Gradient-based Hyperparameter Optimization through Reversible Learning

Dougal Maclaurin[†] MACLAURIN@PHYSICS.HARVARD.EDU
David Duvenaud[†] DDUVENAUD@SEAS.HARVARD.EDU
Ryan P. Adams RPA@SEAS.HARVARD.EDU

Abstract

Tuning hyperparameters of learning algorithms is hard because gradients are usually unavailable. We compute exact gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives backwards through the *entire training procedure*. These gradients allow us to optimize thousands of hyperparameters, including step-size and momentum schedules, weight initialization distributions, richly parameterized regularization schemes, and neural network architectures. We compute hyperparameter gradients by exactly reversing the dynamics of stochastic gradient descent with momentum.

1. Introduction

Machine learning systems abound with hyperparameters. These can be parameters that control model complexity, such as L_1 and L_2 penalties, or parameters that specify the learning procedure itself – step sizes, momentum decay parameters and initialization conditions. Choosing the best hyperparameters is both crucial and frustratingly difficult.

The current gold standard for hyperparameter selection is gradient-free model-based optimization (Snoek et al., 2012; Bergstra et al., 2011; 2013; Hutter et al., 2011). Hyperparameters are chosen to optimize the validation loss after complete training of the model parameters. These approaches have demonstrated that automatic tuning of hyperparameters can yield state-of-the-art performance. However, in general they are not able to effectively optimize more than 10 to 20 hyperparameters.

Why not use gradients? Reverse-mode differentiation allows gradients to be computed with a similar time cost to the original objective function. This approach is taken almost universally for optimization of elementary[1] parameters. The problem with taking gradients with respect to hyperparameters is that computing the validation loss requires an inner loop of elementary optimization, which makes naïve reverse-mode differentiation infeasible from a memory perspective. Section 2 describes this problem and proposes a solution, which is the main technical contribution of this paper.

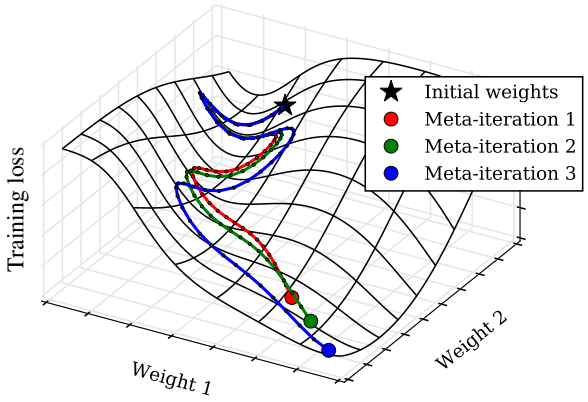

Figure 1. Hyperparameter optimization by gradient descent. Each meta-iteration runs an entire training run of stochastic gradient descent to optimize elementary parameters (weights 1 and 2). Gradients of the validation loss with respect to hyperparameters are then computed by propagating gradients back through the elementary training iterations. Hyperparameters (in this case, learning rate and momentum schedules) are then updated in the direction of this hypergradient.

Gaining access to gradients with respect to hyperparamters opens up a garden of delights. Instead of straining to eliminate hyperparameters from our models, we can embrace them, and richly hyperparameterize our models. Just as having a high-dimensional elementary parameterization gives a flexible model, having a high-dimensional hyperparameterization gives flexibility over model classes, regularization, and training methods. Section 3 explores these new opportunities.

1.1. Contributions

- We give an algorithm that exactly reverses stochastic gradient descent with momentum to compute gradi-

[†]The order of these two authors is random. See github.com/hips/author-roulette

[1]Since this paper is about hyperparameters, we use "elementary" to unambiguously denote the other sort of parameter, the "parameter-that-is-just-a-parameter-and-not-a-hyperparameter".

ents with respect to all continuous training parameters.

- We show how to efficiently store only the information needed to exactly reverse learning dynamics. For example, when the momentum term is 0.9, this method reduces the memory requirements of reverse-mode differentiation of hyperparameters by a factor of 200.
- We show that these gradients allow optimization of validation loss with respect to thousands of hyperparameters. For example, we optimize fine-grained learning-rate schedules, per-layer initialization distributions of neural network parameters, per-input regularization schemes, and per-pixel data preprocessing.
- We provide insight into learning procedures by examining optimized learning-rate schedules and initialization procedures, comparing them to standard advice in the literature.

2. Hypergradients

Reverse-mode differentiation (RMD) has been an asset to the field of machine learning (LeCun et al., 1989) (see the 7 for a refresher). The RMD method, known as "back-propagation" in the deep learning community, allows the gradient of a scalar loss with respect to its parameters to be computed in a single backward pass. This increases the computational burden by only a factor of two over evaluating the loss itself, regardless of the number of parameters. Obtaining the same sort of information by either forward-mode differentiation or brute force finite differences would require a separate pass for each parameter and would make deep learning entirely infeasible.

