Forward and Reverse Gradient-Based Hyperparameter Optimization

Luca Franceschi [1 2] **Michele Donini** [1] **Paolo Frasconi** [3] **Massimiliano Pontil** [1 2]

Abstract

We study two procedures (reverse-mode and forward-mode) for computing the gradient of the validation error with respect to the hyperparameters of any iterative learning algorithm such as stochastic gradient descent. These procedures mirror two methods of computing gradients for recurrent neural networks and have different trade-offs in terms of running time and space requirements. Our formulation of the reverse-mode procedure is linked to previous work by Maclaurin et al. (2015) but does not require reversible dynamics. The forward-mode procedure is suitable for real-time hyperparameter updates, which may significantly speed up hyperparameter optimization on large datasets. We present experiments on data cleaning and on learning task interactions. We also present one large-scale experiment where the use of previous gradient-based methods would be prohibitive.

1. Introduction

The increasing complexity of machine learning algorithms has driven a large amount of research in the area of hyperparameter optimization (HO) — see, e.g., (Hutter et al., 2015) for a review. The core idea is relatively simple: given a measure of interest (e.g. the misclassification error) HO methods use a validation set to construct a *response function* of the hyperparameters (such as the average loss on the validation set) and explore the hyperparameter space to seek an optimum.

Early approaches based on grid search quickly become impractical as the number of hyperparameters grows and are even outperformed by random search (Bergstra & Bengio, 2012). Given the high computational cost of evaluating the response function, Bayesian optimization approaches provide a natural framework and have been extensively studied in this context (Snoek et al., 2012; Swersky et al., 2013; Snoek et al., 2015). Related and faster sequential model-based optimization methods have been proposed using random forests (Hutter et al., 2011) and tree Parzen estimators (Bergstra et al., 2011), scaling up to a few hundreds of hyperparameters (Bergstra et al., 2013).

In this paper, we follow an alternative direction, where gradient-based algorithms are used to optimize the performance on a validation set with respect to the hyperparameters (Bengio, 2000; Larsen et al., 1996). In this setting, the validation error should be evaluated at a minimizer of the training objective. However, in many current learning systems such as deep learning, the minimizer is only approximate. Domke (2012) specifically considered running an iterative algorithm, like gradient descent or momentum, for a given number of steps, and subsequently computing the gradient of the validation error by a back-optimization algorithm. Maclaurin et al. (2015) considered reverse-mode differentiation of the response function. They suggested the idea of reversing parameter updates to achieve space efficiency, proposing an approximation capable of addressing the associated loss of information due to finite precision arithmetics. Pedregosa (2016) proposed the use of inexact gradients, allowing hyperparameters to be updated before reaching the minimizer of the training objective. Both (Maclaurin et al., 2015) and (Pedregosa, 2016) managed to optimize a number of hyperparameters in the order of one thousand.

In this paper, we illustrate two alternative approaches to compute the hypergradient (i.e., the gradient of the response function), which have different trade-offs in terms of running time and space requirements. One approach is based on a Lagrangian formulation associated with the parameter optimization dynamics. It encompasses the reverse-mode differentiation (RMD) approach used by Maclaurin et al. (2015), where the dynamics corresponds to stochastic gradient descent with momentum. We do not assume reversible parameter optimization dynamics. A well-known drawback of RMD is its space complexity: we need to store the whole trajectory of training iterates in order to compute the hypergradient. An alternative approach that we consider overcomes this problem by computing

[1]Computational Statistics and Machine Learning, Istituto Italiano di Tecnologia, Genoa, Italy [2]Department of Computer Science, University College London, UK [3]Department of Information Engineering, Università degli Studi di Firenze, Italy. Correspondence to: Luca Franceschi <luca.franceschi@iit.it>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

the hypergradient in forward-mode and it is efficient when the number of hyperparameters is much smaller than the number of parameters. To the best of our knowledge, the forward-mode has not been studied before in this context.

As we shall see, these two approaches have a direct correspondence to two classic alternative ways of computing gradients for recurrent neural networks (RNN) (Pearlmutter, 1995): the Lagrangian (reverse) way corresponds to back-propagation through time (Werbos, 1990), while the forward way corresponds to real-time recurrent learning (RTRL) (Williams & Zipser, 1989). As RTRL allows one to update parameters after each time step, the forward approach is suitable for real-time hyperparameter updates, which may significantly speed up the overall hyperparameter optimization procedure in the presence of large datasets. We give experimental evidence that the real-time approach is efficient enough to allow for the automatic tuning of crucial hyperparameters in a deep learning model. In our experiments, we also explore constrained hyperparameter optimization, showing that it can be used effectively to detect noisy examples and to discover the relationships between different learning tasks.

The paper is organized in the following manner. In Section 2 we introduce the problem under study. In Section 3.1 we derive the reverse-mode computation. In Section 3.2 we present the forward-mode computation of the hypergradient, and in Section 3.3 we introduce the idea of real-time hyperparameter updates. In Section 4 we discuss the time and space complexity of these methods. In Section 5 we present empirical results with both algorithms. Finally in Section 6 we discuss our findings and highlight directions of future research.

