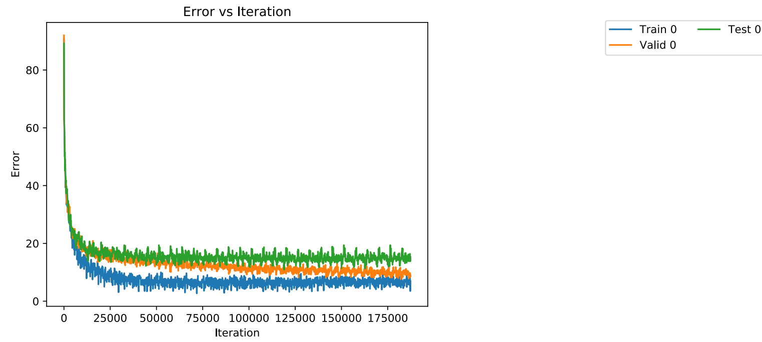


Figure 3.15: Errors with the adaptive method, first experiment

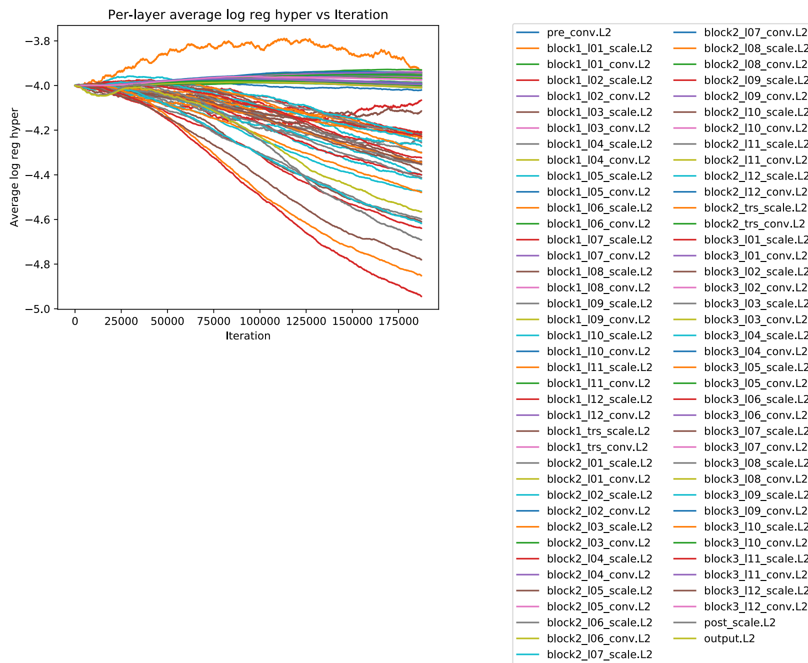


Figure 3.16: Per-layer average log L2 decay hyperparameters with the adaptive method, first experiment