Applying RMD to hyperparameter optimization was proposed by Bengio (2000) and Baydin & Pearlmutter (2014), and applied to small problems by Domke (2012). However, the naïve approach fails for real-sized problems because of memory constraints. RMD requires that intermediate variables be maintained in memory for the reverse pass. Evaluating the validation loss requires training the model, which may require many elementary iterations. Conventional RMD stores this entire training trajectory, $\mathbf{w}_1 \ldots \mathbf{w}_T$ in memory. In large neural networks, the amount of memory required to store the millions of parameters being trained is typically close to the amount of physical RAM available (Sutskever et al., 2014). If storing the parameter vector takes $\sim$1GB, and the parameter vector is updated tens of thousands of times (the number of mini batches times the number of epochs) then storing the learning history is unmanageable even with physical storage.

Imagine that we could exactly trace a training procedure backwards, starting from the trained parameter values and working back to the initial parameters. Then we could recompute the learning trajectory on the fly during the reverse pass of RMD rather than storing it in memory. This is not possible in general, but we will show that for the popular training procedure of stochastic gradient descent with momentum, we can do exactly this, storing a small number of auxiliary bits to handle finite precision arithmetic.

2.1. Reversible learning with exact arithmetic

Stochastic gradient descent (SGD) with momentum (Algorithm 1) can be seen as a physical simulation of a system moving through a series of fixed force fields indexed by time t. With exact arithmetic this procedure is reversible. This lets us write Algorithm 2, which reverses the steps in Algorithm 1, interleaved with computations of gradients. It outputs the gradient of a function of the trained weights $f(\mathbf{w})$ (such as the validation loss) with respect to the initial weights $\mathbf{w}_1$, the learning-rate and momentum schedules, and any other hyperparameters which affect training gradients.

Algorithm 1 Stochastic gradient descent with momentum

input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

Algorithm 2 Reverse-mode differentiation of SGD

input: $\mathbf{w}_T$, $\mathbf{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\gamma = \mathbf{0}$
initialize $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
for $t = T$ **counting down to** 1 **do**
  $d\alpha_t = d\mathbf{w}^\mathsf{T} \mathbf{v}_t$
  $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$ } exactly reverse
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ } gradient descent
  $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$ } operations
  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
  $d\gamma_t = d\mathbf{v}^\mathsf{T}(\mathbf{v}_t + \mathbf{g}_t)$
  $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\mathbf{v} = \gamma_t d\mathbf{v}$
end for
output gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_1$, $\mathbf{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

Computations of steps 11 and 12 both require a Hessian-vector product, but these can be computed exactly by applying RMD to the dot product of the gradient with a vector (Pearlmutter, 1994). Thus the time complexity of reverse SGD is $\mathcal{O}(T)$, the same as forward SGD.

2.2. Reversible learning with finite precision arithmetic

In practice, Algorithm 2 fails utterly due to finite numerical precision. The problem is the momentum decay term γ. Every time we apply step 8 to reduce the velocity, we lose information. Assuming we are using a fixed-point representation, [2] each multiplication by $\gamma < 1$ shifts bits to the right, destroying the least significant bits. This is more than a pedantic concern. Attempting to carry out the reverse training requires repeated multiplication by $1/\gamma$. Errors accumulate exponentially, and the reversed learning procedure ends far from the initial point (and usually overflows). Do we need $\gamma < 1$? Unfortunately we do. $\gamma > 1$ results in unstable dynamics, and $\gamma = 1$, recovers the leapfrog integrator (Hut et al., 1995), a perfectly reversible set of dynamics, but one that does not converge.

This problem is quite a deep one: optimization necessarily discards information. Ideally, optimization maps all initializations to the same optimum, a many-to-one mapping with no hope of inversion. Put another way, optimization moves a system from a high-entropy initial state to a low-entropy (hopefully zero entropy) optimized final state.

It is interesting to consider the analogy with physical dynamics. The γ term is analogous to a drag term in the simulation of Hamiltonian dynamics. Having $\gamma < 1$ corresponds to *dissipative* dynamics which generates heat, increases the entropy of the environment and is not therefore not reversible. But we must have dissipation in order for our system to converge to equilibrium.

If we want to reverse the dynamics, there is no choice but to store the extra bits discarded by the γ operation. But we can at least try to be parsimonious about the number of extra bits we store. This is what the next section addresses.

2.3. Optimal storage of discarded entropy

This section gives the technical details of how to efficiently store the information discarded each time the momentum decay operation (Step 8) is applied.

If $\gamma = 0.5$, we can simply store the single bit that falls off at each iteration, and if $\gamma = 0.25$ we could store two bits. But for fine-grained control over γ we need a way to store the information lost when we multiply by, say, $\gamma = 0.9$, which will be less than one bit on average. Here we give a procedure which achieves exactly this.