2. Hyperparameter Optimization

We focus on training procedures based on the optimization of an objective function $J(w)$ with respect to w (e.g. the regularized average training loss for a neural network with weights w). We see the training procedure by stochastic gradient descent (or one of its variants like momentum, RMSProp, Adam, etc.) as a dynamical system with a state $s_t \in \mathbb{R}^d$ that collects weights and possibly accessory variables such as velocities and accumulated squared gradients. The dynamics are defined by the system of equations

$$s_t = \Phi_t(s_{t-1}, \lambda) \;\; t = 1, \ldots, T \tag{1}$$

where T is the number of iterations, s_0 contains initial weights and initial accessory variables, and, for every $t \in \{1, \ldots, T\}$,

$$\Phi_t : (\mathbb{R}^d \times \mathbb{R}^m) \to \mathbb{R}^d$$

is a smooth mapping that represents the operation performed by the t-th step of the optimization algorithm (i.e. on mini-batch t). Finally, $\lambda \in \mathbb{R}^m$ is the vector of hyperparameters that we wish to tune.

As simple example of these dynamics occurs when training a neural network by gradient descent with momentum (GDM), in which case $s_t = (v_t, w_t)$ and

$$\begin{array}{rcl} v_t & = & \mu v_{t-1} + \nabla J_t(w_{t-1}) \\ w_t & = & w_{t-1} - \eta(\mu v_{t-1} - \nabla J_t(w_{t-1})) \end{array} \tag{2}$$

where J_t is the objective associated with the t-th mini-batch, μ is the rate and η is the momentum. In this example, $\lambda = (\mu, \eta)$.

Note that the iterates $s_1, \ldots, s_T$ implicitly depend on the vector of hyperparameters λ. Our goal is to optimize the hyperparameters according to a certain error function E evaluated at the last iterate s_T. Specifically, we wish to solve the problem

$$\min_{\lambda \in \Lambda} f(\lambda) \tag{3}$$

where the set $\Lambda \subset \mathbb{R}^d$ incorporates constraints on the hyperparameters, and the response function $f : \mathbb{R}^m \to \mathbb{R}$ is defined at $\lambda \in \mathbb{R}^m$ as

$$f(\lambda) = E(s_T(\lambda)). \tag{4}$$

We highlight the generality of the framework. The vector of hyperparameters λ may include components associated with the training objective, and components associated with the iterative algorithm. For example, the training objective may depend on hyperparameters used to design the loss function as well as multiple regularization parameters. Yet other components of λ may be associated with the space of functions used to fit the training objective (e.g. number of layers and weights of a neural network, parameters associated with the kernel function used within a kernel based method, etc.). The validation error E can in turn be of different kinds. The simplest example is to choose E as the average of a loss function over a validation set. We may however consider multiple validation objectives, in that the hyperparameters associated with the iterative algorithm (μ and γ in the case of momentum mentioned above) may be optimized using the training set, whereas the regularization parameters would typically require a validation set, which is distinct from the training set (in order to avoid over-fitting).

3. Hypergradient Computation

In this section, we review the reverse-mode computation of the gradient of the response function (or hypergradient) under a Lagrangian perspective and introduce a forward-mode strategy. These procedures correspond to the reverse-mode and the forward-mode algorithmic differentiation schemes (Griewank & Walther, 2008). We finally introduce a real-time version of the forward-mode procedure.

Algorithm 1 REVERSE-HG

Input: λ current values of the hyperparameters, s_0 initial optimization state
Output: Gradient of validation error w.r.t. λ
for $t = 1$ **to** T **do**
  $s_t = \Phi_t(s_{t-1}, \lambda)$
end for
$\alpha_T = \nabla E(s_T)$
$g = 0$
for $t = T-1$ **downto** 1 **do**
  $g = g + \alpha_{t+1} B_{t+1}$
  $\alpha_t = \alpha_{t+1} A_{t+1}$
end for
return g

Algorithm 2 FORWARD-HG

Input: λ current values of the hyperparameters, s_0 initial optimization state
Output: Gradient of validation error w.r.t. λ
$Z_0 = 0$
for $t = 1$ **to** T **do**
  $s_t = \Phi_t(s_{t-1}, \lambda)$
  $Z_t = A_t Z_{t-1} + B_t$
end for
return $\nabla E(s) Z_T$

3.1. Reverse-Mode

The reverse-mode computation leads to an algorithm closely related to the one presented in (Maclaurin et al., 2015). A major difference with respect to their work is that we do not require the mappings Φ_t defined in Eq. (1) to be invertible. We also note that the reverse-mode calculation is structurally identical to back-propagation through time (Werbos, 1990).

We start by reformulating problem (3) as the constrained optimization problem

$$\begin{array}{ll} \min\limits_{\lambda, s_1, \ldots, s_T} & E(s_T) \\ \text{subject to} & s_t = \Phi_t(s_{t-1}, \lambda),\ t \in \{1, \ldots, T\}. \end{array} \tag{5}$$

This formulation closely follows a classical Lagrangian approach used to derive the back-propagation algorithm (LeCun, 1988). Furthermore, the framework naturally allows one to incorporate constraints on the hyperparameters.

