HAL
open science

Nonlinear Acceleration of Deep Neural Networks

Damien Scieur, Edouard Oyallon, Alexandre D 'Aspremont, Francis Bach

To cite this version:

Damien Scieur, Edouard Oyallon, Alexandre D 'Aspremont, Francis Bach. Nonlinear Acceleration of Deep Neural Networks. 2018. hal-01799269

HAL Id: hal-01799269

https://hal.science/hal-01799269

Preprint submitted on 24 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Nonlinear Acceleration of Deep Neural Networks

Damien Scieur[1,3,4], **Edouard Oyallon**[1,5], **Alexandre d'Aspremont**[2,1,3,4] **and Francis Bach**[1,3,4]

[1]INRIA ; [2]CNRS
[3]Département d'informatique de l'École normale supérieure
[4]PSL Research University, 75005 Paris, France
[5]CVN, CentraleSupelec, Université Paris-Saclay

Abstract

Regularized nonlinear acceleration (RNA) is a generic extrapolation scheme for optimization methods, with marginal computational overhead. It aims to improve convergence using only the iterates of simple iterative algorithms. However, so far its application to optimization was theoretically limited to gradient descent and other single-step algorithms. Here, we adapt RNA to a much broader setting including stochastic gradient with momentum and Nesterov's fast gradient. We use it to train deep neural networks, and empirically observe that extrapolated networks are more accurate, especially in the early iterations. A straightforward application of our algorithm when training ResNet-152 on ImageNet produces a top-1 test error of 20.88%, improving by 0.8% the reference classification pipeline. Furthermore, the code runs offline in this case, so it never negatively affects performance.

1 Introduction

Stochastic gradient descent is a popular and effective method to train neural networks [11, 4]. A lot of efforts have been invested in accelerating stochastic algorithms, in particular by deriving methods that adapt to the structure of the problem. Algorithms such as RMSProp [18] or Adam [10] are examples of direct modifications of gradient descent, and estimate some statistical momentum during optimization to speed up convergence. Unfortunately, these methods can fail to converge on some simple problems [14], and may fail to achieve state-of-the-art test accuracy on image classification problems. Other techniques have been developed to improve convergence speed or accuracy, such as adaptive batch-size for distributed SGD [7], or quasi-second-order methods which improve the rate of convergence of stochastic algorithms [1]. However, only a limited number of settings are covered by such techniques, which are not compatible with state-of-the-art architectures.

The approach we propose here is completely different as our method is built on top of existing optimization algorithms. Classical algorithms typically retain only the last iterate or the average [13] of iterates as their best estimate of the optimum, throwing away all the information contained in the converging sequence of iterates. As it is highly wasteful from a statistical perspective, extrapolation schemes estimate the optimum of an optimization problem using a weighted average of the last iterates produced by an algorithm, where the weights depend on the iterates.

For example, Aitken-Δ^2 or Wynn's ε-algorithm (a good survey can be found in [2]), provide an improved estimate of the limit of a sequence using the last few iterates, and these methods have been extended to the vector case, where they are known as Anderson acceleration [19], minimal polynomial extrapolation [3] or reduced rank extrapolation [5].

Network ensembling [20] can also be seen as combining several neural networks to improve convergence. However, contrary to our strategy, ensembling consists in training different networks *from different starting points* and then averaging the predictions or the parameters. Averaging the weights

Preprint. Work in progress.

of successive different neural networks from SGD iterations has been studied in [9], and our method can also be seen as an extension of this averaging idea for SGD, but with non-uniform adaptive weights.

Recent results by [16] adapted classical extrapolation techniques such as Aitken's Δ^2 and minimal polynomial extrapolation to design regularized extrapolation schemes accelerating convergence of basic methods such as gradient descent. They showed in particular that using only iterates from a very basic fixed-step gradient descent, these extrapolation algorithms produced solutions reaching the optimal convergence rate of [12], *without any modification to the original algorithm*. However, these results were limited to single-step algorithms such as gradient descent, thus excluding the much faster momentum-based methods such as SGD with momentum or Nesterov's algorithm. Our results here seek to accelerate these accelerated methods.

Overall, nonlinear acceleration has marginal computational complexity. On convex problems, the online version (which modifies iterations) is competitive with L-BFGS in our experiments (see Figure 1), and is robust to misspecified strong convexity parameters. On neural network training problems, the offline version improves both the test accuracy of early iterations, as well as the final accuracy (see Figures 2 and 4), with only minor modifications of existing learning pipelines (see Appendix C). It never hurts performance as it runs on top of the algorithm and does not affect iterations. Finally, our scheme produces smoother learning curves. When training neural networks, we observe in our experiments in Figure 3 that the convergence speedup produced by acceleration is much more significant in early iterations, which means it could serve for "rapid prototyping" of network architectures, with significant computational savings. [1]

2 Regularized Nonlinear Acceleration

2.1 Vector extrapolation methods

Vector extrapolation methods look at the sequence $\{x_0, \ldots, x_k\}$ produced by an iterative algorithm, and try to find its limit x^*. Typically, they assume x_k was produced by a fixed-point iteration with function g as follow,

$$x_{k+1} = g(x_k). \tag{1}$$

In the case of optimization methods, x^* is the function minimizer and g usually corresponds to a gradient step. In most cases, convergence analysis bounds are based on a Taylor approximation of g, and produce only local rates.

Recently, [16] showed global rates of convergence of regularized versions of Anderson acceleration. These results show in particular that, without regularization, classical extrapolation methods are highly unstable when applied to the iterates produced by optimization algorithms. However, the results hold only for sequences generated by (1) where g has a symmetric Jacobian.

To give a bit of intuition on extrapolation, let f be a (potentially) noisy objective function. We are interested in finding its minimizer $x^* \in \mathbb{R}^d$. To find this point, we typically use an iterative optimization algorithm and after N iterations obtain a sequence of points $\{x_i\} = \{x_0, x_1, \ldots, x_N\}$ converging to the critical point x^* where the gradient is zero. Vector extrapolation algorithms find linear combinations of the iterates $\{x_i\}$ with coefficients c_i to minimize the norm of the gradient, i.e.,

$$x_{\text{extr}} = \sum_{i=1}^{N} c_i x_i \quad \text{where} \;\; c \approx \arg\min_c \left\| \nabla f \left(\sum_{i=1}^{N} c_i x_i \right) \right\|_2 . \tag{2}$$

However, this problem is non-linear and hard to solve. The difference between extrapolation algorithms resides in the way they approximate the solution to (2).