We represent the velocity $\mathbf{v}$ and parameter $\mathbf{w}$ vectors with 64-bit integers. With an implied radix point this can be a fixed-point representation of the reals. We represent γ as a rational number, n/d. When we divide each v by d we use integer division. In order to be able to reverse the process we just need to store the remainder, v modulo s, in some "information buffer", B. If B were an integer and $n = 2$, the remainder r would just be a single bit, and we could store it in B by left-shifting B's bits and adding r. For arbitrary n, we can do the base-n analogue of this operation: multiply B by n and add r. Eventually, B will overflow. We need a way to either detect this, store the bits, and start a fresh integer, or else we can just use an arbitrary size integer that grows as needed. (Python's "long" integer type supports this). This procedure allows division by n while storing the remainder in $\log_2(n)$ bits on average.

When we multiply by the numerator of n/d we don't need to store anything extra, since integer division will bring us back to exactly the same point anyway. But the procedure as it stands would store three bits when $\gamma = 7/8$, whereas it should store less than one ($\log_2(8/7) = 0.19$). Our solution is the following: when we multiply v by n, there is an opportunity to add a nonnegative integer smaller than n to the result without affecting the reverse process (integer division by n). We can get such an integer from the information buffer by dividing it by n and recording B modulo n. We are using the velocity v as an information buffer itself! Algorithm 3 illustrates the entire process.

Algorithm 3 Exactly reversible multiplication by a ratio

1: **Input:** Information buffer i, value c, ratio n/d
2: $i = i \times d$ ▷ make room for new digit
3: $i = i + (c \mod d)$ ▷ store digit lost by division
4: $c = c \div d$ ▷ divide by denominator
5: $c = c \times n$ ▷ multiply by numerator
6: $c = c + (i \mod n)$ ▷ add digit from buffer
7: $i = i \div n$ ▷ shorten information buffer
8: **return** updated buffer i, updated value c

We could also have used an arithmetic coding scheme for our information buffer (MacKay, 2003, Chapter 6). How much does this procedure save us? When $\gamma = 0.98$, we will have to store only 0.029 bits on average. Compared to storing a new 32-bit integer or floating-point number at each iteration, this reduces memory requirements by a factor of one thousand.

The standard way to save memory in RMD is checkpointing. Checkpointing stores the entire parameter vector on only a fraction of the training steps, and recomputes the missing steps of the training procedure (forwards) as needed during the backward pass. However, this would require too much memory to be practical for large neural nets trained for thousands of minibatches.

[2] We assume fixed-point representation to simplify the discussion (and the implementation). Courbariaux et al. (2014) show that fixed-point arithmetic is sufficient to train deep networks. Floating point representation doesn't fix the problem, it just defers the loss of information from the division step to the addition step.

3. Experiments

In typical machine learning applications, only a few hyperparameters (less than 20) are optimized. Since each experiment only yields a single number (the validation loss), the search rapidly becomes more difficult as the dimension of the hyperparameter vector increases. In contrast, when hypergradients are available, the amount of information gained from each training run grows along with the number of hyperparameters, allowing us to optimize thousands of hyperparameters. How can we take advantage of this new ability?

This section shows several proof-of-concept experiments in which we can more richly parameterize training and regularization schemes in ways that would have been previously impractical to optimize.

3.1. Gradient-based optimization of gradient-based optimization

Modern neural net training procedures often employ various heuristics to set learning rate schedules, or set their shape using one or two hyperparameters set by cross-validation (Dahl et al., 2014; Sutskever et al., 2013). These schedule choices are supported by a mixture of intuition, arguments about the shape of the objective function, and empirical tuning.

To more directly shed light on good learning rate schedules, we jointly optimized separate learning rates for *every single learning iteration* of training of a deep neural network, as well as separately for weights and biases in each layer. Each meta-iteration trained a network for 100 iterations of SGD, meaning that the learning rate schedules were specified by 800 hyperparameters (100 iterations $\times$ 4 layers $\times$ 2 types of parameters). To avoid learning an optimization schedule that depended on the quirks of a particular random initialization, each evaluation of hypergradients used a different random seed. These random seeds were used both to initialize network weights and to choose mini batches. The network was trained on 10,000 examples of MNIST, and had 4 layers, of sizes 784, 50, 50, and 50.

Because learning schedules can implicitly regularize networks (Erhan et al., 2010), for example by enforcing early stopping, for this experiment we optimized the learning rate schedules on the training error rather than on the validation set error. Figure 2 shows the results of optimizing learning rate schedules separately for each layer of a deep neural network. When Bayesian optimization was used to choose a fixed learning rate for all layers and iterations, it chose a learning rate of 2.4.

Meta-optimization strategies We experimented with several standard stochastic optimization methods for meta-optimization, including SGD, RMSprop (Tieleman & Hinton, 2012), and minibatch conjugate gradients. The results in this section used Adam (Kingma & Ba, 2014), a variant of RMSprop that includes momentum. We typically ran for 50 meta-iterations, and used a meta-step size of 0.04. Figure 3 shows the elementary and meta-learning curves that generated the hyperparameters shown in Figure 2.