The Lagrangian of problem (5) is

$$\mathcal{L}(s, \lambda, \alpha) = E(s_T) + \sum_{t=1}^{T} \alpha_t (\Phi_t(s_{t-1}, \lambda) - s_t) \tag{6}$$

where, for each $t \in \{1, \ldots, T\}$, $\alpha_t \in \mathbb{R}^d$ is a row vector of Lagrange multipliers associated with the t-th step of the dynamics.

The partial derivatives of the Lagrangian are given by

$$\frac{\partial \mathcal{L}}{\partial \alpha_t} = \Phi_t(s_{t-1}, \lambda) - s_t, \quad t \in \{1, \ldots, T\} \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial s_t} = \alpha_{t+1} A_{t+1} - \alpha_t, \quad t \in \{1, \ldots, T-1\} \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial s_T} = \nabla E(s_T) - \alpha_T \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \sum_{t=1}^{T} \alpha_t B_t, \tag{10}$$

where for every $t \in \{1, \ldots, T\}$, we define the matrices

$$A_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial s_{t-1}}, \quad B_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda}. \tag{11}$$

Note that $A_t \in \mathbb{R}^{d \times d}$ and $B_t \in \mathbb{R}^{d \times m}$.

The optimality conditions are then obtained by setting each derivative to zero. In particular, setting the right hand side of Equations (8) and (9) to zero gives

$$\alpha_t = \begin{cases} \nabla E(s_T) & \text{if } t = T, \\ \nabla E(s_T) A_T \cdots A_{t+1} & \text{if } t \in \{1, \ldots, T-1\}. \end{cases}$$

Combining these equations with Eq. (10) we obtain that

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \nabla E(s_T) \sum_{t=1}^{T} \left(\prod_{s=t+1}^{T} A_s \right) B_t.$$

As we shall see this coincides with the expression for the gradient of f in Eq. (15) derived in the next section. Pseudo-code of REVERSE-HG is presented in Algorithm 1.

3.2. Forward-Mode

The second approach to compute the hypergradient appeals to the chain rule for the derivative of composite functions, to obtain that the gradient of f at λ satisfies[1]

$$\nabla f(\lambda) = \nabla E(s_T) \frac{ds_T}{d\lambda} \tag{12}$$

where $\frac{ds_T}{d\lambda}$ is the $d \times m$ matrix formed by the total derivative of the components of s_T (regarded as rows) with respect to the components of λ (regarded as columns).

Recall that $s_t = \Phi_t(s_{t-1}, \lambda)$. The operators Φ_t depends on the hyperparameter λ both directly by its expression and indirectly through the state s_{t-1}. Using again the chain

[1] Remember that the gradient of a scalar function is a row vector.

rule we have, for every $t \in \{1, \dots, T\}$, that

$$\frac{ds_t}{d\lambda} = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial s_{t-1}} \frac{ds_{t-1}}{d\lambda} + \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda}. \quad (13)$$

Defining $Z_t = \frac{ds_t}{d\lambda}$ for every $t \in \{1, \dots, T\}$ and recalling Eq. (11), we can rewrite Eq. (13) as the recursion

$$Z_t = A_t Z_{t-1} + B_t, \quad t \in \{1, \dots, T\}. \quad (14)$$

Using Eq. (14), we obtain that

$$\begin{aligned} \nabla f(\lambda) &= \nabla E(s_T) Z_T \\ &= \nabla E(s_T)(A_T Z_{T-1} + B_T) \\ &= \nabla E(s_T)(A_T A_{T-1} Z_{T-2} + A_T B_{T-1} + B_T) \\ &\vdots \\ &= \nabla E(s_T) \sum_{t=1}^{T} \left(\prod_{s=t+1}^{T} A_s \right) B_t. \end{aligned} \quad (15)$$

Note that the recurrence (14) on the Jacobian matrix is structurally identical to the recurrence in the RTRL procedure described in (Williams & Zipser, 1989, eq. (2.10)).

From the above derivation it is apparent that $\nabla f(\lambda)$ can be computed by an iterative algorithm which runs in parallel to the training algorithm. Pseudo-code of FORWARD-HG is presented in Algorithm 2. At first sight, the computation of the terms in the right hand side of Eq. (14) seems prohibitive. However, in Section 4 we observe that if m is much smaller than d, the computation can be done efficiently.

3.3. Real-Time Forward-Mode

For every $t \in \{1, \dots, T\}$ let $f_t : \mathbb{R}^m \to \mathbb{R}$ be the response function at time t: $f_t(\lambda) = E(s_t(\lambda))$. Note that f_T coincides with the definition of the response function in Eq. (4). A major difference between REVERSE-HG and FORWARD-HG is that the *partial* hypergradients

$$\nabla f_t(\lambda) = \frac{dE(s_t)}{d\lambda} = \nabla E(s_t) Z_t \quad (16)$$

are available in the second procedure at each time step t and not only at the end.

