Embedded hyper-parameter tuning by Simulated Annealing

Matteo Fischetti *
Department of Information Engineering
University of Padova, Italy
matteo.fischetti@unipd.it

Matteo Stringher
Department of Information Engineering
University of Padova, Italy
stringher.matteo@gmail.com

Abstract

We propose a new metaheuristic training scheme that combines Stochastic Gradient Descent (SGD) and Discrete Optimization in an unconventional way. Our idea is to define a discrete neighborhood of the current SGD point containing a number of "potentially good moves" that exploit gradient information, and to search this neighborhood by using a classical metaheuristic scheme borrowed from Discrete Optimization.

In the present paper we investigate the use of a simple Simulated Annealing (SA) metaheuristic that accepts/rejects a candidate new solution in the neighborhood with a probability that depends both on the new solution quality and on a parameter (the *temperature*) which is modified over time to lower the probability of accepting worsening moves. We use this scheme as an automatic way to perform hyper-parameter tuning, hence the title of the paper. A distinctive feature of our scheme is that hyper-parameters are modified within a *single* SGD execution (and not in an external loop, as customary) and evaluated on the fly on the current minibatch, i.e., their tuning is fully embedded within the SGD algorithm.

The use of SA for training is not new, but previous proposals were mainly intended for non-differentiable objective functions for which SGD is not applied due to the lack of gradients. On the contrary, our SA method requires differentiability of (a proxy of) the loss function, and leverages on the availability of a gradient direction to define local moves that have a large probability to improve the current solution.

Computational results on image classification (CIFAR-10) are reported, showing that the proposed approach leads to an improvement of the final validation accuracy for modern Deep Neural Networks such as ResNet34 and VGG16.

1 Introduction

Stochastic Gradient Descent (SGD) is *de facto* the standard algorithm for training Deep Neural Networks (DNNs). Leveraging the gradient, SGD allows one to rapidly find a good solution in the very high dimensional space of weights associated with modern DNNs; moreover, the use of minibatches allows one to exploit modern GPUs and to achieve a considerable computational efficiency.

It is well known that SGD uses a number of hyper-parameters that are usually very hard to optimize, as they depend on the algorithm and on the underlying dataset. Hyper-parameter search is commonly performed manually, via rules-of-thumb or by testing sets of hyper-parameters on a predefined grid [2]. In SGD, momentum [13] or Nesterov [8] are widely recognized to increase the speed of convergence. Instead, effective learning rates are highly dependent on DNN architecture and on the

*corresponding author: http://www.dei.unipd.it/~fisch

Preprint. Under review.

dataset of interest [12], so they are typically selected on a best-practice basis, although methods such as CLR [11] have been proposed to reduce the number of choices. Finally, [1] shows empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid.

In the present paper we investigate the use of an alternative training method borrowed from Mathematical Optimization, namely, the Simulated Annealing (SA) algorithm [4]. The use of SA for training is not new, but previous proposals are mainly intended to be applied for non-differentiable objective functions for which SGD is not applied due to the lack of gradients; see, e.g., [9, 6]. Instead, our SA method requires differentiability of (a proxy of) the loss function, and leverages on the availability of a gradient direction to define local moves that have a large probability to improve the current solution.

A notable application of our approach is in the context of SGD hyper-parameter tuning—hence the title of the paper. Assume all possible hyper-parameter values (e.g., learning rates for SGD) are collected in a discrete set H. At each SGD iteration, we randomly pick one hyper-parameter from H, temporarily implement the corresponding *move* as in the classical SGD method (using the gradient information) and evaluate the new point on the current minibatch. If the loss function does not deteriorate too much, we *accept* the move as in the classical SGD method, otherwise we *reject* it: we step back to the previous point, change the minibatch, randomly pick another hyper-parameter from H, and repeat. The decision of accepting/rejecting a move is based on the classical SA criterion, and depends of the amount of loss-function worsening and on a certain parameter (the *temperature*) which is modified over time to lower the probability of accepting worsening moves.