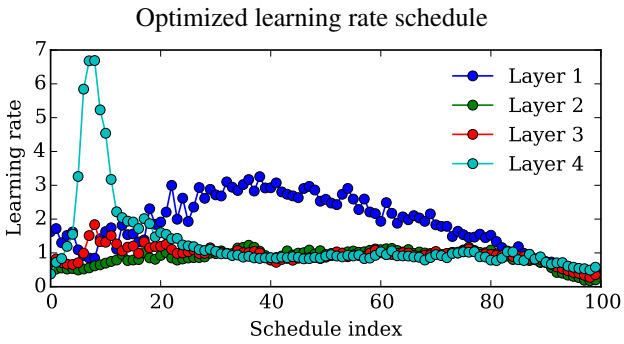

Figure 2. A learning-rate training schedule for the weights in each layer of a neural network, optimized by hypergradient descent. The optimized schedule starts by taking large steps only in the topmost layer, then takes larger steps in the first layer. All layers take smaller step sizes in the last 10 iterations. Not shown are the schedules for the biases or the momentum, which showed less structure.

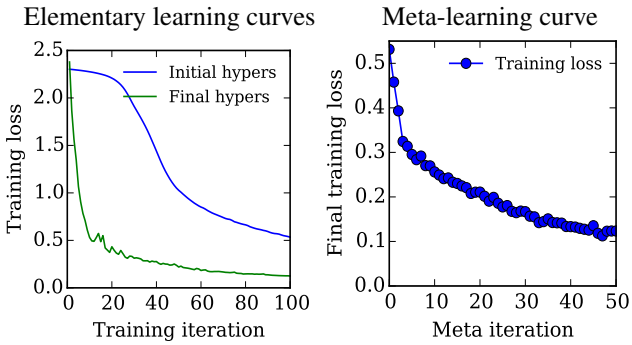

Figure 3. Elementary and meta-learning curves. The meta-learning curve shows the training loss at the end of each elementary iteration.

How smooth are hypergradients? To demonstrate that the hypergradients are smooth with respect to time steps in the training schedule, Figure 4 shows the hypergradient with respect to the step size training schedule at the beginning of training, averaged over 100 random seeds.

Optimizing weight initialization scales We optimized a separate weight initialization scale hyperparameter for each type of parameter (weights and biases) in each layer - a total of 8 hyperparameters. Results are shown in Figure 5.

Interestingly, the initialization scale chosen for the first layer weights matches a heuristic which says to choose an

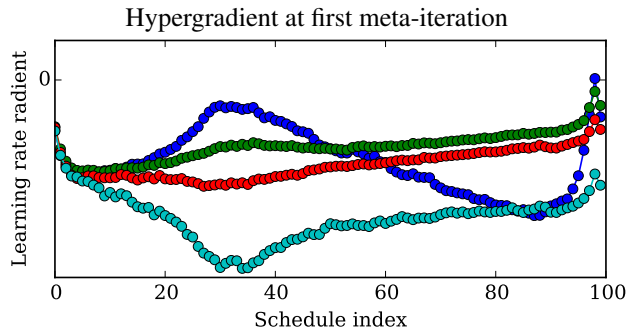

Figure 4. The initial gradient of the cross-validation loss with respect to the training schedule, averaged over 100 random weight initializations and mini batches. Colors correspond to the same layers as in Figure 2.

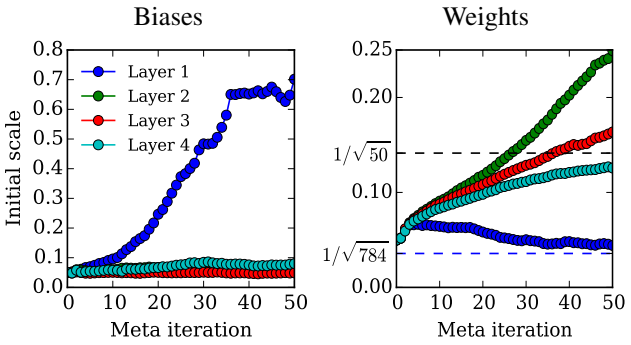

Figure 5. Meta-learning curves for the initialization scales of each layer in a 4-layer deep neural network. *Left:* Initialization scales for biases. *Right:* Initialization scales for weights. Dashed lines show a heuristic which gives an average total activation of 1. For the first layer it is $(1/\sqrt{784})$ and for subsequent layers $(1/\sqrt{50})$.

initialization scale of $1/\sqrt{N}$, where N is the number of weights in the layer.