The availability of partial hypergradients is significant since we are allowed to update hyperparameters several times in a single optimization epoch, without having to wait until time T. This is reminiscent of the real-time updates suggested by Williams & Zipser (1989) for RTRL. The real-time approach may be suitable in the case of a data stream (i.e. $T = \infty$), where REVERSE-HG would be hardly applicable. Even in the case of finite (but large) datasets it is possible to perform one hyperparameter update after a hyper-batch of data (i.e. a set of minibatches) has been processed. Algorithm 2 can be easily modified to yield a partial hypergradient when $t \mod \Delta = 0$ (for some hyper-batch size Δ) and letting t run from 1 to ∞, reusing examples in a circular or random way. We use this strategy in the phone recognition experiment reported in Section 5.3.

4. Complexity Analysis

We discuss the time and space complexity of Algorithms 1 and 2. We begin by recalling some basic results from the algorithmic differentiation (AD) literature.

Let $F : \mathbb{R}^n \mapsto \mathbb{R}^p$ be a differentiable function and suppose it can be evaluated in time $c(n, p)$ and requires space $s(n, p)$. Denote by J_F the $p \times n$ Jacobian matrix of F. Then the following facts hold true (Griewank & Walther, 2008) (see also Baydin et al. (2015) for a shorter account):

(i) For any vector $r \in \mathbb{R}^n$, the product $J_F r$ can be evaluated in time $O(c(n, p))$ and requires space $O(s(n, p))$ using forward-mode AD.

(ii) For any vector $q \in \mathbb{R}^p$, the product $J_F^\intercal q$ has time and space complexities $O(c(n, p))$ using reverse-mode AD.

(iii) As a corollary of item (i), the whole J_F can be computed in time $O(nc(n, p))$ and requires space $O(s(n, p))$ using forward-mode AD (just use unitary vectors $r = e_i$ for $i = 1, \dots, n$).

(iv) Similarly, J_F can be computed in time $O(pc(n, p))$ and requires space $O(c(n, p))$ using reverse-mode AD.

Let $g(d, m)$ and $h(d, m)$ denote time and space, respectively, required to evaluate the update map Φ_t defined by Eq. (1). Then the response function $f : \mathbb{R}^m \mapsto \mathbb{R}$ defined in Eq. (3) can be evaluated in time $O(Tg(d, m))$ (assuming the time required to compute the validation error $E(\lambda)$ does not affect the bound[2]) and requires space $O(h(d, m))$ since variables s_t may be overwritten at each iteration. Then, a direct application of Fact (i) above shows that Algorithm 2 runs in time $O(Tmg(d, m))$ and space $O(h(d, m))$. The same results can also be obtained by noting that in Algorithm 2 the product $A_t Z_{t-1}$ requires m Jacobian-vector products, each costing $O(g(d, m))$ (from Fact (i)), while computing the Jacobian B_t takes time $O(mg(d, m))$ (from Fact (iii)).

Similarly, a direct application of Fact (ii) shows that Algorithm 1 has both time and space complexities $O(Tg(d, m))$. Again the same results can be obtained by

[2]This is indeed realistic since the number of validation examples is typically lower than the number of training iterations.

noting that $\alpha_{t+1}A_{t_1}$ and $\alpha_t B_t$ are transposed-Jacobian-vector products that in reverse-mode take both time $O(g(d,m))$ (from Fact (ii)). Unfortunately in this case variables s_t cannot be overwritten, explaining the much higher space requirement.

As an example, consider training a neural network with k weights[3], using classic iterative optimization algorithms such as SGD (possibly with momentum) or Adam, where the hyperparameters are just learning rate and momentum terms. In this case, $d = O(k)$ and $m = O(1)$. Moreover, $g(d,m)$ and $h(d,m)$ are both $O(k)$. As a result, Algorithm 1 runs in time and space $O(Tk)$, while Algorithm 2 runs in time $O(Tk)$ and space $O(k)$, which would typically make a dramatic difference in terms of memory requirements.

5. Experiments

In this section, we present numerical simulations with the proposed methods. All algorithms were implemented in TensorFlow and the software package used to reproduce our experiments is available at `https://github.com/lucfra/RFHO`. In all the experiments, hypergradients were used inside the Adam algorithm (Kingma & Ba, 2014) in order to minimize the response function.

5.1. Data Hyper-cleaning

The goal of this experiment is to highlight one potential advantage of constraints on the hyperparameters. Suppose we have a dataset with label noise and due to time or resource constraints we can only afford to cleanup (by checking and correcting the labels) a subset of the available data. Then we may use the cleaned data as the validation set, the rest as the training set, and assign one hyperparameter to each training example. By putting a sparsity constraint on the vector of hyperparameters λ, we hope to bring to zero the influence of noisy examples, in order to generate a better model. While this is the same kind of data sparsity observed in support vector machines (SVM), our setting aims to get rid of erroneously labeled examples, in contrast to SVM which puts zero weight on redundant examples. Although this experimental setup does not necessarily reflect a realistic scenario, it aims to test the ability of our HO method to effectively make use of constraints on the hyperparameters[4]

We instantiated the above setting with a balanced subset of $N = 20000$ examples from the MNIST dataset, split into three subsets: $\mathcal{D}_{\rm tr}$ of $N_{\rm tr} = 5000$ training examples, $\mathcal{V}$ of $N_{\rm val} = 5000$ validation examples and a test set containing the remaining samples. Finally, we corrupted the labels of 2500 training examples, selecting a random subset $\mathcal{D}_f \subset \mathcal{D}_{\rm tr}$.