A distinctive feature of our scheme is that hyper-parameters are modified within a *single* SGD execution (and not in an external loop, as customary) and evaluated on the fly on the current minibatch, i.e., their tuning is fully embedded within the SGD algorithm.

Computational results are reported, showing that the proposed approach leads to an improvement of the final validation accuracy for modern DNN architectures (ResNet34 and VGG16 on CIFAR-10). Also, it turns out that the random seed used within our algorithm modifies the search path in a very significant way, allowing one to use this seed as a single hyper-parameter to be tuned in an external loop for a further improved generalization.

2 Simulated Annealing

The basic SA algorithm for a generic optimization problem can be outlined as follows. Let S be the set of all possible feasible solutions, and $f : S \to \mathbb{R}$ be the objective function to be minimized. An optimal solution s^* is a solution in S such that $f(s^*) \leq f(s)$ holds for all $s \in S$.

SA is an iterative method that constructs a trajectory of solutions $s^{(0)}, \cdots, s^{(k)}$ in S. At each iteration, SA considers moving from the current feasible solution $s^{(i)}$ (say) to a candidate new feasible solution s_{new} (say). Let $\Delta(s^{(i)}, s_{new}) = f(s_{new}) - f(s^{(i)})$ be the objective function *worsening* when moving from $s^{(i)}$ to s_{new}—positive if s_{new} is strictly worse than $s^{(i)}$. The hallmark of SA is that worsening moves are not forbidden but accepted with a certain *acceptance probability* $p(s^{(i)}, s_{new}, T)$ that depends on the amount of worsening $\Delta(s^{(i)}, s_{new})$ and on a parameter $T > 0$ called *temperature*. A typical way to compute the acceptance probability is through *Metropolis' formula* [7]:

$$p(s, s_{new}, T) = \begin{cases} e^{-\Delta(s^{(i)}, s_{new})/T} & \text{if} \quad \Delta(s^{(i)}, s_{new}) > 0 \\ 1 & \text{if} \quad \Delta(s^{(i)}, s_{new}) \leq 0 \, . \end{cases} \tag{1}$$

Thus, the probability of accepting a worsening move is large if the amount of worsening $\Delta(s^{(i)}, s') > 0$ is small and the temperature T is large. Note that the probability is 1 when $\Delta(s^{(i)}, s') \leq 0$, meaning that improving moves are always accepted by the SA method.

Temperature T is a crucial parameter: it is initialized to a certain value T_0 (say), and iteratively decreased during the SA execution so as to make worsening moves less and less likely in the final iterations. A simple update formula for T is based on a *cooling factor* $\alpha \in (0, 1)$ and reads

$$T = \alpha \cdot T \, ; \tag{2}$$

typical ranges for α are $0.95 - 0.99$ (if cooling is applied at each SA iteration) or $0.7 - 0.8$ (if cooling is only applied at the end of a "computational epoch", i.e., after several SA iterations with a constant temperature).

The basic SA scheme is outlined in Algorithm 1; more advanced implementations are possible, e.g., the temperature can be restored multiple times to the initial value.

Algorithm 1 : SA

Input: function f to be minimized, initial temperature $T_0 > 0$, cooling factor $\alpha \in (0, 1)$, number of iterations $nIter$
Output: the very last solution $s^{(nIter)}$

1: Compute an initial solution $s^{(0)}$ and initialize $T = T_0$
2: **for** $i = 0, \ldots, nIter - 1$ **do**
3: Pick a new tentative solution s_{new} in a convenient neighborhood $\mathcal{N}(s^{(i)})$ of $s^{(i)}$
4: $worsening = f(s_{new}) - f(s^{(i)})$
5: $prob = e^{-worsening/T}$
6: **if** $random(0, 1) < prob$ **then**
7: $s^{(i+1)} = s_{new}$
8: **else**
9: $s^{(i+1)} = s^{(i)}$
10: **end if**
11: $T = \alpha \cdot T$
12: **end for**

At Step 6, $random(0, 1)$ is a pseudo-random value uniformly distributed in [0,1]. Note that, at Step 5, the acceptance probability $prob$ becomes larger than 1 in case $worsening < 0$, meaning that improving moves are always accepted (as required).

