Gradient-based Hyperparameter Optimization

Paolo Frasconi
Università degli Studi di Firenze, Italy
http://ai.dinfo.unifi.it/paolo/

Joint work with Luca Franceschi (IIT and UCL), Michele Donini (IIT), Massimiliano Pontil (IIT and UCL)

EMMCVPR 2017 — Venezia, November 1st, 2017

Hyperparameter Optimization

Most machine learning algorithms depend on the values of some variables that must be decided before learning starts.

At least three kinds of hyperparameters:

- **Regularization** (e.g. amount of L2 or L1 penalty, dropout, multitask, etc.)
- **Hypothesis space** (e.g. variables in the kernel, layers in neural nets, etc.)
- **Optimization** (e.g. learning rate, momentum, etc.)

HO: tune hyperparameters automatically

Some approaches to HO

- Grid search (trivial): only practical for 1–2 hyperparameters
- Random search: better than grid search (J. Bergstra and Bengio 2012) — 32 hyperparameters
- Bayesian approaches (J. Bergstra, Yamins, et al. 2013) (Hyperopt) — 238 hyperparameters
- Spearmint (Snoek et al. 2012) — 288 hyperparameters
- Sequential Model-Based (SMBO, SMAC) (Hutter et al. 2011)
- Tree-structured Parzen Estimator (J. S. Bergstra et al. 2011; Thornton et al. 2011)

Gradient-based Hyperparameter Optimization

Early works limited to few hyperparameters: (Bengio 2000; Larsen et al. 1996)

More recent works capable to handle one thousand hyperparameters (Maclaurin et al. 2015; Pedregosa 2016)
Simple example: Ridge regression

Prediction function: \( g : \mathbb{R}^d \mapsto \mathbb{R}, \quad g(x; w) = w^\top x \)

Learning problem:
\[
J(w, \lambda) = \sum_{(x,y) \in T} \left[ y - g(x; w) \right]^2 + \lambda \|w\|^2
\]

Closed-form solution:
\[
\hat{w}(\lambda) = \arg\min_w J(w, \lambda) = (X^\top X + \lambda I)^{-1} X^\top Y
\]

Response function:
\[
f(\lambda) = \sum_{(x,y) \in V} \left[ y - g(x; \hat{w}(\lambda)) \right]^2
\]
\[
= \sum_{(x,y) \in V} \left[ y - ((X^\top X + \lambda I)^{-1} X^\top Y)^\top x \right]^2
\]

HO as a bilevel program

Two sets of variables:
- **outer** variables: hyperparameters \( \lambda \)
- **inner** variables: parameters \( w \)

Optimize outer problem subject to optimum of the inner problem:
\[
\min_{\lambda} f(\lambda, \hat{w})
\]
\[
\text{s.t.} \quad \hat{w} \in \arg\min_w J(\lambda, w)
\]

In HO, the outer problem is the validation loss and the inner problem is the training objective.

We assume that the objectives are differentiable. However in general there is no closed-form solution. In facts, objectives may be non-convex. Thus we introduce learning dynamics encompassing those in stochastic gradient descent algorithms such as Nesterov, Adam, RMSProp etc.

**Example: SGD with momentum on neural networks**

Dynamical system:
\[
\begin{align*}
    v_t &= \mu v_{t-1} - \eta \nabla J_t(w_{t-1}) \\
    w_t &= w_{t-1} + v_t
\end{align*}
\]

\(w_t\) are the weights, \(v_t\) the velocities.

\(\mu\) and \(\eta\) are optimization hyperparameters.

\(J_t\) is the lower objective for the \(t\)-th minibatch.

**Learning dynamics in general**

\[s_t = \Phi_t(s_{t-1}, \lambda) \quad t = 1, \ldots, T\]

The state \(s_t\) contains parameters and accessory variables (e.g. velocities).

\(\Phi_t : (\mathbb{R}^d \times \mathbb{R}^m) \to \mathbb{R}^d\) is a smooth mapping representing the operation performed by the \(t\)-th step of the optimization algorithm (on minibatch \(t\)).

The iterates \(s_1, \ldots, s_T\) depend on the hyperparameters \(\lambda\) both explicitly and implicitly.