3.2. Optimizing regularization parameters

Regularization is often important for generalization performance. Typically, a single parameter controls a single L_2 norm or sparsity penalty on the entire parameter vector of a neural network. Because different types of parameters in different layers play different roles, it is reasonable to suspect that separate regularization hyperparameter for each parameter type would improve performance. Indeed, Snoek et al. (2012) optimized separate regularization parameters for each layer in a neural network, and found that it improved performance.

We can take this idea even further, and introduce a separate regularization penalty for each individual parameter in a neural network. We use a simple model as an example – logistic regression, which can be seen as a neural network without a hidden layer. We choose this model because every weight corresponds to an input-pixel and output-label pair, meaning that these 7,840 hyperparameters might be relatively interpretable. Figure 6 shows a set of regularization hyperparameters learned for a logistic regression network. Because each parameter corresponds to a particular input, this regularization scheme could be seen as a generalization of automatic relevance determination (MacKay & Neal, 1994).

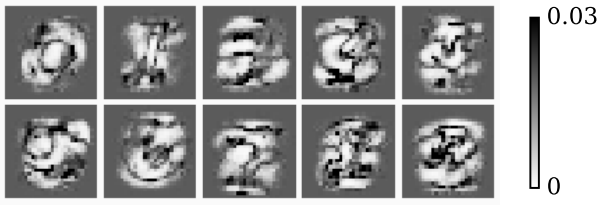

Figure 6. Optimized L_2 regularization hyperparameters for each weight in a logistic regression trained on MNIST. The weights corresponding to each output label (0 through 9 respectively) have been rendered separately. High values (black) indicate strong regularization.

3.3. Optimizing training data

We can use Algorithm 2 to take the gradient with respect to any parameter the training procedure depends on. This includes the training data, which can be viewed as just another set of hyperparameters. By chaining gradients through transformations of the data, we can compute gradients of the validation objective with respect to data preprocessing, weighting, or augmentation procedures.

We demonstrate a simple proof-of-concept where an *entire training set* is learned by gradient descent, starting from blank images. Figure 7 shows a training set, the pixels of which were optimized to improve performance on a validation set of 10,000 examples from MNIST. We optimized 10 training examples, each having a different fixed label, again from 0 to 9 respectively. Learning the labels of a larger training set might shed light on which classes are difficult to distinguish and so require more examples.

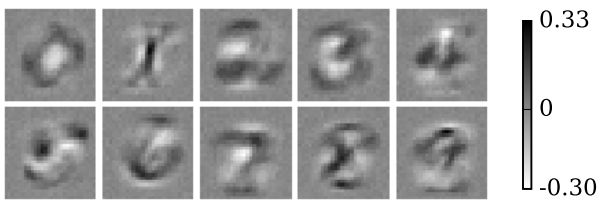

Figure 7. A dataset generated purely through meta-learning. Each pixel is treated as a hyperparameter, which are all optimized to maximize validation-set performance. Training labels are fixed in order from 0 to 9. Some optimal pixel values are negative.

3.4. Optimizing initial parameters

The last remaining parameter to SGD is the initial parameter vector. Treating this vector as a hyperparameter blurs the distinction between learning and meta-learning. In the extreme case where all elementary learning rates are set to zero, the training set ceases to matter and the meta-learning procedure exactly reduces to elementary learning on the validation set. Due to philosophical vertigo, we chose not to optimize the initial parameter vector.

3.5. Learning continuously parameterized architetures

Many of the notable successes in deep learning have come from novel architectures adapted to particular domains: convolutional neural nets, recurrent neural nets and multitask neural nets. We can think of these architectures as hard constraints that force particular weights to be zero and tie particular pairs of weights together. By softening these hard architectural constraints we can form continuous (but very high-dimensional) parameterizations of architecture. Having access to hypergradients makes learning these softened architectures feasible.

We illustrate this "architecture learning" with a multitask learning problem, the Omniglot data set (Lake, 2014). This data set consists of 28x28 pixel greyscale images of characters from 50 alphabets with up to 55 characters in each alphabet but only 15 examples of each character. Rather than learning a separate neural net for each alphabet, a multitask approach would be for all the neural nets to share a single first layer, pooling statistical strength to learn generic Gabor-like filters, while maintaining separate higher layers specific to each alphabet.

We can parameterize any architecture based on weight tying or weight absence with a pairwise quadratic penalty on the weights, $\mathbf{w}^T A \mathbf{w}$, where A is a number-of-weights by number-of-weights matrix. Learning this enormous matrix is clearly infeasible but we can implicitly build such a matrix from lower dimensional structures of manageable size.

For the Omniglot problem, we learn a penalty for each alphabet pair, separately for each neural net layer. Thus, for ten three-layer neural networks, the penalty matrix A is fully described by three ten-by-ten matrices. An architecture with fully independent nets for each alphabet corresponds to three diagonal matrices while an architecture with a mutual lower layer corresponds to two diagonal matrices for the upper layers and a matrix of all ones for the lowest layer (Figure 9).