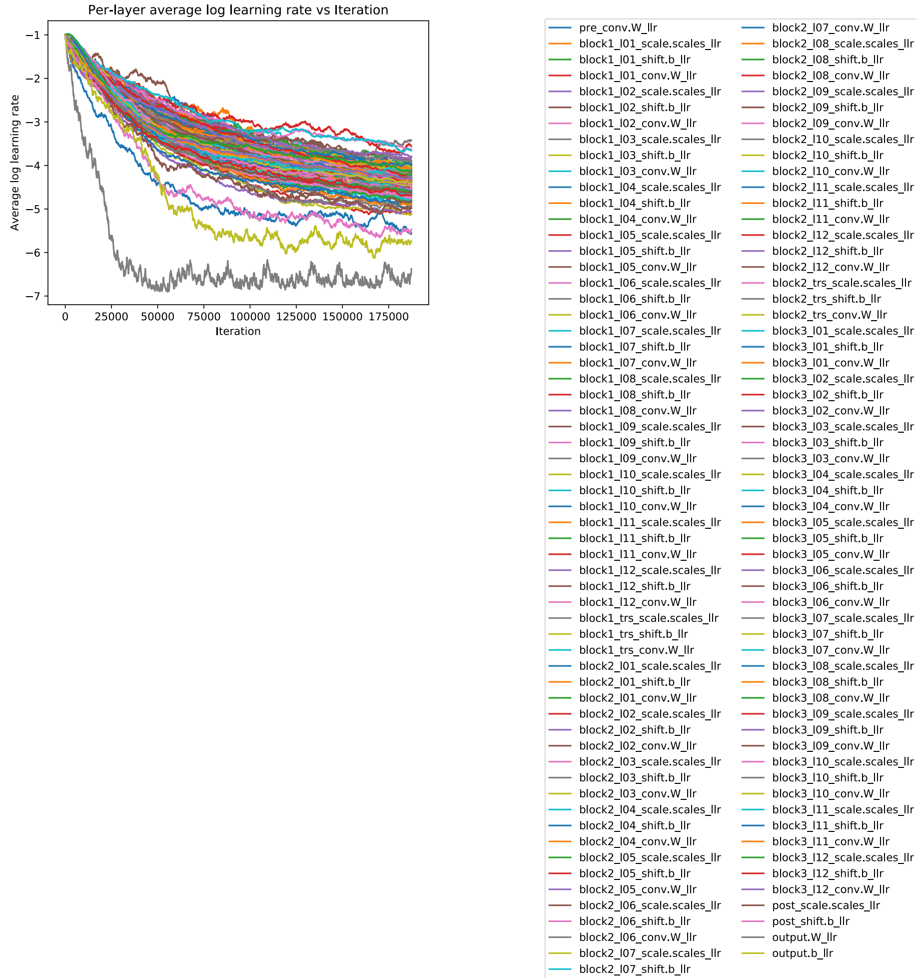


Figure 3.17: Per-layer average log learning rates with the adaptive method, first experiment

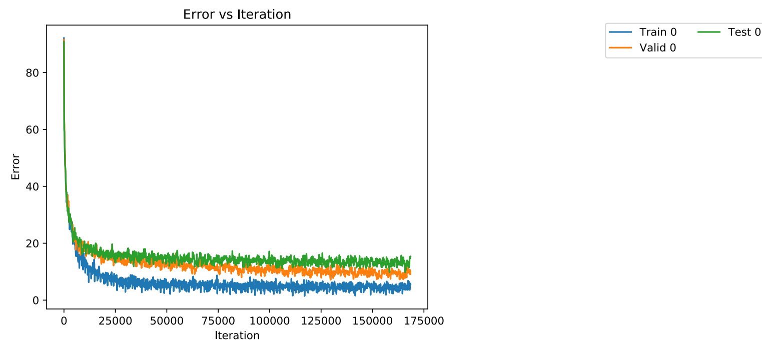


Figure 3.18: Errors with the adaptive method, second experiment

The average of these is approximately **13%**. Further changes in these directions to the adaptive method should be explored.

Simultaneously decreasing the log L2 hyperparameter learning rate to 0.01, while increasing the number of examples in the validation set to 22,000 (so training set decreased to 28,000) and using narrower clipping on the log-learning rates *worsened* the test performance, despite making the validation and test set performances closer. The number of epochs was even increased to 600 (to compensate for the smaller training set).

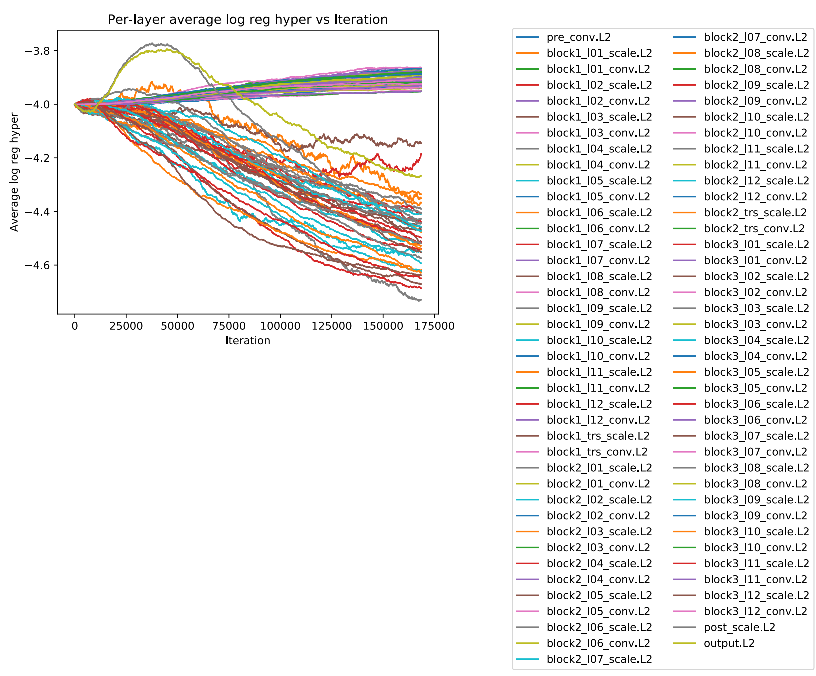


Figure 3.19: Per-layer average log L2 decay hyperparameters with the adaptive method, second experiment