**Hyperparameter Optimization**

Change the bilevel program to use the parameters at the last iterate \(s_T\) rather than \(\hat{w}\):
\[
\min_{\lambda} f(\lambda)
\]

where \(f : \mathbb{R}^m \to \mathbb{R}\) is the response function redefined as
\[
f(\lambda) = E(s_T(\lambda))
\]

Hypergradient:
\[
\nabla f(\lambda) = \nabla E(s_T) \frac{ds_T}{d\lambda}
\]
Hyperparameter Optimization

Minibatch 1 Minibatch 2 Minibatch T Validation set

Similar to a recurrent neural network but:

- Minibatches are like inputs to the RNN
- The state of the RNN are like the parameters of the model
- Hyperparameter are like the weights of the RNN
- The validation error is like the training loss of the RNN

Indeed (Maclaurin et al. 2015) proposed to use backpropagation (without mentioning BPTT or RNNs)

Algorithmic Differentiation

Most (complex) functions of interest in ML can be computed by composing elementary operations whose derivatives are readily available.

Algorithmic differentiation more effective than alternative ways of computing derivatives such as:

- Numerical Differentiation (subject to rounding-off errors)
- Symbolic Differentiation (subject to expressions of exploding sizes)

Backpropagation (Werbos 1982) is perhaps the most widely known AD technique in machine learning.

Algorithmic Differentiation for RNNs

Not surprisingly, both reverse mode and forward mode AD was popular for training RNNs in the late 1980’s.

Backpropagation through time (see e.g., Werbos 1988, Pearlmutter 1989) is reverse mode AD.

Real-time recurrent learning (see e.g., Mozer 1989, Williams & Zipser 1989) is forward mode AD.

Describe a function \( y = f(x) \) via a computation graph (essentially a circuit) where each node contains a value \( v_i \).

There are two main approaches to AD:

- **Forward mode**: for each node \( i \) and a fixed input \( j \), define

  \[
  \dot{v}_i = \frac{\partial v_i}{\partial x_j}
  \]

  \( \dot{v}_i \) can be computed from \( \dot{v}_k \)’s (\( k \) parents of \( i \))

- **Reverse mode**: for each node in the graph and a fixed output \( y_j \), define

  \[
  \bar{v}_i = \frac{\partial y_j}{\partial v_i}
  \]

  \( \bar{v}_i \) can be computed from \( \bar{v}_k \)’s for each child (\( k \) children of \( i \)) but all \( v_i \) must be stored in memory!
The HO problem can be reformulated as a constrained optimization problem:

\[
\min_{\lambda, s_t, \ldots, s_T} \mathcal{E}(s_T)
\]

\[
s_t = \Phi_t(s_{t-1}, \lambda), \ t \in \{1, \ldots, T\}
\]

Classical Lagrangian formalism used to derive backprop (LeCun 1988)

\[
\mathcal{L}(s, \lambda, \alpha) = \mathcal{E}(s_T) + \sum_{t=1}^{T} \alpha_t(\Phi_t(s_{t-1}, \lambda) - s_t) \quad \alpha_t \in \mathbb{R}^d
\]

Constraints on hyperparameters can be specified naturally

The base step for the recursion is derived from

\[
\frac{\partial \mathcal{L}}{\partial s_T} = \nabla \mathcal{E}(s_T) - \alpha_T
\]

Finally the whole hypergradient is

\[
\frac{\partial \mathcal{L}}{\partial \lambda} = \sum_{t=1}^{T} \alpha_t \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda} B_t
\]

where \(B_t\) is a \((d \times m)\) matrix

\[
\text{Reverse-HG}(\lambda, s_0)
\]

1. **Inputs:** Current hyperparameters, \(\lambda\), initial state, \(s_0\)
2. **Outputs:** Hypergradient at \(\lambda\)
3. **for** \(t = 1\) **to** \(T\)
4. \(s_t = \Phi_t(s_{t-1}, \lambda)\) // \(d\) vector, all must be stored
5. \(\alpha_T = \nabla \mathcal{E}(s_T)\)
6. \(g = 0\)
7. **for** \(t = 1\) **to** \(T\)
8. \(A_{t+1} = \frac{\partial \Phi_{t+1}(s_t, \lambda)}{\partial s_t} B_t\) // \(d \times d\) matrix
9. \(B_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda} B_t\) // \(d \times m\) matrix
10. \(\alpha_t = \alpha_{t+1} A_{t+1}\) // \(d\) vector
11. \(g = g + \alpha_t B_t\) // \(m\) vector
12. **return** \(g\)
Use chain rule:
\[ \nabla f(\lambda) = \nabla E(s_T) \frac{ds_T}{d\lambda} \]