[1] The source code for the numerical experiments can be found on GitHub: `https://github.com/windows7lover/RegularizedNonlinearAcceleration`

2.2 Regularized Nonlinear Acceleration (RNA) Algorithm.

In this paper, instead of considering the iterations produced by (1), we will look at a pair of sequences $\{x_k\}$, $\{y_k\}$ generated by

$$\begin{cases} x_k = g(y_{k-1}) \\ y_k = \sum_{i=1}^{k} \alpha_i^{(k)} x_i + \sum_{i=0}^{k-1} \beta_i^{(k)} y_i, \end{cases} \tag{3}$$

where both x_k, y_k converge to x^*. In this section, we will develop an extrapolation scheme for (3).

Algorithm 1 Regularized Nonlinear Acceleration (Complexity: $\mathcal{O}(N^2 d)$ if $N \ll d$)

Input: Sequences of length N from (3): $\{y_0, \ldots, y_{N-1}\}$, $\{x_1, \ldots, x_N\}$. Regularization param. λ.
Compute matrix of residues $R = [x_1 - y_0, \ldots, x_N - y_{N-1}]$.
Solve the linear system $(R^T R + \lambda I)z = \mathbf{1}$.
Normalize $c = z/(\mathbf{1}^T z)$.
Output: The extrapolated point $x_{\text{extr}}^N = \sum_{i=1}^{N} c_i x_i$.

Intuition. In this section, we show how to design the RNA algorithm for the special case where $g(x)$ is the gradient step with fixed step size h,

$$g(x) = x - h\nabla f(x). \tag{4}$$

The RNA algorithm approximates (2) by assuming that the function is approximately quadratic in the neighbourhood of $\{x_i\}$. This is a common assumption in optimization for the design of second-order methods, such as the Newton's method or BFGS, and implies that ∇f is approximately linear, so

$$\left\| \nabla f \left(\sum_{i=1}^{N} c_i x_i \right) \right\|_2 \approx \left\| \sum_{i=1}^{N} c_i \nabla f(x_i) \right\|_2 .$$

Since we apply g on y_{k-1} in (3), in view of (4) we have access to the gradient $\nabla f(y_{k-1})$ instead of $\nabla f(x_k)$. The optimal coefficients can be recovered by solving

$$\arg\min_c \left\| \sum_{i=1}^{N} c_i \nabla f(y_{i-1}) \right\| \qquad \text{s.t. } \sum_{i=1}^{N} c_i = 1, \tag{5}$$

where the constraint on c ensures convergence [16]. Even if we do not have an explicit access to the gradient, we can recover it from the differences between sequences $\{x_k\}$ and $\{y_k\}$ since

$$x_k - y_{k-1} = x_k - g(x_k) = -h\nabla f(y_{k-1}).$$

Writing $R = [\nabla f(y_0), \nabla f(y_1), \ldots]$, we can solve (5) explicitly, with

$$c = \frac{(R^T R)^{-1}\mathbf{1}}{\mathbf{1}^T (R^T R)^{-1}\mathbf{1}} \qquad \text{or again} \qquad (R^T R)z = \mathbf{1} \quad \text{then} \quad c = \frac{z}{\sum_{i=1}^{N} z_i}.$$

Omitting the regularization, these are the two main steps of the RNA Algorithm 1. In Appendix A we give a geometric interpretation of the algorithm.

Because the gradients are increasingly colinear as the algorithm converges, the matrix of gradients R quickly becomes ill-conditioned. Without regularization, solving (5) is highly unstable. We illustrate the impact of regularization on the conditioning in Figure 6 (in Appendix A), when optimizing a quadratic function. Even for this simple problem, the extrapolation coefficients c_i are large and oscillate between highly negative and positive values, and regularization dampens this behaviour.

Complexity. Roughly speaking, the RNA algorithm assumes that iterates follow a vector auto-regressive process of the form $(x_{k+1} - y_{k+1}) = A(x_k - y_k)$ for some matrix A [16], which is true when using gradient descent on quadratic functions, and holds asymptotically otherwise, provided some regularity conditions. The output of Algorithm 1 could be achieved by identifying the matrix A, recovering the quadratic function, then computing its minimum explicitly. Of course, the RNA algorithm does not perform these steps explicitly, so its complexity is bounded by $N^2 d$ where d is the dimension of the iterates and N the number of iterates used in estimating the optimum. In our experiments, N is typically equal to 5, so we can consider Algorithm 1 to be linear $O(d)$. In practice, computing the extrapolated solution on a CPU is faster than a single forward pass on a mini-batch.

Stochastic gradients. In the stochastic case, the concept of "iteration" is not as straightforward since one stochastic gradient is usually very noisy and non-informative, while it is not feasible to compute a full gradient due to problem scales. In this paper, we consider one iteration being one pass on the data with the stochastic algorithm and estimate the gradient on the fly.

Note that the extrapolated point x_{extr} in Algorithm 1 is computed only from the two sequences $\{x_k\}$ and $\{y_k\}$. Its computation does not require the function g, or access to the data set. Therefore, RNA can be used offline. On the other hand, we will see in the next subsection that it is possible to use RNA "online": we combine the extrapolation with the original algorithm. This often improves the observed rate of convergence.

Offline versus online acceleration. We now discuss several strategies for implementing RNA. The most basic one uses it offline: we simply generate an auxiliary sequence x_{extr}^N without interfering with the original algorithm. The main advantage of this strategy is to be *at least as good as* the vanilla algorithm, since we keep the sequences $\{x_k\}$ and $\{y_k\}$ unchanged. In addition, we can apply it *after* the N iterations of (3), since we only need the sequences and nothing else.

However, since x_{extr}^N is a better estimate of the optimum, restarting iterations from the extrapolated point can potentially improve convergence. The experiments in [16] restart the gradient method after a fixed number of iterations, to produce sequences that converge faster than classical accelerated methods in most cases.