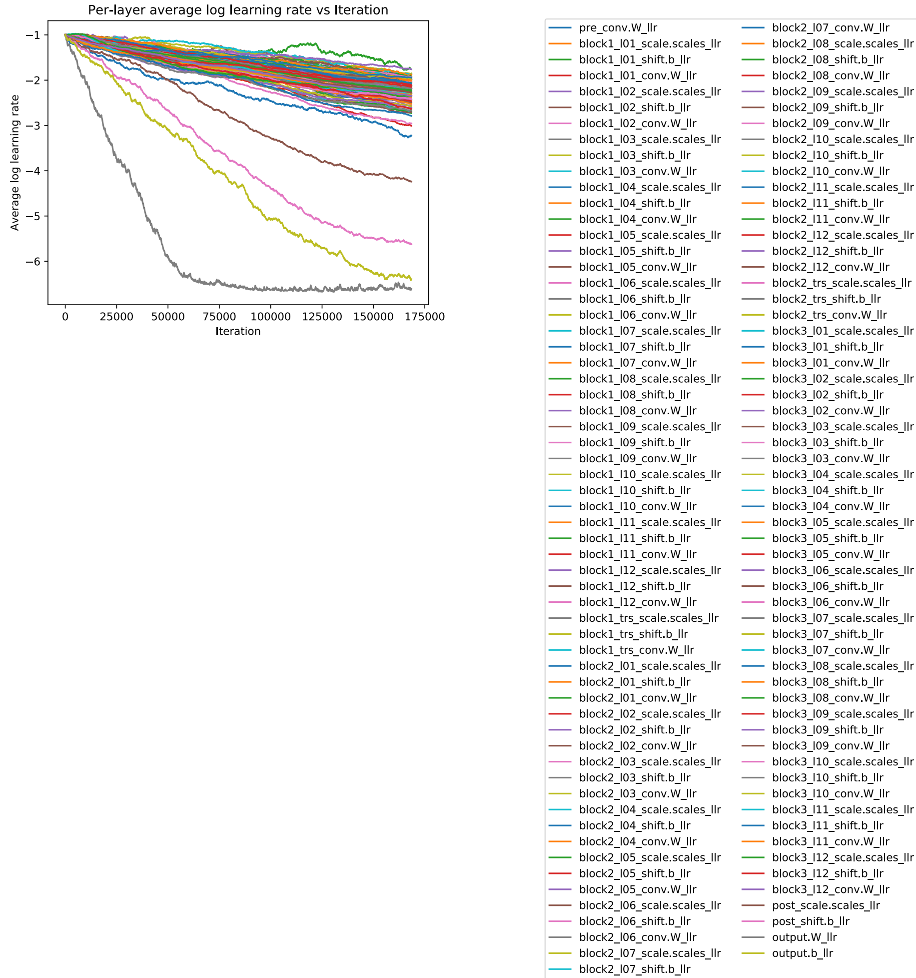


Figure 3.20: Per-layer average log learning rates with the adaptive method, second experiment

Chapter 4

Discussion and Directions for Future Work

It is still very unclear whether or not DrMAD can be made useful for per-parameter hyperparameters, although experiments with properly fixed randomness or with more hyperparameter sharing to avoid the decoupling of hyperparameters and parameters should be conducted. Experiments with regular momentum instead of Nesterov momentum would also be more consistent with the original derivation of DrMAD.[22]

The adaptive method shows some promise. Further experiments of interest include initialization with poor hyperparameters and to address overfitting to the validation sets by increasing their size and taking fewer samples from them.

More hyperparameters could also be tuned in future work, e.g. additive noise to the units as in [47]. Variants of SGD with added noise [11, 17, 26, 34, 42], which have shown promise for escaping saddle points, could have the scale of their noise tuned, most likely with the adaptive method, since it seems unlikely that DrMAD can be made sensitive enough to such noise.

References

[1] Luís Almeida, Thibault Langlois, José D Amaral, and Alexander Plakhov. Parameter adaptation in stochastic optimization. *On-Line Learning in Neural Networks, Publications of the Newton Institute*, pages 111–134, 1998.

[2] Luís Almeida, Thibault Langlois, D Amaral Jos'e, et al. On-line step size adaptation. In *INESC. 9 Rua Alves Redol, 1000*. Citeseer, 1997.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[4] Lukas Balles and Philipp Hennig. Follow the signs for robust stochastic optimization. *arXiv preprint arXiv:1705.07774*, 2017.

[5] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. *arXiv preprint arXiv:1703.04782*, 2017.

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

[7] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

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

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

[10] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[11] Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic Gradient Hamiltonian Monte Carlo. In *International Conference on Machine Learning*, pages 1683–1691, 2014.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[13] 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.

[14] Sander Dieleman, Jan Schlter, Colin Raffel, Eben Olson, Sren Kaae Snderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gbor Takcs, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. Lasagne: First release., August 2015.

[15] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

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

[17] Simon S Du, Chi Jin, Jason D Lee, Michael I Jordan, Barnabas Poczos, and Aarti Singh. Gradient descent can take exponential time to escape saddle points. *arXiv preprint arXiv:1705.10412*, 2017.

[18] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[19] 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.

[20] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. *arXiv preprint arXiv:1703.01785*, 2017.