Plug in the learning dynamics:
\[ \frac{ds_t}{d\lambda} = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial s_{t-1}} \frac{ds_{t-1}}{d\lambda} + \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda} \]

\[ Z_t(d \times m) = A_t(d \times d) Z_{t-1}(d \times m) + B_t(d \times m) \]

\[ \nabla f(\lambda) = \nabla E(s_T) Z_T \]
[for \( t = 1 \) to \( T \)]
\[ s_t = \Phi_t(s_{t-1}, \lambda) \quad // \text{d vector} \]
\[ A_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial s_{t-1}} \quad // \text{d \times d matrix} \]
\[ B_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda} \quad // \text{d \times m matrix} \]
\[ Z_t = A_t Z_{t-1} + B_t \quad // \text{d \times m matrix} \]
\[ \text{// Memory for } s_t \text{ can be reused in this case!} \]
\[ \text{return } \nabla E(s) Z_T \]
For $t \in \{1, \ldots, T\}$ define

$$f_t(\lambda) = E(s_t(\lambda))$$

(the previous response function is $f_T$)

Partial hypergradients are available in forward mode:

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

Significant: we can update hyperparameters several times in a single optimization epoch, without having to wait until time $T$

Similar to RTRL, applicable to data streams (or large datasets)

---

**Analysis**

The two approaches have different time/space tradeoffs

Reverse mode needs to store the whole history of parameter updates — (Maclaurin et al. 2015) proposed to “invert” the update dynamics and recompute the trace rather than storing it in memory

Forward mode does not scale well with the number of hyperparameters

---

**RTHO($\lambda, s_0$)**

1. **Inputs:** initial hyperparameters, $\lambda$, initial state, $s_0$
2. **Outputs:** Final parameters, $s_T$
3. $Z_0 = 0$
4. for $t = 1$ to $T$
   5. $s_t = \Phi_t(s_{t-1}, \lambda)$ // d vector
   6. $A_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial s_{t-1}}$ // $d \times d$ matrix
   7. $B_t = \frac{\partial \Phi_t(s_{t-1}, \lambda)}{\partial \lambda}$ // $d \times m$ matrix
   8. $Z_t = A_tZ_{t-1} + B_t$ // $d \times m$ matrix
   9. // Memory for $A_t, B_t, Z_t$ can be reused!
10. if $t == 0$ (mod $\Delta$)
11. $\lambda = \lambda - \eta \nabla E(s_t)Z_t$
12. return $s_T$
Results from algorithmic differentiation (AD)

Let $F : \mathbb{R}^n \mapsto \mathbb{R}^p$ be any differentiable function

Let $c(n, p)$ and $s(n, p)$ be the time and space to evaluate $F$

Also let $J_F$ the $p \times n$ Jacobian matrix of $F$

General results (Baydin et al. 2015; Griewank and Walther 2008):

(i) For any $r \in \mathbb{R}^n$, $J_F r$ can be evaluated in time $O(c(n, p))$ and space $O(s(n, p))$ using forward-mode AD — hence the whole $J_F$ can be computed in time $O(nc(n, p))$ and space $O(s(n, p))$

(ii) For any vector $q \in \mathbb{R}^p$, the product $J_F^T q$ can be evaluated in both time and space $O(c(n, p))$ using reverse-mode AD — hence $J_F$ can be computed in time $O(pc(n, p))$ and space $O(c(n, p))$

Analysis of hypergradient computation (1)

Cost to evaluate the update map $\Phi_t$:

- time $g(d, m)^1$
- space $h(d, m)^2$

Then the response function $f(\lambda) : \mathbb{R}^m \mapsto \mathbb{R}$ can be evaluated in time $O(Tg(d, m))$ and space $O(h(d, m))$

Notes:

1 assuming the time required to compute the validation error does not affect the bound (realistic since the number of validation examples is typically lower than the number of training iterations.