Momentum acceleration. Concretely, we extend the results of [16] to handle iterative algorithms of the form (3) where g is a nonlinear iteration with a symmetric Jacobian. It allows push the approach a bit further, using the extrapolated point online, at *each step*. This was not possible in the scheme detailed in [16] because, as many other extrapolation schemes, [16] requires iterations of the form of (1).

In fact, the class of algorithms following (3) contains most common optimization schemes such as gradient descent with line-search or averaging and momentum-based methods. In fact, picking an algorithm is exactly equivalent to choosing values for $\alpha_i^{(k)}$ and $\beta_i^{(k)}$.

For example, in the case of Nesterov's method for $L-$smooth and convex functions, we get

$$g(x) = x - \frac{1}{L}\nabla f(x), \quad \alpha_k^{(k)} = 1 + \frac{k-2}{k+1}, \quad \alpha_{k-1}^{(k)} = -\frac{k-2}{k+1}, \quad \beta_{i\leq k-1} = \alpha_{i\leq k-2}^{(k)} = 0, \quad (6)$$

while for the gradient method with momentum to train neural networks we obtain

$$g(x) = x - h\nabla f(x), \quad \alpha_k^{(k)} = 1, \quad \beta_{k-1}^{(k)} = m, \quad \beta_{k-2}^{(k)} = -m, \quad \beta_{i\leq k-3} = \alpha_{i\leq k-1}^{(k)} = 0, \quad (7)$$

where h is the learning rate and m the momentum parameter. Running N iterations of (3) produces two sequences of iterates $\{y_0, \ldots, y_{N-1}\}$ and $\{x_1, \ldots, x_N\}$ converging to some minimizer x^* at a certain rate. In comparison, the setting of [16] corresponds to the special case of (3) where $y_k = x_k$, i.e., $\alpha_k = 1$ and all other coefficients are zero. In (6) and (7) we assume $g(x)$ constant over time, and a variable learning rate is captured by the $\alpha_i^{(k)}$ and β_i^k instead.

Because the extrapolation in (2) is a linear combination of previous iterates, it matches exactly the description of y_k in (3), with $\alpha_i^k = c_i$ and RNA can thus be directly injected in the algorithmic scheme in (3) as follows,

$$\begin{aligned} x_k &= g(y_{k-1}) \\ y_k &= x_{\text{extr}}^{(k)} = \textbf{RNA}(\{y_0, \ldots, y_{k-1}\},\ \{x_1, \ldots, x_k\}),\ \lambda). \end{aligned} \quad (8)$$

This trick potentially improves the previous version of RNA with "restarts", since we benefit from acceleration more often. We will see that this online version often improves significantly the rate of convergence of gradient and Nesterov methods for convex losses.

3 Optimal Convergence Rate for Linear Mappings

We now analyze the rate of convergence of the extrapolation step produced by Algorithm 1 as a function of the length of the sequences $\{x_k\}$ and $\{y_k\}$. We restrict our result to the specific case when g is the linear mapping

$$g(x) = G(x - x^*) + x^*. \quad (9)$$

This holds asymptotically if g is smooth enough. The matrix G should be symmetric, and its norm is strictly bounded by one, i.e., $\|G\|_2 \leq 1 - \kappa < 1$. Typically, $1-\kappa$ is the rate of convergence of the "vanilla" algorithm, where κ is usually linked to the condition number of the problem. The assumptions are the same as [16] for the linear case, except now the linear mapping g is coupled with an extra linear combination step in (3) which allows us to handle momentum terms or accelerated methods.

3.1 Convergence Bound

We now prove that the RNA algorithm applied to the iterates in (3) reaches an optimal rate of convergence when g is a linear mapping and when $\{x_k\}$ and $\{y_k\}$ are generated by (3). The theorem is valid for any choice of $\alpha_i^N,\ \beta_i^N$ (up to some mild assumptions), i.e., for any algorithm which can be written as (3).

To prove the convergence, we will bound the residue $\|x_k - g(x_k)\|$, which corresponds to the norm of the gradient when g is a gradient step. This is a classical convergence bound for algorithms applied to non-convex problems. In particular, the following Theorem shows the rate of convergence to a critical point x^* where $\|\nabla f(x^*)\| = 0$.

Because a critical point is not guaranteed to be a local minimum, the extrapolation can converge to a saddle point or a local maximum if (3) is doing so. However, if (3) converges to a minimum, then Algorithm 1 too.

Theorem 3.1. *Let $\{x_1, \ldots, x_N\}$ and $\{y_0, \ldots, y_{N-1}\}$ be the two sequences produced by running N iterations of the optimization algorithm in* (3)*, where $g(x)$ is the linear mapping* (9)*. Assume G is symmetric with $\|G\| \leq 1 - \kappa < 1$ and $\alpha_i^{(N)}$, $\beta_i^{(N)}$ are arbitrary coefficients such that*

$$\alpha_N^{(N)} \neq 0 \quad \textit{and} \quad \textstyle\sum_{i=1}^N \alpha_i^{(N)} + \sum_{i=0}^{N-1} \beta_i^{(N)} = 1.$$

Let $x_{extr}^{(N)}$ be the output of the RNA Algorithm 1 using these sequences. When $\lambda = 0$, the rate of convergence of the residue $\|g(x_{extr}^{(N)}) - x_{extr}^{(N)}\|_2$ is optimal and bounded by

$$\begin{aligned} \|g(x_{extr}^{(N)}) - x_{extr}^{(N)}\|_2 &\leq 2\frac{2\xi^N}{1+\xi^{2N}}\|g(x_0) - x_0\|_2, \quad \textit{where } \xi = \tfrac{1-\sqrt{\kappa}}{1+\sqrt{\kappa}} \\ &\leq 2(1-\sqrt{\kappa})^N \|g(x_0) - x_0\|_2. \end{aligned} \tag{10}$$

In particular, when $g(x)$ is a simple gradient step on a quadratic function f, this means

$$\|\nabla f(x_{extr}^{(N)})\| \leq 2(1-\sqrt{\kappa})^N \|\nabla f(x_0)\|_2.$$

Proof. For clarity, we remove the superscript (N) in the scope of this proof, and assume $\beta_i = 0$ without loss of generality. By definition,

$$x_{\text{extr}} = \sum_{i=1}^N c_i x_i\ , \quad \text{so} \quad g(x_{\text{extr}}) - x_{\text{extr}} = (G - I)\Big(\sum_{i=1}^N c_i x_i - x^*\Big).$$