We considered a plain softmax regression model with parameters W (weights) and b (bias). The error of a model (W,b) on an example x was evaluated by using the cross-entropy $\ell(W,b,x)$ both in the training objective function, $E_{\rm tr}$, and in the validation one, $E_{\rm val}$. We added in $E_{\rm tr}$ an hyperparameter vector $\lambda \in [0,1]^{N_{\rm tr}}$ that weights each example in the training phase, i.e. $E_{\rm tr}(W,b) = \frac{1}{N_{\rm tr}} \sum_{i\in\mathcal{D}_{\rm tr}} \lambda_i \ell(W,b,x_i)$.

According to the general HO framework, we fit the parameters (W,b) to minimize the training loss and the hyperparameters λ to minimize the validation error. The sparsity constraint was implemented by bounding the $L1$-norm of λ, resulting in the optimization problem

$$\min_{\lambda\in\Lambda} E_{\rm val}(W_T, b_T) \qquad (P_{HO})$$

where $\Lambda = \{\lambda : \lambda \in [0,1]^{N_{\rm tr}}, \|\lambda\|_1 \leq R\}$ and (W_T, b_T) are the parameters obtained after T iterations of gradient descent on the training objective. Given the high dimensionality of λ, we solved (P_{HO}) iteratively computing the hypergradients with REVERSE-HG method and projecting Adam updates on the set Λ.

We are interested in comparing the following three test set accuracies:

- Oracle: the accuracy of the minimizer of $E_{\rm tr}$ trained on clean examples only, i.e. $(\mathcal{D}_{\rm tr}\setminus\mathcal{D}_f)\cup\mathcal{V}$; this setting is effectively taking advantage of an oracle that tells which examples have a wrong label;
- Baseline: the accuracy of the minimizer of $E_{\rm tr}$ trained on all available data $\mathcal{D}\cup\mathcal{V}$;
- DH-R: the accuracy of the data hyper-cleaner with a given value of the $L1$ radius, R. In this case, we first optimized hyperparameters and then constructed a cleaned training set $\mathcal{D}_c \subset \mathcal{D}_{\rm tr}$ (keeping examples with $\lambda_i > 0$); we finally trained on $\mathcal{D}_c \cup \mathcal{V}$.

We are also interested in evaluating the ability of the hyper-cleaner to detect noisy samples. Results are shown in Table 1. The data hyper-cleaner is robust with respect to the choice of R and is able to identify corrupted examples, recovering a model that has almost the same accuracy as a model produced with the help of an oracle.

Figure 1 shows how the accuracy of DH-1000 improves with the number of hyper-iterations and the progression of the amount of discarded examples. The data hyper-cleaner starts by discarding mainly corrupted examples, and while

[3]This includes linear SVM and logistic regression as special cases.

[4]We note that a related approach based on reinforcement learning is presented in (Fan et al., 2017).

Table 1: Test accuracies for the baseline, the oracle, and DH-R for four different values of R. The reported F_1 measure is the performance of the hyper-cleaner in correctly identifying the corrupted training examples.

	Accuracy %	F_1
Oracle	90.46	1.0000
Baseline	87.74	-
DH-1000	90.07	0.9137
DH-1500	90.06	0.9244
DH-2000	90.00	0.9211
DH-2500	90.09	0.9217

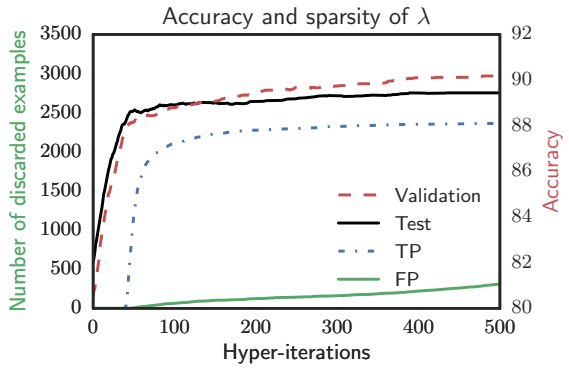

Figure 1: Right vertical axis: accuracies of DH-1000 on validation and test sets. Left vertical axis: number of discarded examples among noisy (True Positive, TP) and clean (False Positive, FP) ones.

the optimization proceeds, it begins to remove also a portion of cleaned one. Interestingly, the test set accuracy continues to improve even when some of the clean examples are discarded.

5.2. Learning Task Interactions

This second set of experiments is in the multitask learning (MTL) context, where the goal is to find simultaneously the model of multiple related tasks. Many MTL methods require that a task interaction matrix is given as input to the learning algorithm. However, in real applications, this matrix is often unknown and it is interesting to learn it from data. Below, we show that our framework can be naturally applied to learning the task relatedness matrix.