2 since variables $s_t$ may be overwritten at each iteration

Analysis of FORWARD-HG

Apply Fact (i) from AD: FORWARD-HG takes time $O(Tmg(d, m))$ and space $O(h(d, m))$

Result can also be obtained by noting that product $A_t Z_{t-1}$ requires $m$ Jacobian-vector products, each costing $O(g(d, m))$, while computing the Jacobian $B_t$ takes time $O(mg(d, m))$

Analysis of REVERSE-HG

Apply Fact (ii) from AD: REVERSE-HG takes both time and space $O(Tg(d, m))$

Results can also be obtained by noting that $\alpha_t A_t$ and $\alpha_t B_t$ are transposed-Jacobian-vector products that in reverse-mode take both time $O(g(d, m))$

Note that in this case variables $s_t$ cannot be overwritten, explaining the much higher space requirement
Neural network with \( k \) weights trained by SGD or Adam

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 \( O(k) \)

Hence, REVERSE-HG takes time and space \( O(Tk) \) while FORWARD-HG takes time \( O(Tk) \) and space \( O(k) \)

In this case there is a dramatic difference in terms of memory requirements

Data hyper-cleaning: Setting

Noisy labels but can only afford to check a subset of them

Train on noisy data \( D \), cleaned data \( C \) as validation

One hyperparameter for each training example:

\[
J(\lambda, w) = \frac{1}{n} \sum_{i=1}^{n} \lambda^{(i)} \ell( y^{(i)}, f(x^{(i)}; w) )
\]

HO problem:

\[
\min_{\lambda} \sum_{(x,y) \in C} \ell(y, g(x; \hat{w}))
\]

s.t. \( \hat{w} = \arg \min_w J(\lambda, w) \)

\( \lambda^{(i)} \in [0, 1] \)

\( |\lambda|_1 < R \)

Data hyper-cleaning: experimental setup

MNIST digits, 5000 validation (cleaned) examples, 5000 training examples (50% corruption rate), 10000 test examples

\( g(x) = \text{softmax}(wx) \), \( \ell \) cross-entropy loss

REVERSE-HG to compute hypergradients, Adam to optimize hyperparameters
Data hyper-cleaning: performance measures

Oracle: test accuracy after fitting $w$ on validation plus cleaned portion of the training set

Baseline: test accuracy after fitting $w$ on validation and (noisy) train

DH-$R$: test accuracy for the hyper-cleaner for a given $L1$ radius $R$ (fit $w$ on validation plus training examples having $\lambda^{(i)}>0$)

---

Data hyper-cleaning: Results

Goal is to tune the hyperparameters $\lambda = (C, \rho)$ of a multi-task regularizer (Evgeniou et al. 2005)

$$\Omega(w, \lambda) = \sum_{j=1}^{K} \sum_{k=1}^{K} c_{j,k} \|w_j - w_k\|^2 + \rho \sum_{k=1}^{K} \|w_k\|^2$$

where $w_k$ are the parameters for task $k$ and $K$ is the number of tasks

$C$ is a symmetric non-negative matrix and $\rho > 0$

Training objective:

$$J(w, \lambda) = \sum_{(x,y) \in T} \ell(g(x, w), y) + \Omega(w, \lambda)$$

As before, the classifier $g$ is a (linear) softmax regressor and $\ell$ the cross-entropy loss.
Multi-task learning: setup

Datasets: CIFAR-10 and CIFAR-100
Features: Inception-V3 model trained on ImageNet (Szegedy et al. 2015)

Few-shots learning setup:
- CIFAR-10: 50 training examples (5 per class), 50 validation examples
- CIFAR-100: 300 training examples (3 per class), 300 validation examples

Multi-task learning: variants

SLT: single task learning, i.e. $C = 0$ and applying HO to $\rho$
NMTL: naive MTL scenario where all $C_{j,k} = a$, and applying HO to $a$ and $\rho$
HMTL: REVERSE-HG for tuning both $C$ and $\rho$
HMTL-S: additional constraint $\sum_{j,k} c_{j,k} \leq R$ to prevent spurious task interactions due to the few shot learning setting