We use five alphabets from the Omniglot set. To see whether our multitask learning system is able to learn high level similarities as well as low-level similarities, we repeat these five alphabets with the images rotated by 90 degrees (Figure 8) to make ten alphabets total.

Figure 8. Top: Example characters from 5 alphabets taken from the Omniglot dataset. *Bottom:* Those same alphabets with each character rotated by 90°. Distinguishing characters within each of these 10 alphabets constitute the 10 tasks in our multi-task learning experiment.

Figure 9 shows the learned penalties (normalized by row and column to have ones on the diagonal, akin to a correlation matrix). We see that the lowest layer has been partially shared, across all alphabets equally, with the upper layers much less shared. Interestingly, the top layer penalty learns to share weights between the rotated alphabets.

	Input weights	Middle weights	Output weights	Train error	Test error
Separate networks				0.61	1.34
Tied weights				0.90	1.25
Learned sharing				0.60	**1.13**

Figure 9. Results of the Omniglot multitask experiment. Each matrix shows the degree of weight sharing between each pair of tasks for that layer. *Top*: A separate network is trained independently for each task. *Middle*: The lowest-level features were forced to be shared. *Bottom*: The degree of weight sharing between tasks was optimized by hyperparameter optimization.

3.6. Implementation Details

Automatic differentiation (AD) software packages such as Theano (Bastien et al., 2012; Bergstra et al., 2010) are mainstays of deep learning, significantly speeding up development time by providing gradients automatically. Since we required access to the internal logic of RMD in order to implement Algorithm 2, we implemented our own automatic differentiation package for Python, available at

github.com/HIPS/autograd. This package differentiates standard Numpy (Oliphant, 2007) code, and can differentiate code containing while loops, branches, and even gradient evaluations.

Code for all experiments in this paper is available at github.com/HIPS/hypergrad.

4. Limitations

Back-propagation for training neural networks has several pitfalls that were later addressed by analysis and engineering. Likewise, the use of hypergradients also has several apparent difficulties that need to be addressed before it becomes practical. This section explores several issues with this technique that became apparent in our experiments.

When are gradients meaningful? Bengio et al. (1994) noted that "learning long-term dependencies with gradient descent is difficult." Our situation is even worse: We are using gradients to optimize functions which depend on their hyperparameters through hundreds of iterations of SGD. To make things worse, each elementary iteration's gradient itself depends on forward- and then back-propagation through a neural network. Thus the same issues that sometimes make elementary learning difficult are compounded.

For example, Pearlmutter (1996, Chapter 4) showed that large learning rates induce chaotic behavior in the learning dynamics, making the gradient uninformative about the medium-term shape of the training objective. This phenomenon is related to the exploding-gradient problem (Pascanu et al., 2012).

Figure 10 illustrates this phenomenon when training a neural network having 2 hidden layers for 50 elementary iterations. We partially addressed this problem in our experiments by initializing learning rates to be relatively small, and stopping meta-optimization when the magnitude of the meta-gradient began to grow.

Overfitting How many hyperparameters can we fruitfully optimize? One limitation is overfitting the validation objective, in the same way that optimizing too many parameters can overfit the training objective. However, the same rules of thumb still apply – the size of the validation set, assuming examples are i.i.d., gives a rough guide to how many hyperparameters can be optimized.

Discrete parameters Of course, gradients are not necessarily useful for optimizing discrete hyperparameters such as the number of layers, or hyperparameters that affect discrete changes such as dropout regularization parameters. Some of these difficulties could be addressed by parameterizing apparently discrete choices in a continuous manner. For instance, the per-hidden-unit regularization of section 3.2 is an example of a continuous way to choose the number of hidden units.

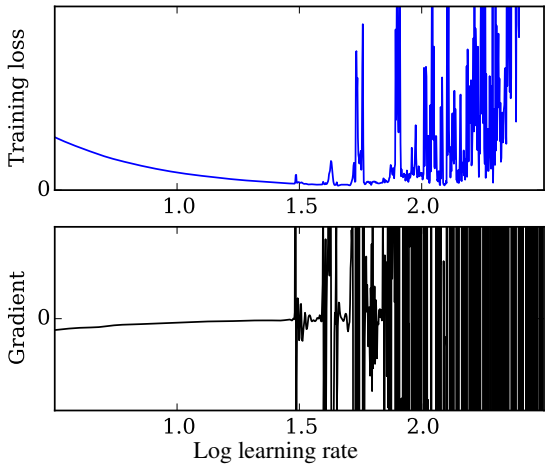

Figure 10. Top: Loss after training as a function of learning rate. *Bottom:* Gradient of loss with respect to learning rate. When the learning rate is high, the gradient becomes uninformative about the medium-term behavior of the function. To maintain stability during meta-learning, we initialize using a small learning rate so as to approach the minimum from the left.