Since $\|G\|_2 \leq (1-\kappa) < 1$, we can bound $\|G - I\|_2 \leq 2$, hence if c is constructed as in Algorithm 1

$$\begin{aligned} \left\|(G-I)\left(\sum_{i=1}^N c_i x_i - x^*\right)\right\| &\leq 2\left\|\sum_{i=1}^N c_i x_i - x^*\right\|, \\ &= 2\min_{\mathbf{1}^T c = 1}\Big\|\sum_{i=1}^N c_i(x_i - x^*)\Big\|. \end{aligned} \tag{11}$$

We will now show by induction that all terms $x_k - x^*$ are computed using matrix polynomials in G of degree exactly k satisfying $p(1) = 1$ applied to the vector $(x_0 - x^*)$. This holds trivially for $k = 1$. Assume now this is true for k with

$$x_k - x^* = p_k(G)(x_0 - x^*), \quad p(1) = 1, \quad \text{and} \quad \deg(p_k) = k.$$

In this case,

$$y_k = \sum_{i=1}^{k} \alpha_i p_i(G)(x_0 - x^*) = q_k(G)(x_0 - x^*)$$

where q_k is a polynomial of degree $\leq k$. In fact, we can show that $\deg(q_k) = k$. Indeed,

$$\deg(q_k) = \deg\big(\sum_{i=1}^{k} \alpha_i p_i\big) = \deg(\alpha_k p_k)$$

because by the induction hypothesis, $\deg(p_i) < k$ when $i < k$. Since $\alpha_k \neq 0$ by assumption,

$$\deg(q_k) = \deg(p_k) = k.$$

We will now show that $q_k(1) = 1$. Indeed,

$$q_k(1) = \sum_{i=1}^{k} \alpha_i p_i(1) = \sum_{i=1}^{k} \alpha_i = 1,$$

where the first equality is obtained by the induction hypothesis, and the second using our assumption on the coefficients α_i. Finally,

$$x_{k+1} - x^* = G q_k(G)(x_0 - x^*) = p_{k+1}(x)(x_0 - x^*)$$

which means $p_{k+1}(x) = x q_k(x)$. Clearly,

$$\deg(p_{k+1}) = 1 + \deg q_k = k + 1 \qquad \text{and} \qquad p_{k+1}(1) = 1 q_k(1) = 1,$$

which proves the induction. This means that the family of polynomial $\{p_i\}$ generates $\mathcal{P}_N$, the subspace of polynomial of degree $\leq N$. We can thus rewrite (11) using polynomials,

$$\begin{aligned} \min_{\mathbf{1}^T c = 1} \Big\| \sum_{i=1}^{N} c_i (x_i - x^*) \Big\| &= \min_{c : \mathbf{1}^T c = 1} \Big\| \sum_{i=1}^{N} c_i p_i(G)(x_0 - x^*) \Big\| \\ &= \min_{p \in \mathcal{P}_N} \Big\| p(G)(x_0 - x^*) \Big\|. \end{aligned}$$

The explicit solution involves rescaled Chebyshev polynomials described in [6], whose optimal value is exactly (10), as stated, e.g., in Proposition 2.2 of [16]. ■

More concretely, this last result allows us to apply RNA to stochastic gradient algorithms featuring a momentum term. It also allows using a full extrapolation step at each iteration of Nesterov's method, instead of the simple momentum term in the classical formulation in this method. As we will observe in the numerical section, this yields significant computation gains in both cases.

The previous result also shows that our method is *adaptive*. Whatever the algorithm we use for optimizing a quadratic function, if the iteration $g(x)$ converges, the coefficients α_i and β_i sum to one and we have α_N non-zero, we obtain an optimal rate of convergence. For example, when the strong convexity parameter is unknown, Nesterov's method (6) has a rate of convergence in $O(1/k^2)$ on a quadratic functions, even if the function is strongly convex. By post-processing the iterates, we transform this method into an optimal algorithm, automatically adapting to the strong convexity constant, hence extrapolation *adaptively recovers the optimal rate of convergence* even when a bad momentum parameter is used.

The setting of Theorem 3.1 assumes we store indefinitely the points x_k and y_k. In practice we only keep a constant window, but it still improves the convergence speed by a constant factor (see Section 4 and Figure 1 for more details). The link between the theorem and a windowed version of RNA is similar to the one between BFGS, whose optimal convergence is also proved for quadratics, and its limited memory version L-BFGS used in practice.

Because it applies only to quadratics, our result is essentially asymptotic. However, an argument similar to that detailed in [15] would also give us non asymptotic, albeit less explicit, bounds. Similarly, the bound looks only at the non-regularized version, but it is possible to extend it using the solution of the regularized Chebyshev polynomial ([15], Proposition 3.3). Overall, as for the theoretical bounds on BFGS, these global bounds on RNA performance are highly conservative and do not faithfully reflect its numerical efficiency.

4 Numerical Experiments

The following numerical experiments seek to highlight the benefits of RNA in its offline and online versions when applied to the gradient method (with or without momentum term). Since the complexity grows quadratically with the number N of points in the sequences $\{x_i\}$ and $\{y_i\}$, we will use RNA with a fixed window size ($N = 5$ for stochastic and $N = 10$ for convex problems) in all these experiments. These values are sufficiently large to show a significant improvement in the rate of convergence, but can of course be fine-tuned. For simplicity, we fix $\lambda = 10^{-8} \cdot \|R^T R\|_2$.