Multi-task learning: results

<table>
<thead>
<tr>
<th></th>
<th>CIFAR-10</th>
<th>CIFAR-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL</td>
<td>67.47±2.78</td>
<td>18.99±1.12</td>
</tr>
<tr>
<td>NMTL</td>
<td>69.41±1.90</td>
<td>19.19±0.75</td>
</tr>
<tr>
<td>HMTL</td>
<td>70.85±1.87</td>
<td>21.15±0.36</td>
</tr>
<tr>
<td>HMTL-S</td>
<td>71.62±1.34</td>
<td>22.09±0.29</td>
</tr>
<tr>
<td>(Dinuzzo et al. 2011)</td>
<td>69.96±1.85</td>
<td></td>
</tr>
<tr>
<td>(Jawanpuria et al. 2015) ($p = 2$)</td>
<td>70.30±1.05</td>
<td></td>
</tr>
<tr>
<td>(Jawanpuria et al. 2015) ($p = 4/3$)</td>
<td>70.96±1.04</td>
<td></td>
</tr>
</tbody>
</table>
Phone classification: Dataset

TIMIT phonetic recognition dataset (Garofolo et al. 1993)

- 5040 sentences, 1.5 million 25ms speech acoustic frames
- 73% train 23% validation, 4% test
- 123-dimensional feature per frame (40 Mel cepstral coefficients + energy, with their delta and delta-delta)
- Window of 11 frames around the target (1353-dimensional input vectors)
- 183 classes (HMM monophone states)

Phone classification: Multi-task setting

Rationale for MTL: domain specific information of related tasks used as inductive bias for the primary task

- Primary task: phone recognition
- Secondary task: phonetic context embedding vectors (300-dimensional) of triphones, proposed in (Badino 2016)

Phone classification: Network

The network is simple but not tiny (about 16 million weights)

Phone classification: Optimization problem

Hyperparameters: learning rate $\eta$, momentum term $\mu$, importance of the secondary task $\rho$

Outer objective

$$\min_{\rho,\eta,\mu} \quad E(w_T, w_{p,T})$$

s.t. \quad $\rho, \eta \geq 0$

$$0 \leq \mu \leq 1$$

where the inner objective is

$$J(w, w_p, w_s) = J_p(w, w_p) + \rho J_s(w, w_s)$$
There are more than 10^7 parameters, reverse mode is not possible (because of memory).
Forward mode on the other hand is very time consuming.
RTHO effective and fast.

Frame level phone-state classification accuracy on standard TIMIT test set and execution time in minutes on one Titan X GPU:

<table>
<thead>
<tr>
<th>Accuracy %</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Aux task, η, μ as in (Badino 2016)</td>
<td>59.81</td>
</tr>
<tr>
<td>Random Search</td>
<td>60.36</td>
</tr>
<tr>
<td>RTHO</td>
<td>61.97</td>
</tr>
<tr>
<td>RTHO with null teacher (all HP=0)</td>
<td>61.38</td>
</tr>
</tbody>
</table>

Need better theory to explain RTHO (e.g. convergence rate).

The stochastic or real-time HO approach can also be applied in the case of reverse-mode by truncating hypergradients propagation (similar to truncated BPTT) — encouraging results in (Grazzi 2017).

We also lack statistical theory for HO: Can many hyperparameters overfit the validation set? Can we establish bounds?
Many recent works on meta-learning or learning-to-optimize can be formulated within a framework that is compatible with HO. For example, meta-learning can be seen as a bilevel program:

\[
\begin{align*}
\min_{\zeta} \quad & E(\zeta, \hat{\theta}) \\
\text{s.t.} \quad & \hat{\theta} \in \arg\min_{\theta} J(\zeta, \theta)
\end{align*}
\]

where

- \( E \) is the test error in the meta-training episodes
- \( J \) is the training error in the meta-training episodes
- \( \zeta \) are (hyper)parameters that index a class of hypothesis spaces
- \( \theta \) are parameters used to fit meta-train episodes

By tuning \( \zeta \) we select a particular hypothesis space that is hopefully well suited for novel (meta-test) learning episodes. Preliminary results on MiniImagenet are comparable or better than those reported in (Ravi & Larochelle 2016) for 1-shot learning.

Thank You!

Code available: https://github.com/lucfra/RFHO