2.1 A naive implementation for training without gradients

In the context of training, one is interested in minimizing a *loss function* $L(w)$ with respect to a large-dimensional vector $w \in \Re^M$ of so-called *weights*. If $L(w)$ is differentiable (which is not required by the SA algorithm), there exists a gradient $\nabla(w)$ giving the steepest increasing direction of L when moving from a given point w.

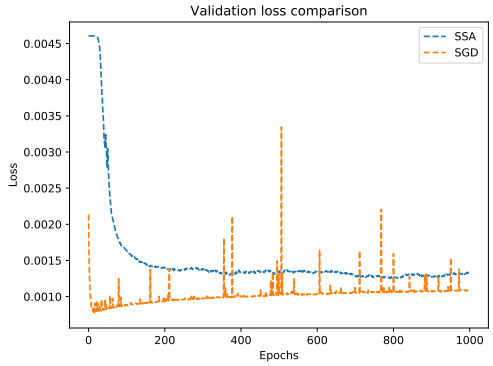
Here is a very first attempt to use SA in this setting. Given the current solution (i.e., set of weights) w, we generate a random move $\Delta(w) \in \Re^M$ and then we evaluate the loss function in the nearby point $w' := w - \epsilon\Delta(w)$, where ϵ is a small positive real number. If the norm of $\epsilon\Delta(w)$ is small enough and L is differentiable, due to Taylor's approximation we know that

$$L(w') \simeq L(w) - \epsilon \, \nabla^T(w)\Delta(w) \, . \tag{3}$$

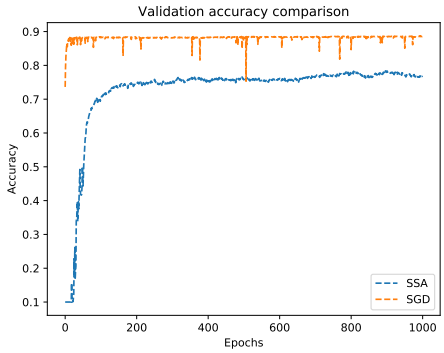
Thus the objective function improves if $\nabla(w)^T\Delta(w) > 0$. As we work in the continuous space, in the attempt of improving the objective function we can also try to move in the opposite direction and move to $w'' := w + \epsilon \, \Delta(w)$. Thus, our actual move from the current w consists of picking the best (in terms of objective function) point w_{new}, say, between the two nearby points w' and w'': if w_{new} improves $L(w)$, then we surely accept this move; otherwise we accept it according to the Metropolis' formula (1). Note that the above SA approach is completely derivative free: as a matter of fact, SA could optimize directly over discrete functions such as the accuracy in the context of classification.

In a preliminary work we implemented the simple scheme above in a stochastic manner, using minibatches when evaluating $L(w')$ and $L(w'')$, very much in the spirit of the SGD algorithm. Figure 1 compares the performance of the resulting Stochastic SA algorithm, called `SSA`, with that of a straightforward SGD implementation with constant learning rate and no momentum/Nesterov acceleration, using the Fashion-MNIST [14] dataset and the VGG16 [10] architecture. Figure 1(d) reports accuracy on both the training and the validation sets, showing that `SSA` does not suffer from overfitting as the accuracy on the training and validation sets are almost identical—a benefit deriving from the derivative-free nature of `SSA`. However, `SSA` is clearly unsatisfactory in terms of validation accuracy (which is much worse than the SGD one) in that it does not exploit well the VGG16 capacity.