4.1 Logistic Regression

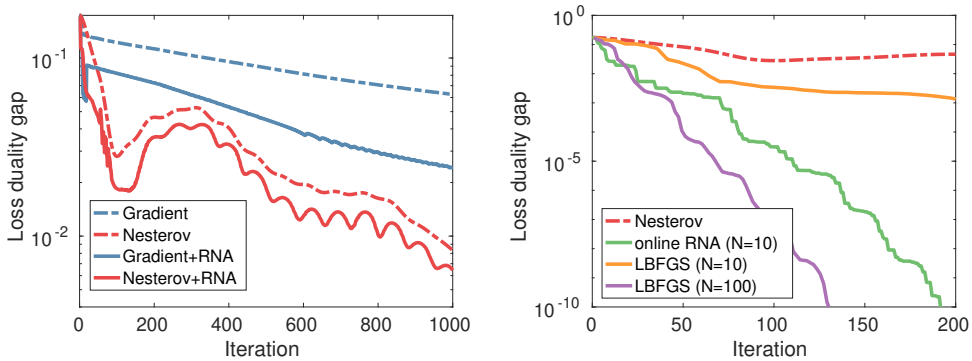

Figure 1: Logistic loss on Madelon [8]. Left: offline acceleration on gradient and Nesterov algorithms, both with backtracking line-search. Right: online RNA algorithm performs as well as L-BFGS methods, though RNA does not use line-search and requires 10 times less memory.

We solve a classical regression problem on the Madelon-UCI dataset [8] using the logistic loss with ℓ_2 regularization. The regularization has been set such that the condition number of the function is equal to 10^6. We compare to standard algorithms such as the simple gradient scheme, Nesterov's method for smooth and strongly convex objectives [12] and L-BFGS. For the step length parameter, we used a backtracking line-search strategy. We compare these methods with their offline RNA accelerated counterparts, as well as with the online version of RNA described in (8). Results are reported in Figure 1.

On Figure 1, we observe that offline RNA improves the convergence speed of gradient descent and Nesterov's method. However, the improvement is only a constant factor: the curves are shifted but have the same slope. Meanwhile, the online version greatly improves the rate of convergence, transforming the basic gradient method into an optimal algorithm competitive with line-search L-BFGS.

In opposition to most quasi-newton methods (such as L-BFGS), RNA does *not* require a Wolfe line-search to be convergent. This is because the algorithm is stabilized with a Tikhonov regularization. In addition, the regularization in a way controls the impact of the noise in the iterates, making the RNA algorithm suitable for stochastic iterations [15].

4.2 Image Classification

We now describe experiments with CNNs for image classification. Because one stochastic iteration is not informative due to the noise, we refer to x_k as the model parameters (including batch normalization statistics) corresponding to the final iteration of the epoch k. In this case, we do not have an explicit access to "$(x_k - y_{k-1})$", so we will estimate it during the stochastic steps. Let $y_k^{(t)}$ be the parameters of the network at epoch k after t stochastic iterations, and $x_k^{(t+1)}$ be the parameters after one stochastic gradient step. Then, for a data set of size D,

$$x_k - y_{k-1} \approx \frac{1}{D}\sum_{t=1}^{D}(x_k^{(t+1)} - y_k^{(t)}) = -h\frac{1}{D}\sum_{t=1}^{D}\nabla f(y_k^{(t)}).$$

This means the matrix R in Algorithm 1 will be the matrix of (estimated) gradients as described in (5). In Appendix C, we provide a pseudo-code implementation of RNA algorithm.

Because the learning curve is highly dependent on the learning rate schedule, we decided to use a linearly decaying learning rate to better illustrate the benefits of acceleration, even if acceleration also works with a constant learning rate schedule (see [17] and Figure 3). In all our experiments, until epoch T, the learning rate decreases linearly from an initial value h_0 to a final value h_T, with

$$h_k = h_0 + (k/T)(h_T - h_0). \tag{12}$$

We then continue the optimization during 10 additional epochs using h_T to stabilize the curve. We summarize the parameters used for the optimization in Table 1.

CIFAR10. CIFAR-10 is a standard 10-class image dataset comprising $5 \cdot 10^4$ training samples and 10^4 samples for testing. Except for the linear learning rate schedule above, we follow the standard practice for CIFAR-10. We applied the standard augmentation via padding of 4 pixels. We trained the networks VGG19, ResNet-18 and DenseNet121 during 100 epochs ($T = 90$) with a weight decay of $5 \cdot 10^{-4}$.

We observe in Figure 7 in Appendix B that the online version does not perform as well as in the convex case. More surprisingly, it is outperformed by its offline version (Figure 2) which computes the iterates on the side.

In fact, the offline experiments detailed in Figure 2 exhibit much more significant gains. It produces a similar test accuracy, and the offline version converges faster than SGD, especially for early iterations. We reported speedup factors to reach a certain tolerance in Tables 2, 3 and 4. This suggests that the offline version of RNA is a good candidate for training neural networks, as it converges faster while guaranteeing performance *at least* as good as the reference algorithm. Additional figures of networks VGG19 and DenseNet121 can be found in Appendix B, Figure 8.

ImageNet. Here, we apply the RNA algorithm to the standard ImageNet dataset. We trained the networks during 90 epochs ($T = 80$) with a weight decay of 10^{-4}. We reported the test accuracy on Figure 4 for the networks ResNet-50 and ResNet-152. We only tested the offline version of RNA here, because in previous experiments it gives better result than its online counterpart.

We again observe that the offline version of Algorithm 1 improves the convergence speed of SGD with and without momentum. In addition, we show a substantial improvement of the accuracy over the non-accelerated baseline. The improvement in the accuracy is reported in Figure 5. Interestingly, the resulting training loss is smoother than its non accelerated counterpart, which indicates a noise reduction.

5 Conclusion

We extend the Regularized Nonlinear Acceleration scheme in [16] to cover algorithms such as stochastic gradient methods with momentum and Nesterov's method. As the original scheme, it has optimal complexity on convex quadratic problems, but it is also amenable to non-convex optimization problems such as deep CNNs training.

As an online algorithm, RNA substantially improves the convergence rate of Nesterov's algorithm on convex problems such as logistic regression. On the other hand, when applied offline to CNN training, it improves both accuracy and convergence speed. This could be prove useful for fast prototyping of neural networks architectures.