5. Related work

The most closely-related work is Domke (2012), who derived algorithms to compute reverse-mode derivatives of gradient descent with momentum and L-BFGS, using them to update the hyperparameters of CRF image models. However, his approach relied on naïve caching of all parameter vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_T$, making it impractical for large models with many training iterations.

Larsen et al. (1998), Eigenmann & Nossek (1999), Chen & Hagan (1999), Bengio (2000), Abdel-Gawad & Ratner (2007), and Foo et al. (2008) showed that gradients of regularization parameters are available in closed form when training has converged exactly to a local minimum. In contrast, our procedure can compute exact gradients of any type of hyperparameter, whether or not learning has converged.

Support vector machines Chapelle et al. (2002) introduced a differentiable bound on the SVM loss in order to be able to compute derivatives with respect to hundreds of hyperparameters, including weighting parameters for each input dimension in the kernel. However, this bound was not tight, since optimizing the SVM objective requires a discrete selection of training points.

Bayesian methods For Bayesian models with a closed-form marginal likelihood, gradients with respect to all continuous hyperparameters are usually available. For example, this ability has been used to construct complex kernels for Gaussian process models (Rasmussen & Williams, 2006, Chapter 5). Variational inference also allows gradient-based tuning of hyperparameters in Bayesian neural-network models such as deep Gaussian processes (Hensman & Lawrence, 2014). However, it does not provide gradients with respect to training parameters.

Gradients with respect to Markov chain parameters Salimans et al. (2014) tune the step-size and mass-matrix parameters of Hamiltonian Monte Carlo by chaining gradients from a lower bound on the marginal likelihood through several iterations of leapfrog dynamics. Because they used only a small number of steps, all intermediate values could be stored naïvely. Our reversible-dynamics memory-tape approach could be used to dramatically extend the number of HMC iterations used in this approach.

6. Extensions and future work

Bayesian optimization with gradients Hypergradients could be used with parallel, model-based optimization of hyperparameters. For example, Gaussian-process-based optimization methods could incorporate gradient information (Solak et al., 2003). Such methods could make use of parallel evaluations of hypergradients, which might be too slow to evaluate in a sequential manner.

Reversible elementary computation Recurrent neural network models can require so much memory to differentiate that checkpointing is required simply to compute their elementary gradients (Martens & Sutskever, 2012). Reversible computation might offer memory savings for some architectures. For example, evaluations of Long Short-Term Memory (Hochreiter & Schmidhuber, 1997) or a Neural Turing Machines (Graves et al., 2014) rely on long chains of mostly-small updates of parameters. Exactly reversing these dynamics might allow more memory-efficient elementary gradient evaluations of their outputs on very long input sequences.

Exactly reversing other learning methods The memory saving trick from Section 2.3 could presumably be applied to other momentum-based variants of SGD such as RMSprop (Tieleman & Hinton, 2012) or Adam (Kingma & Ba, 2014).

7. Conclusion

In this paper, we derived a computationally efficient procedure for computing gradients through stochastic gradient descent with momentum. We showed how the approximate reversibility of learning dynamics can be used to drastically reduce the memory requirement for exactly back-propagating gradients through hundreds of training iterations.

We showed how these gradients allow the optimization of validation loss with respect to thousands of hyperparameters, something which was previously infeasible. This new ability allows the automatic tuning of most details of training neural networks. We demonstrated the tuning of detailed training schedules, regularization schedules, and neural network architectures.

Acknowledgments

We would like to thank Christian Steinruecken, Oren Rippel, and Matthew James Johnson for helpful discussions. We also thank Brenden Lake for graciously providing the Omniglot dataset. Thanks to Jason Rolfe for helpful feedback. We thank Analog Devices International and Samsung Advanced Institute of Technology for their support.

Appendix: Forward vs. reverse-mode differentiation

By the chain rule, the gradient of a set of nested functions is given by the product of the individual derivatives of each function:

$$\frac{\partial f_4(f_3(f_2(f_1(x))))}{\partial x} = \frac{\partial f_4}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x}$$

If each function has multivariate inputs and outputs, the gradients are Jacobian matrices.

Forward and reverse mode differentiation differ only by the order in which they evaluate this product. Forward-mode differentiation works by multiplying gradients in the same order as the functions are evaluated:

$$\frac{\partial f_4(f_3(f_2(f_1(x))))}{\partial x} = \frac{\partial f_4}{\partial f_3} \cdot \left(\frac{\partial f_3}{\partial f_2} \cdot \left(\frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} \right) \right)$$

Reverse-mode multiplies the gradients in the opposite order, starting from the final result:

$$\frac{\partial f_4(f_3(f_2(f_1(x))))}{\partial x} = \left(\left(\frac{\partial f_4}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \right) \cdot \frac{\partial f_2}{\partial f_1} \right) \cdot \frac{\partial f_1}{\partial x}$$

In an optimization setting, the final result of the nested functions, f_4, is a scalar, while the input x and intermediate values, $f_1 - f_3$, can be vectors. In this scenario the advantage of reverse-mode differentiation is very clear. Let's imagine that the dimensionality of all the intermediate vectors is D. In reverse mode, we start from the (scalar) output, and multiply by the next $D \times D$ Jacobian at each step. The

value we accumulate is just a D-dimensional vector. In forward mode, however, we must accumulate an entire $D \times D$ matrix at each step. But do we have still have to compute and instantiate the $D \times D$ Jacobian matrices themselves either way? In general, yes. But in the (common) case that the vector-to-vector functions are either elementwise operations or (reshaped) matrix multiplications, the Jacobian matrices can actually be very sparse, and multiplication by the Jacobian can be performed efficiently without instantiation (Pearlmutter & Siskind, 2008).