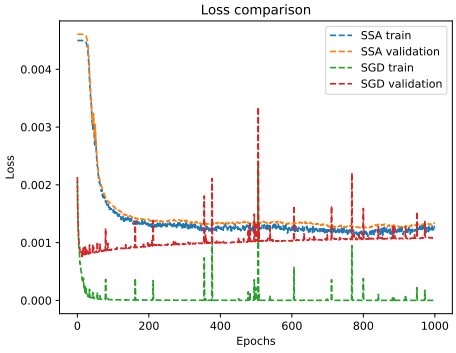
We are confident that the above results could be improved by a more advanced implementation. E.g., one could vary the value of ϵ during the algorithm, and/or replace the loss function by (one minus) the accuracy evaluated on the current minibatch—recall that `SSA` does not require the objective function be differentiable. However, even an improved `SSA` implementation is unlikely to be competitive with



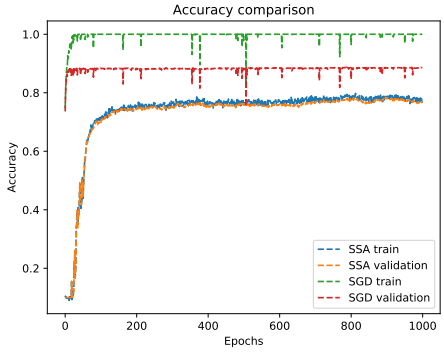
(a) Validation loss



(b) Validation accuracy



(c) Loss comparison



(d) Accuracy comparison

Figure 1: Naive SA implementation for VGG16 on Fashion-MNIST. SGD: learning rate $\eta = 0.001$, no momentum/Nesterov acceleration. `SSA`: $\epsilon = 0.01$, $\alpha = 0.97$, $T_0 = 1$. Subfigure (d) clearly shows that `SSA` has no overfitting but is not able to exploit well the capacity of VGG16, resulting into an unsatisfactory final accuracy.

SGD. In our view, the main drawback of the `SSA` algorithm (as stated) is that, due the very large dimensional space, the random direction $\pm\Delta(w)$ is very unlikely to lead to a substantial improvement of the objective function as the effect of its components tend to cancel out randomly. Thus, a more clever definition of the basic move is needed to drive `SSA` in an effective way.

3 Improved SGD training by SA

We next introduce an unconventional way of using SA in the context of training. We assume the function $L(w)$ to be minimized be differentiable, so we can compute its gradient $\nabla(w)$. From SGD we borrow the idea of moving in the anti-gradient direction $-\nabla(w)$, possibly corrected using momentum/Nesterov acceleration techniques. Instead of using a certain *a priori* learning rate η, however, we randomly pick one from a discrete set H (say) of possible candidates. In other words, at each SA iteration the move is selected randomly in a discrete neighborhood $\mathcal{N}(w^{(i)})$ whose elements correspond to SGD iterations with different learning rates. An important feature of our method is that H can (actually, should) contain unusually large learning rates, as the corresponding moves can be discarded by the Metropolis' criterion if they deteriorate the objective function too much.

A possible interpretation of our approach is in the context of SGD hyper-parameter tuning. According to our proposal, hyper-parameters are collected in a discrete set H and sampled *within a single SGD execution*: in our tests, H just contains a number of possible learning rates, but it could involve other parameters/decisions as well, e.g., applying momentum, or Nesterov (or none of the two) at the current SGD iteration, or alike. The key property here is that any element in H corresponds to a reasonable (non completely random) move, so picking one of them at random has a significant

probability of improving the objective function. As usual, moves are accepted according to the Metropolis' criterion, so the set H can also contain "risky choices" that would be highly inefficient if applied systematically within a whole training epoch.

Algorithm 2 : `SGD-SA`

Parameters: A set of learning rates H, initial temperature $T_0 > 0$
Input: Differentiable loss function L to be minimized, cooling factor $\alpha \in (0, 1)$, number of epochs $nEpochs$, number of minibatches N
Output: the best performing $\mathrm{w}^{(i)}$ on the validation set at the end of each epoch

1: Divide the training dataset into N minibatches
2: Initialize $i = 0,\ T = T_0,\ w^{(0)}$ = random_initialization()
3: **for** $t = 1, \ldots, nEpochs$ **do**
4: **for** $n = 1, \ldots, N$ **do**
5: Extract the n-th minibatch (x, y)
6: Compute $L(w^{(i)}, x, y)$ and its gradient $v =$ backpropagation$(w^{(i)}, x, y)$
7: Randomly pick a learning rate η from H
8: $w_{new} = w^{(i)} - \eta\, v$
9: Compute $L(w_{new}, x, y)$
10: $worsening = L(w_{new}, x, y) - L(w^{(i)}, x, y)$
11: $prob = e^{-worsening/T}$
12: **if** $random(0, 1) < prob$ **then**
13: $w^{(i+1)} = w_{new}$
14: **else**
15: $w^{(i+1)} = w^{(i)}$
16: **end if**
17: $i = i + 1$
18: **end for**
19: $T = \alpha \cdot T$
20: **end for**