Acknowledgements

We acknowledge support from the European Union's Seventh Framework Programme (FP7-PEOPLE-2013-ITN) under grant agreement n.607290 SpaRTaN and from the European Research Council (grant SEQUOIA 724063). Alexandre d'Aspremont was partially supported by the data science joint research initiative with the fonds AXA pour la recherche and Kamet Ventures. Edouard Oyallon was partially supported by a postdoctoral grant from DPEI of Inria (AAR 2017POD057) for the collaboration with CWI.

	h_0	h_T	momentum
SGD and Online RNA (8)	1.0	0.01	0
SGD + momentum	0.1	0.001	0.9

Table 1: Parameters used in (12) to generate the learning rate for optimizers. We used the same setting for their accelerated version with RNA.

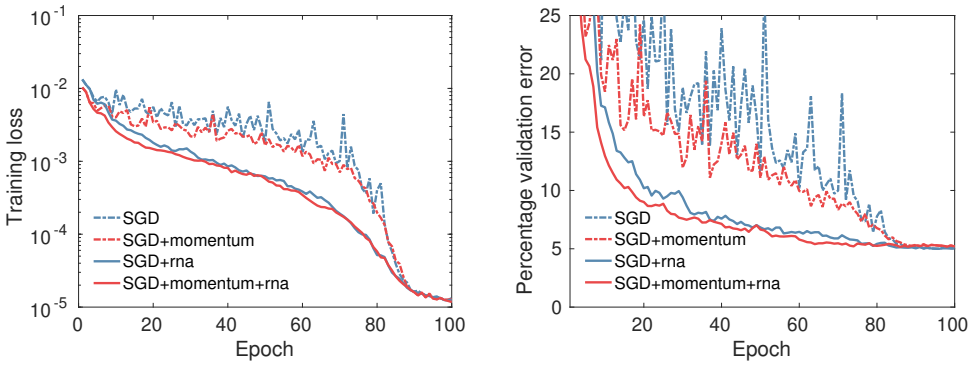

Figure 2: Resnet-18 on Cifar10, 100 epochs. SGD with and without momentum, and their off-line accelerated versions with a window size 5. Left: training loss. Right: top-1 validation error.

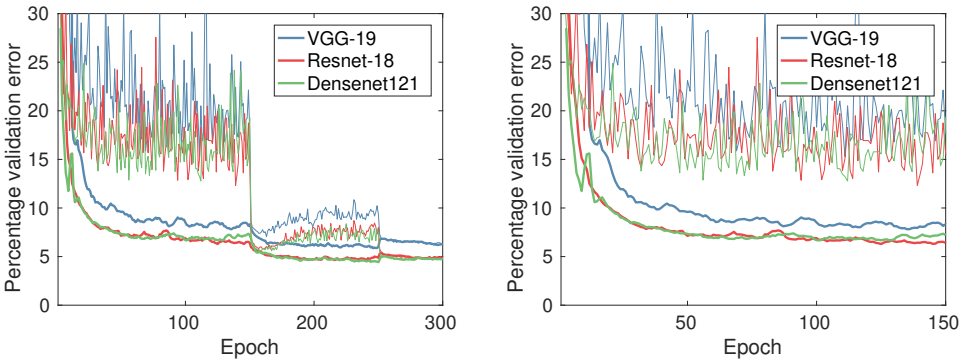

Figure 3: Prototyping networks: acceleration (bottom) gives a smoother convergence, producing a clearer ranking of architectures, earlier (flat learning rate). Right plot zooms on left one.

Tolerance	SGD	SGD+momentum	SGD+RNA	SGD+momentum+RNA
5.0%	68 (0.87×)	59	21 (2.81×)	**16** (3.69×)
2.0%	78 (0.99×)	77	47 (1.64×)	**40** (1.93×)
1.0%	82 (1.00×)	82	67 (1.22×)	**59** (1.39×)
0.5%	84 (1.02×)	86	75 (1.15×)	**63** (1.37×)
0.2%	86 (1.13×)	97	**84** (1.15×)	85 (1.14×)

Table 2: Number of epochs required to reach the best test accuracy + *Tolerance*% on CIFAR10 with a Resnet18, using several algorithms (best accuracy is 5% here). The speed-up compared to the SGD+momentum baseline is in parenthesis.

Tolerance	SGD	SGD+momentum	SGD+RNA	SGD+momentum+RNA
5.0%	69 (0.87×)	60	26 (2.31×)	**24** (2.50×)
2.0%	83 (0.99×)	82	52 (1.58×)	**45** (1.82×)
1.0%	84 (1.02×)	86	71 (1.21×)	**60** (1.43×)
0.5%	89 (0.98×)	87	73 (1.19×)	**62** (1.40×)
0.2%	**N/A**	90	99 (0.90×)	**63** (1.43×)

Table 3: Number of epochs required to reach the best test accuracy + *Tolerance*% on CIFAR10 with VGG19. Here, the best accuracy is 6.54%

Tolerance	SGD	SGD+momentum	SGD+RNA	SGD+momentum+RNA
5.0%	65 (0.86×)	56	22 (2.55×)	**13** (4.31×)
2.0%	80 (0.98×)	78	45 (1.73×)	**38** (2.05×)
1.0%	83 (1.00×)	83	60 (1.38×)	**56** (1.48×)
0.5%	87 (0.99×)	86	80 (1.08×)	**66** (1.30×)
0.2%	92 (1.01×)	93	86 (1.08×)	**75** (1.24×)

Table 4: Number of epochs required to reach the best test accuracy + *Tolerance*% on CIFAR10 with DenseNet121. Here, the best accuracy is 4.62%.

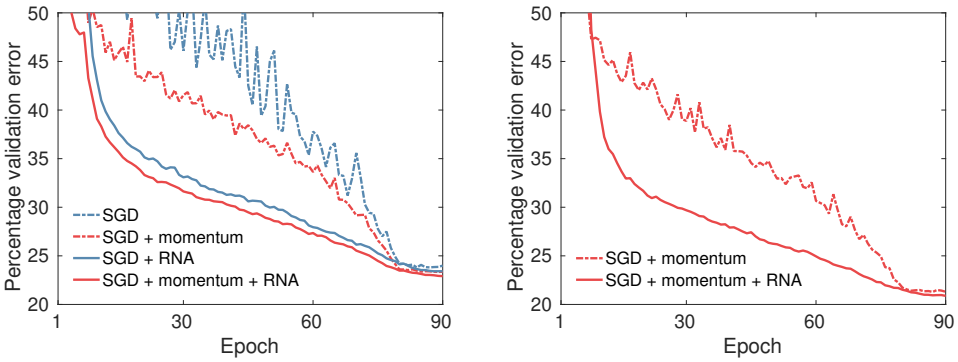

Figure 4: Training a Resnet-52 (*left*) and ResNet-152 (*right*) on validation ImageNet for 90 epochs using SGD with and without momentum, and their off-line accelerated versions.