The main drawback of reverse-mode differentiation is that intermediate values must be maintained in memory during the forward pass. In sections 2.1 and 2.3, we show how to drastically reduce the memory requirements of reverse-mode differentiation when differentiating through the entire learning procedure.

References

Abdel-Gawad, Ahmed and Ratner, Simon. Adaptive optimization of hyperparameters in L2-regularised logistic regression. Technical report, 2007.

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

Baydin, A. G. and Pearlmutter, B. A. Automatic differentiation of algorithms for machine learning. In *Proceedings of the AutoML Workshop at the International Conference on Machine Learning (ICML)*, 2014.

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

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5 (2):157–166, 1994.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

Bergstra, James, Bardenet, Rémi, Bengio, Yoshua, Kégl, Balázs, et al. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, 2011.

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

Chapelle, Olivier, Vapnik, Vladimir, Bousquet, Olivier, and Mukherjee, Sayan. Choosing multiple parameters for support vector machines. *Machine learning*, 46(1-3): 131–159, 2002.

Chen, Dingding and Hagan, Martin T. Optimal use of regularization and cross-validation in neural network modeling. In *International Joint Conference on Neural Networks*, volume 2, pp. 1275–1280. IEEE, 1999.

Courbariaux, Matthieu, Bengio, Yoshua, and David, Jean-Pierre. Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

Dahl, George E, Jaitly, Navdeep, and Salakhutdinov, Ruslan. Multi-task neural networks for QSAR predictions. *arXiv preprint arXiv:1406.1231*, 2014.

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

Eigenmann, Robert and Nossek, Josef A. Gradient based adaptive regularization. In *Proceedings of the 1999 IEEE Signal Processing Society Workshop on Neural Networks*, pp. 87–94. IEEE, 1999.

Erhan, Dumitru, Bengio, Yoshua, Courville, Aaron, Manzagol, Pierre-Antoine, Vincent, Pascal, and Bengio, Samy. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

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

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Hensman, James and Lawrence, Neil D. Nested variational compression in deep Gaussian processes. *arXiv preprint arXiv:1412.1370*, 2014.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hut, P., Makino, J., and McMillan, S. Building a better leapfrog. *Astrophysical Journal, Part 2 - Letters*, 443: L93–L96, April 1995. doi: 10.1086/187844.

Hutter, Frank, Hoos, Holger H, and Leyton-Brown, Kevin. Sequential model-based optimization for general algorithm configuration. In *Proceedings of LION-5*, volume 6683, pp. 507–523. Springer, 2011.

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

Lake, Brenden M. *Towards more human-like concept learning in machines: Compositionality, causality, and learning-to-learn*. PhD thesis, Massachusetts Institute of Technology, 2014.

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

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

MacKay, David J.C. *Information theory, inference, and learning algorithms*. Cambridge University Press, 2003.

MacKay, David J.C. and Neal, Radford M. Automatic relevance determination for neural networks. In *Technical Report*. Cambridge University, 1994.

Martens, James and Sutskever, Ilya. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*, pp. 479–535. Springer, 2012.

Oliphant, Travis E. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. Understanding the exploding gradient problem. *arXiv preprint arXiv:1211.5063*, 2012.

Pearlmutter, Barak. *An investigation of the gradient descent process in neural networks*. PhD thesis, Carnegie Mellon University, 1996.

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

Pearlmutter, Barak A. and Siskind, Jeffrey Mark. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

Rasmussen, Carl E. and Williams, Christopher K.I. *Gaussian Processes for Machine Learning*. The MIT Press, Cambridge, MA, USA, 2006.

Salimans, Tim, Kingma, Diederik P., and Welling, Max. Markov chain Monte Carlo and variational inference: Bridging the gap. *arXiv preprint arXiv:1410.6460*, 2014.

Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, pp. 2960–2968, 2012.

Solak, E., Murray Smith, R., Leithead, W.E., Leith, D., and Rasmussen, Carl E. Derivative observations in Gaussian process models of dynamic systems. *Advances in Neural Information Processing Systems*, pp. 1057–1064, 2003.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1139–1147, 2013.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pp. 3104–3112. Curran Associates, Inc., 2014.

Tieleman, T. and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning, 2012.