[21] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. On hyperparameter optimization in learning systems. 2017.

[22] Jie Fu, Hongyin Luo, Jiashi Feng, Kian Hsiang Low, and Tat-Seng Chua. DrMAD: distilling reverse-mode automatic differentiation for optimizing hyperparameters of deep neural networks. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1469–1475. AAAI Press, 2016.

[23] Jie Fu, Hongyin Luo, Jiashi Feng, Kian Hsiang Low, and Tat-Seng Chua. DrMAD: distilling reverse-mode automatic differentiation for optimizing hyperparameters of deep neural networks (code), 2016.

[24] Marcus Gallagher and Tom Downs. Visualization of learning in neural networks using principal component analysis. In *in Proc. International Conference on Computational Intelligence and Multimedia Applications, Gold*. Citeseer, 1997.

[25] Marcus Gallagher and Tom Downs. Visualization of learning in multilayer perceptron networks using principal component analysis. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 33(1):28–34, 2003.

[26] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle pointsonline stochastic gradient for tensor decomposition. In *Conference on Learning Theory*, pages 797–842, 2015.

[27] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv:1412.6544*, 2014.

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

[29] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[30] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

[31] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration.

[32] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates. In *AAAI*, pages 822–829, 2017.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[34] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. How to escape saddle points efficiently. *arXiv preprint arXiv:1703.00887*, 2017.

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

[36] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536, 2017.

[37] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Jan Larsen, Claus Svarer, Lars Nonboe Andersen, and Lars Kai Hansen. Adaptive regularization in neural network modeling. In *Neural Networks: Tricks of the Trade*, pages 113–132. Springer, 1998.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[42] Chunyuan Li, Changyou Chen, David E Carlson, and Lawrence Carin. Preconditioned stochastic gradient langevin dynamics for deep neural networks. In *AAAI*, volume 2, page 4, 2016.

[43] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[44] Zachary C Lipton. Stuck in a what? adventures in weight space. *arXiv preprint arXiv:1602.07320*, 2016.

[45] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. Particle swarm optimization for hyper-parameter selection in deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 481–488. ACM, 2017.

[46] Pablo Ribalta Lorenzo, Jakub Nalepa, Luciano Sanchez Ramos, and José Ranilla Pastor. Hyper-parameter selection in deep neural networks using parallel particle swarm optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1864–1871. ACM, 2017.

[47] Jelena Luketina, Mathias Berglund, Klaus Greff, and Tapani Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. In *International Conference on Machine Learning*, pages 2952–2960, 2016.

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

[49] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning (code), 2015.

[50] Dougal Maclaurin, David Duvenaud, and Matt Johnson. Autograd: Efficiently computes derivatives of numpy code. https://github.com/HIPS/autograd, 2015.

[51] Pierre-Yves Massé and Yann Ollivier. Speed learning on the fly. *arXiv preprint arXiv:1511.02540*, 2015.

[52] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[53] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pages 543–547, 1983.

[54] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011.

[55] Barak A Pearlmutter. Fast exact multiplication by the Hessian. *Neural computation*, 6(1):147–160, 1994.

[56] Fabian Pedregosa. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pages 737–746, 2016.

[57] Python Software Foundation. Python language reference.

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

[59] Afshin Rostamizadeh, Ameet Talwalkar, Giulia DeSalvo, Kevin Jamieson, and Lisha Li. Efficient hyperparameter optimization and infinitely many armed bandits. In *5th International Conference on Learning Representations*, 2017.

[60] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[61] Levent Sagun, Léon Bottou, and Yann LeCun. Singularity of the hessian in deep learning. *arXiv preprint arXiv:1611.07476*, 2016.

[62] Levent Sagun, Utku Evci, V Ugur Guney, Yann Dauphin, and Leon Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2017.

[63] Levent Sagun, Utku Evci, V Ugur Guney, Yann Dauphin, and Leon Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2017.

[64] Jan Schlter. Densely Connected Convolutional Network (DenseNet) in Lasagne, 2017.

[65] Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

[66] 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.

[67] 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.

[68] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015.

[69] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust Bayesian neural networks. In *Advances in Neural Information Processing Systems*, pages 4134–4142, 2016.

[70] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[71] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[72] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[73] Theano Development Team. Theano: theano.sandbox.cuda.dnn cuDNN. 2017.

[74] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. 2012.

[75] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[76] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.

[77] Anqi Wu, Mikio C Aoi, and Jonathan W Pillow. Exploiting gradients and Hessians in Bayesian optimization and bayesian quadrature. *arXiv preprint arXiv:1704.00060*, 2017.

[78] Jian Wu, Matthias Poloczek, Andrew Gordon Wilson, and Peter I Frazier. Bayesian optimization with gradients. *arXiv preprint arXiv:1703.04389*, 2017.

[79] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[80] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.