	Pytorch	SGD	SGD+mom.	SGD+RNA	SGD+mom.+RNA
Resnet-50	23.85	23.808	23.346	23.412 (-0.396%)	**22.914** (-0.432%)
Resnet-152	21.69	N/A	21.294	N/A	**20.884** (-0.410%)

Table 5: Best validation top-1 error percentage on ImageNet. In parenthesis the improvement due to RNA. The first column corresponds to the performance of Pytorch pre-trained models.

References

[1] R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. M. Shi, and P. T. P. Tang. A progressive batching L-BFGS method for machine learning. *arXiv preprint arXiv:1802.05374*, 2018.

[2] C. Brezinski and M. R. Zaglia. *Extrapolation methods: theory and practice*, volume 2. Elsevier, 2013.

[3] S. Cabay and L. Jackson. A polynomial extrapolation method for finding limits and antilimits of vector sequences. *SIAM Journal on Numerical Analysis*, 13(5):734–752, 1976.

[4] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, et al. Recent advances in deep learning for speech research at microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

[5] R. Eddy. Extrapolating to the limit of a vector sequence. In *Information linkage between applied mathematics and industry*, pages 387–396. Elsevier, 1979.

[6] G. H. Golub and R. S. Varga. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order richardson iterative methods. *Numerische Mathematik*, 3(1):157–168, 1961.

[7] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[8] I. Guyon. Design of experiments of the nips 2003 variable selection benchmark, 2003.

[9] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

[10] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] E. Moulines and F. Bach. Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In *Advances in Neural Information Processing Systems*, 2011.

[12] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[13] B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

[14] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018.

[15] D. Scieur, F. Bach, and A. d'Aspremont. Nonlinear acceleration of stochastic algorithms. In *Advances in Neural Information Processing Systems*, pages 3985–3994, 2017.

[16] D. Scieur, A. d'Aspremont, and F. Bach. Regularized nonlinear acceleration. In *Advances In Neural Information Processing Systems*, pages 712–720, 2016.

[17] D. Scieur, E. Oyallon, A. d'Aspremont, and F. Bach. Nonlinear acceleration of cnns. In *Workshop track of International Conference on Learning Representations (ICLR)*, 2018.

[18] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[19] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49(4):1715–1735, 2011.

[20] Z.-H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1-2):239–263, 2002.

Appendices

A Geometric interpretation of RNA

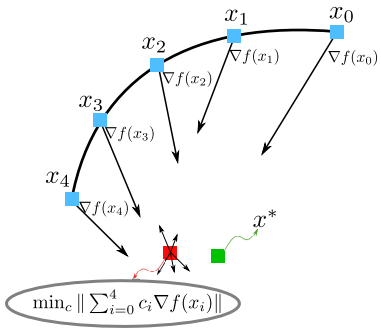

Figure 5: Regularized nonlinear acceleration can be viewed as finding the center of mass (in red). The gradients represent the forces and the green point is the real critical point x^*, whose gradient is equal to zero.

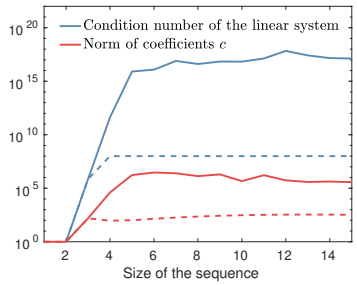

Figure 6: Condition number of the matrix $(R^T R + \lambda I)$ and the norm of the coefficient c produced by Algorithm 1. In plain line the non-regularized version, and in dashed-line the regularized version, with $\lambda = \|R^T R\|_2^2 \cdot 10^{-8}$.

We can give a geometric interpretation of the RNA algorithm. In view of (5), we can link its output with the *center of mass* of an object, whose forces are represented with gradients. In Figure 5 we show the trajectory of a gradient + momentum algorithm, where the extrapolated point is shown in red while the real solution x^* is in green.

B Additional Figures

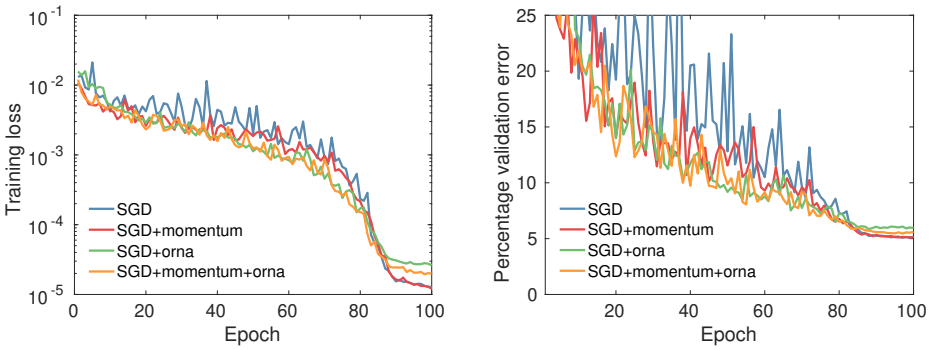

Figure 7: Training a Resnet-18 on Cifar10 for 100 epochs with SGD and the online RNA (8). On the left, the evolution of the loss value. On the right, the error percentage on the testing set. We see the online version does not improve the performance of SGD. This may be explained by the presence of the batchnorm module. The presence of a statistical estimation *while* the optimization is running is not taken into account in the iteration model (9), and thus makes theoretical analysis more complex. In addition, because we are optimizing a nonconvex function, and the online version interferes with the optimization scheme, the final performance is not guaranteed to be as good as the non-accelerated version of the optimizer.