We used CIFAR-10 and CIFAR-100 (Krizhevsky & Hinton, 2009), two object recognition datasets with 10 and 100 classes, respectively. As features we employed the preactivation of the second last layer of Inception-V3 model trained on ImageNet[5]. From CIFAR-10, we extracted 50 examples as training set, different 50 examples as validation set and the remaining for testing. From CIFAR-100, we selected 300 examples as training set, 300 as validation set and the remaining for testing. Finally, we used a one-hot encoder of the labels obtaining a set of labels in $\{0,1\}^K$ ($K = 10$ or $K = 100$).

The choice of small training set sizes is due to the strong discriminative power of the selected features. In fact, using larger sample sizes would not allow to appreciate the advantage of MTL. In order to leverage information among the different classes, we employed a multitask learning (MTL) regularizer (Evgeniou et al., 2005)

$$\Omega_{C,\rho}(W) = \sum_{j,k=1}^{K} C_{j,k}\|w_j - w_k\|_2^2 + \rho \sum_{k=1}^{K} \|w_k\|^2,$$

where w_k are the weights for class k, K is the number of classes, and the symmetric non-negative matrix C models the interactions between the classes/tasks. We used a regularized training error defined as $E_{\rm tr}(W) = \sum_{i\in\mathcal{D}_{\rm tr}} \ell(Wx_i + b, y_i) + \Omega_{C,\rho}(W)$ where $\ell(\cdot,\cdot)$ is the categorical cross-entropy and $b = (b_1, \ldots, b_K)$ is the vector of thresholds associated with each linear model. We wish solve the following optimization problem:

$$\min \left\{ E_{\rm val}(W_T, b_T) \text{ subject to } \rho \geq 0,\ C = C^\intercal,\ C \geq 0 \right\},$$

where (W_T, b_T) is the T-th iteration obtained by running gradient descent with momentum (GDM) on the training objective. We solve this problem using REVERSE-HG and optimizing the hyperparameters by projecting Adam updates on the set $\{(\rho, C) : \rho \geq 0,\ C = C^\intercal,\ C \geq 0\}$. We compare the following methods:

- SLT: single task learning, i.e. $C = 0$, using a validation set to tune the optimal value of ρ for each task;
- NMTL: we considered the naive MTL scenario in which the tasks are equally related, that is $C_{j,k} = a$ for every $1 \leq j, k \leq K$. In this case we learn the two non-negative hyperparameters a and ρ;
- HMTL: our hyperparameter optimization method REVERSE-HG to tune C and ρ;
- HMTL-S: Learning the matrix C with only few examples per class could bring the discovery of spurious relationships. We try to remove this effect by imposing the constraint that $\sum_{j,k} C_{j,k} \leq R$, where[6] $R = 10^{-3}$. In this case, Adam updates are projected onto the set $\{(\rho, C) : \rho \geq 0,\ C = C^\intercal,\ C \geq 0,\ \sum_{j,k} C_{j,k} \leq R\}$.

Results of five repetitions with different splits are presented in Table 2. Note that HMTL gives a visible improvement in

[5] Available at tinyurl.com/h2x8wws

[6] We observed that $R = 10^{-4}$ yielded very similar results.

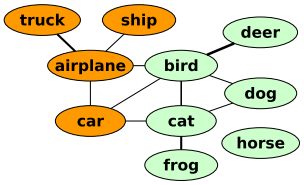

Figure 2: Relationship graph of CIFAR-10 classes. Edges represent interaction strength between classes.

Table 2: Test accuracy±standard deviation on CIFAR-10 and CIFAR-100 for single task learning (STL), naive MTL (NMTL) and our approach without (HMTL) and with (HMTL-S) the L1-norm constraint on matrix C.

	CIFAR-10	CIFAR-100
STL	67.47±2.78	18.99±1.12
NMTL	69.41±1.90	19.19±0.75
HMTL	70.85±1.87	21.15±0.36
HMTL-S	71.62±1.34	22.09±0.29

performance, and adding the constraint that $\sum_{j,k} C_{j,k} \leq R$ further improves performance in both datasets. The matrix C can been interpreted as an adjacency matrix of a graph, highlighting the relationships between the classes. Figure 2 depicts the graph for CIFAR-10 extracted from the algorithm HMTL-S. Although this result is strongly influenced by the choice of the data representations, we can note that animals tends to be more related to themselves than to vehicles and vice versa.

5.3. Phone Classification

The aim of the third set of experiments is to assess the efficacy of the real-time FORWARD-HG algorithm (RTHO). We run experiments on phone recognition in the multitask framework proposed in (Badino, 2016, and references therein). Data for all experiments was obtained from the TIMIT phonetic recognition dataset (Garofolo et al., 1993). The dataset contains 5040 sentences corresponding to around 1.5 million speech acoustic frames. Training, validation and test sets contain respectively 73%, 23% and 4% of the data. The primary task is a frame-level phone state classification with 183 classes and it consists in learning a mapping f_P from acoustic speech vectors to hidden Markov model monophone states. Each 25ms speech frame is represented by a 123-dimensional vector containing 40 Mel frequency scale cepstral coefficients and energy, augmented with their deltas and delta-deltas. We used a window of eleven frames centered around the prediction target to create the 1353-dimensional input to f_P. The secondary (or auxiliary) task consists in learning a mapping f_S from acoustic vectors to 300-dimensional real vectors of context-dependent phonetic embeddings defined in (Badino, 2016).