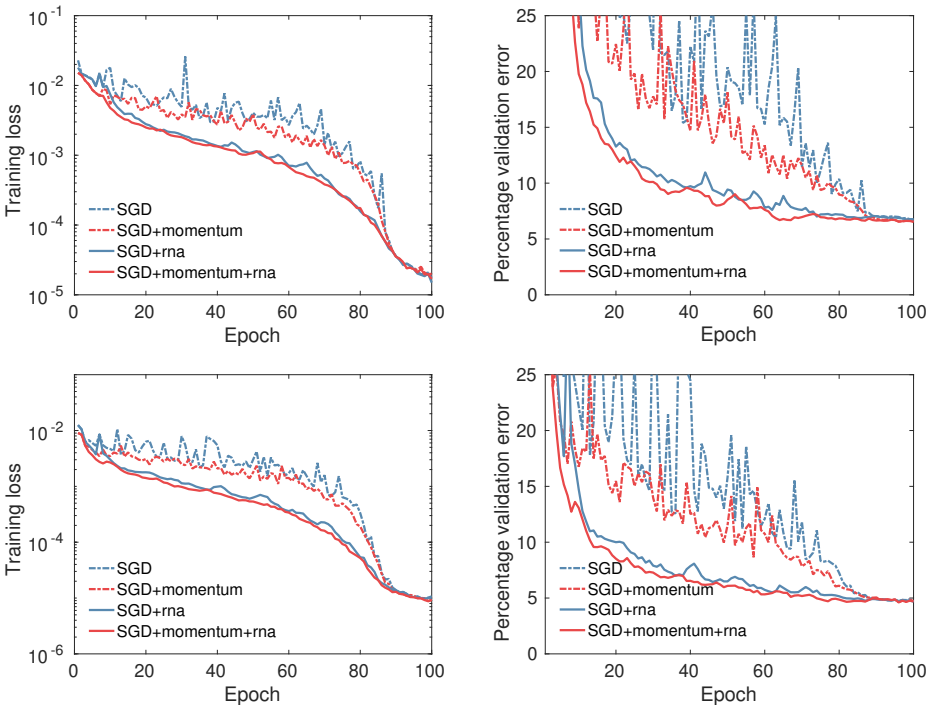

Figure 8: Training a VGG19 (*top*) and a DenseNet-121 (*bottom*) on Cifar10 for 100 epochs. SGD with and without momentum, and their off-line accelerated versions with a window size of 5. On the left, the evolution of the loss value. On the right, the error percentage on the testing set.

C Pseudo-code for Standard Deep Learning Pipelines with RNA

Listing 1: Pseudo-code for RNA

```
class Rna: # This class combines the optimizer with RNA

    def __init__(self,network,optimizer,N,reg):
        # - network: DNN to train
        # - optimizer is, for example, an instance of SGD
        # - N is the length of sequences x_k and y_k
        # - reg is the regularization parameter lambda

        self.network = network
        self.K = K
        self.optimizer = optimizer
        self.reg = reg
        self.history_net = []  # Store the seq. x_k
        self.history_grad = [] # Store the seq. x_k-y_{k-1}
        self.running_grad = [] # Estimation of current grad.
        self.grad_counter = 0

    def step(self):
        # Perform a standard step of the optimizer
        self.optimizer.step()

        # Update the gradient estimate (running average)
        self.grad_counter += 1
        coeff = 1.0/self.grad_counter
        self.running_grad *= 1-coeff
        self.running_grad += coeff*self.network.grad

    def store(self):
        # This function stores the network and the current
        # gradient estimate, then resets the running average

        self.history_net.append(self.network.copy())
        self.history_grad.append(self.running_grad);
        self.running_grad *= 0; # reset estimation

        if(len(self.history_grad)>self.N):
            self.history_grad.pop() # Length(x_k-y_{k-1}) <= N
        if(len(self.history_net)>self.N):
            self.history_net.pop() # Length(x_k) <= N

    def accelerate(self,extrapolated_network):
        # Straightforward application of RNA algorithm

        R = to_matrix(self.history_grad) # Get Matrix R
        RR = matrix_mult(R.transpose(),R) # compute R^T*R
        RR = RR/norm(RR) # Normalization
        RR = RR + self.reg*eye(self.N) # \ell-2 regularization

        # Solve (RR+lambda I)z = 1, then c = z/sum(z)
        z = solve_system(RR,ones(self.N,1))
        c = z/sum(z)

        # Combine previous networks with coefficients c
        x_extr = matrix_mult(to_matrix(self.history_net),c)
        extrapolated_network.load_params(x_extr)
```

Listing 2: Pseudo-code for training a DNN

```
import *packages*

(train_loader,validation_loader) = getDataset(...) # e.g., CIFAR10
network = getNetwork(...) # DNN, e.g., ResNet-18
criterion = getLoss(...) # Loss, e.g., cross-entropy
optimizer = getOptimizer(...) # Optimizer, e.g., SGD with momentum

# Plug optimizer into RNA with default parameters
rna_optimizer = Rna(network, optimizer, N=5, reg=1e-8)

# Offline version: we create a new network on the side
network_extr = network.copy()

for epoch in range(0, max_epoch):

    # Train for one epoch
    for (inputs, targets) in train_loader:
        optimizer.zero_grad()

        # Forward pass
        predictions = network.forward(inputs)
        loss.forward(predictions, targets)

        # Backward pass + optimizer step
        loss.backward()
        rna_optimizer.step()

    # Store the gradient estimate, then recover extrapolation
    rna_optimizer.store()
    rna_optimizer.accelerate(network_extr)

    # For the online acceleration version, uncomment this line
    # network.load_params(network_extr.params)

    # Test on the validation set
    correct = 0
    total = 0
    for (inputs, targets) in val_loader:
        predictions = network_extr(inputs)
        correct += sum(predictions == targets)
        total += length(targets)

    print("Top-1 score: %f", 100.0*correct/total)

# In comparison to standard script, we modified optimizer into
# rna_optimizer and added 4 lines of code:
#     rna_optimizer = Rna(network, optimizer, N=5, reg=1e-8)
#     network_extr = network.copy()
#     rna_optimizer.store()
#     rna_optimizer.accelerate(network_extr)
#
##################################################################
#   Except those lines, no-adhoc modifications of the existing   #
#   Pytorch (or any similar) code have been applied.             #
##################################################################
```