Table 3: Frame level phone-state classification accuracy on standard TIMIT test set and execution time in minutes on one Titan X GPU. For RS, we set a time budget of 300 minutes.

	Accuracy %	Time (min)
Vanilla	59.81	12
RS	60.36	300
RTHO	61.97	164
RTHO-NT	61.38	289

As in previous work, we assume that the two mappings f_P and f_S share inputs and an intermediate representation, obtained by four layers of a feed-forward neural network with 2000 units on each layer. We denote by W the parameter vector of these four shared layers. The network has two different output layers with parameter vectors W^P and W^S each relative to the primary and secondary task. The network is trained to jointly minimize $E_{\text{tot}}(W, W^P, W^S) = E_P(W, W^P) + \rho E_S(W, W^S)$, where the primary error E_P is the average cross-entropy loss on the primary task, the secondary error E_S is given by mean squared error on the embedding vectors and $\rho \geq 0$ is a design hyperparameter. Since we are ultimately interested in learning f_P, we formulate the hyperparameter optimization problem as

$$\min \left\{ E_{\text{val}}(W_T, W_T^P) \text{ subject to } \rho, \eta \geq 0, 0 \leq \mu \leq 1 \right\},$$

where E_{val} is the cross entropy loss computed on a validation set after T iterations of stochastic GDM, and η and μ are defined in (2). In all the experiments we fix a mini-batch size of 500. We compare the following methods:

1. Vanilla: the secondary target is ignored ($\rho = 0$); η and μ are set to 0.075 and 0.5 respectively as in (Badino, 2016).

2. RS: random search with $\rho \sim \mathcal{U}(0, 4)$, $\eta \sim \mathcal{E}(0.1)$ (exponential distribution with scale parameter 0.1) and $\mu \sim \mathcal{U}(0, 1)$ (Bergstra & Bengio, 2012).

3. RTHO: real-time hyperparameter optimization with initial learning rate and momentum factor as in Vanilla and initial ρ set to 1.6 (best value obtained by grid-search in Badino (2016)).

4. RTHO-NT: RTHO with "null teacher," i.e. when the initial values of ρ, η and μ are set to 0. We regard this experiment as particularly interesting: this initial setting, while clearly not optimal, does not require any background knowledge on the task at hand.

We also tried to run FORWARD-HG for a fixed number of epochs, not in real-time mode. Results are not reported

since the method could not make any appreciable progress after running 24 hours on a Titan X GPU.

Test accuracies and execution times are reported in Table 3. Figure 3 shows learning curves and hyperparameter evolutions for RTHO-NT. In Experiments 1 and 2 we employ a standard early stopping procedure on the validation accuracy, while in Experiments 3 and 4 a natural stopping time is given by the decay to 0 of the learning rate (see Figure 3 left-bottom plot). In Experiments 3 and 4 we used a hyper-batch size of $\Delta = 200$ (see Eq. (16)) and a hyper-learning rate of 0.005.

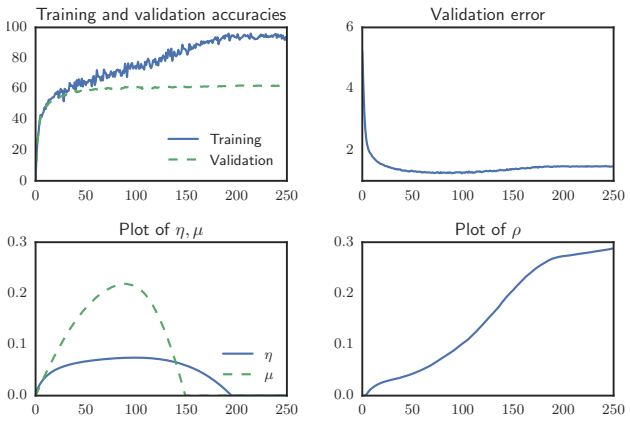

Figure 3: Learning curves and hyperparameter evolutions for RTHO-NT: the horizontal axis runs with the hyper-batches. Top-left: frame level accuracy on mini-batches (Training) and on a randomly selected subset of the validation set (Validation). Top-right: validation error E_{val} on the same subset of the validation set. Bottom-left: evolution of optimizer hyperparameters η and μ. Bottom-right: evolution of design hyperparameter ρ.

The best results in Table 3 are very similar to those obtained in state-of-the-art recognizers using multitask learning (Badino, 2016; 2017). In spite of the small number of hyperparameters, random search yields results only slightly better than the vanilla network (the result reported in Table 3 are an average over 5 trials, with a minimum and maximum accuracy of 59.93 and 60.86, respectively). Within the same time budget of 300 minutes, RTHO-NT is able to find hyperparameters yielding a substantial improvement over the vanilla version, thus effectively exploiting the auxiliary task. Note that the model trained has more that 15×10^6 parameters for a corresponding state of more than 30×10^6 variables. To the best of our knowledge, reverse-mode (Maclaurin et al., 2015) or approximate (Pedregosa, 2016) methods have not been applied to models of this size.

6. Discussion

We studied two alternative strategies for computing the hypergradients of any iterative differentiable learning dynamics. Previous work has mainly focused on the reverse-mode computation, attempting to deal with its space complexity, that becomes prohibitive for very large models such as deep networks.

Our first contribution is the definition and the application of forward-mode computation to HO. Our analysis suggests that for large models the forward-mode computation may be a preferable alternative to reverse-mode if the number of hyperparameters is small. Additionally, forward-mode is amenable to real-time hyperparameter updates, which we showed to be an effective strategy for large datasets (see Section 5.3). We showed experimentally that even starting from a far-from-optimal value of the hyperparameters (the null teacher), our RTHO algorithm finds good values at a reasonable cost, whereas other gradient-based algorithms could not be applied in this context.

Our second contribution is the Lagrangian derivation of the reverse-mode computation. It provides a general framework to tackle hyperparameter optimization problems involving a wide class of response functions, including those that take into account the whole parameter optimization dynamics. We have also presented in Sections 5.1 and 5.2 two non-standard learning problems where we specifically take advantage of a constrained formulation of the HO problem.

We close by highlighting some potential extensions of our framework and direction of future research. First, the relatively low cost of our RTHO algorithm could suggest to make it a standard tool for the optimization of real-valued critical hyperparameters (such as learning rates, regularization factors and error function design coefficient), in context where no previous or expert knowledge is available (e.g. novel domains). Yet, RTHO must be thoroughly validated on diverse datasets and with different models and settings to empirically asses its robustness and its ability to find good hyperparameter values. Second, in order to perform gradient-based hyperparameter optimization, it is necessary to set a descent procedure over the hyperparameters. In our experiments we have always used Adam with a manually adjusted value for the hyper-learning rate. Devising procedures which are adaptive in these hyper-hyperparameters is an important direction of future research. Third, extensions of gradient-based HO techniques to integer or nominal hyperparameters (such as the depth and the width of a neural network) require additional design efforts and may not arise naturally in our framework. Future research should instead focus on the integration of gradient-based algorithm with Bayesian optimization and/or with emerging reinforcement learning hyperparameter optimization approaches (Zoph & Le, 2016). A final important problem is to study the converge properties of RTHO. Results in Pedregosa (2016) may prove useful in this direction.

References

Badino, Leonardo. Phonetic context embeddings for dnn-hmm phone recognition. In *Proceedings of Interspeech*, pp. 405–409, 2016.

Badino, Leonardo. Personal communication, 2017.

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

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

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

Bergstra, James, Yamins, Daniel, and Cox, David D. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. *ICML*, 28:115–123, 2013.

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

Domke, Justin. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pp. 318–326, 2012.

Evgeniou, Theodoros, Micchelli, Charles A, and Pontil, Massimiliano. Learning multiple tasks with kernel methods. *J. Mach. Learn. Res.*, 6:615–637, 2005.

Fan, Yang, Tian, Fei, Qin, Tao, Bian, Jiang, and Liu, Tie-Yan. Learning what data to learn. *arXiv preprint arXiv:1702.08635*, 2017.

Garofolo, John S., Lamel, Lori F., Fisher, William M., Fiscus, Jonathon G., and Pallett, David S. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report*, 93, 1993.

Griewank, Andreas and Walther, Andrea. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, second edition, 2008.

Hutter, Frank, Hoos, Holger H., and Leyton-Brown, Kevin. Sequential model-based optimization for general algorithm configuration. In *Int. Conf. on Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Hutter, Frank, Lücke, Jörg, and Schmidt-Thieme, Lars. Beyond Manual Tuning of Hyperparameters. *KI - Künstliche Intelligenz*, 29(4):329–337, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*, 2014.

Krizhevsky, Alex and Hinton, Geoffrey. Learning multiple layers of features from tiny images. 2009.

Larsen, Jan, Hansen, Lars Kai, Svarer, Claus, and Ohlsson, M. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing*, pp. 62–71. IEEE, 1996.

LeCun, Yann. A Theoretical Framework for Back-Propagation. In Hinton, Geoffrey and Sejnowski, Terrence (eds.), *Proc. of the 1988 Connectionist models summer school*, pp. 21–28. Morgan Kaufmann, 1988.

Maclaurin, Dougal, Duvenaud, David K, and Adams, Ryan P. Gradient-based hyperparameter optimization through reversible learning. In *ICML*, pp. 2113–2122, 2015.

Pearlmutter, Barak A. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural networks*, 6(5):1212–1228, 1995.

Pedregosa, Fabian. Hyperparameter optimization with approximate gradient. In *ICML*, pp. 737–746, 2016.

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

Snoek, Jasper, Rippel, Oren, Swersky, Kevin, Kiros, Ryan, Satish, Nadathur, Sundaram, Narayanan, Patwary, Md Mostofa Ali, Prabhat, Mr, and Adams, Ryan P. Scalable Bayesian Optimization Using Deep Neural Networks. In *ICML*, pp. 2171–2180, 2015.

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

Werbos, Paul J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10): 1550–1560, 1990.

Williams, Ronald J. and Zipser, David. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.