Our basic approach is formalized in Algorithm 2, and will be later referred to as `SGD-SA`. More elaborated versions using momentum/Nesterov are also possible but not investigated in the present paper, as we aim at keeping the overall computational setting as simple and clean as possible.

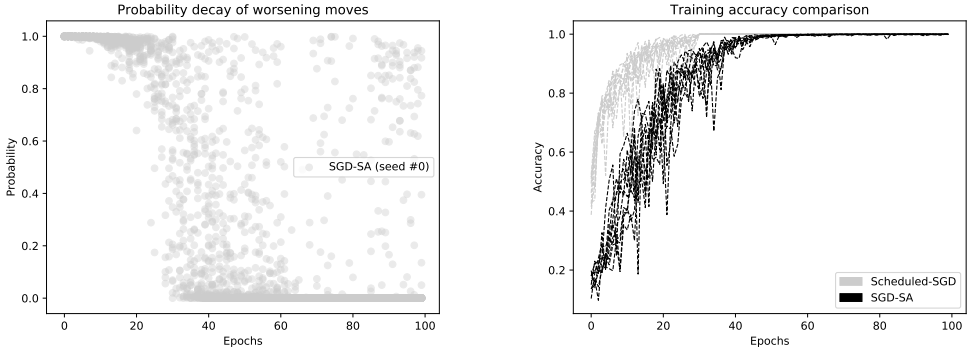
4 Computational analysis of `SGD-SA`

We next report a computational comparison of SGD and `SGD-SA` for a classical image classification task involving the CIFAR-10 [5] dataset. As customary, the dataset was shuffled and partitioned into 50,000 examples for the training set, and the remaining 10,000 for the test set. As to the DNN architecture, we tested two well-known proposals from the literature: VGG16 [10] and ResNet34 [3]. Training was performed for 100 epochs using PyTorch, with minibatch size 512. Tests have been performed using a single NVIDIA TITAN Xp GPU.

Our `Scheduled-SGD` implementation of SGD is quite basic but still rather effective on our dataset: it uses no momentum/Nesterov acceleration, and the learning rate is set according the following schedule: $\eta = 0.1$ for first 30 epochs, 0.01 for the next 40 epochs, and 0.001 for the final 30 epochs. As to `SGD-SA`, we used $\alpha = 0.8$, initial temperature $T_0 = 1$, and learning-rate set $H = \{0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.09, 0.08, 0.07, 0.06, 0.05\}$.

Both `Scheduled-SGD` and `SGD-SA` use pseudo-random numbers generated from an initial random seed, which therefore has some effects of the search path in the weight space and hence on the final solution found. Due to the very large number of weights that lead to statistical compensation effects, the impact of the seed on the initialization of the very first solution $w^{(0)}$ is very limited—a property already known for SGD that is inherited by `SGD-SA` as well. However, random numbers are used by `SGD-SA` also when taking some crucial "discrete" decisions, namely: the selection of the learning rate $\eta \in H$ (Step 7) and the acceptance test (Step 12). As a result, as shown next, the search path of `SGD-SA` is very dependent on the initial seed. Therefore, for both `Scheduled-SGD` and `SGD-SA` we decided to repeat each run 10 times, starting with 10 random seeds, and to report results for each

seed. In our view, this dependency on the seed is in fact a *positive* feature of `SGD-SA`, in that it allows one to treat the seed as a single (quite powerful) hyper-parameter to be randomly tuned in an external loop.



(a) Probability of accepting worsening moves

(b) Training accuracy

Figure 2: Optimization efficiency over the training set (VGG16 on CIFAR-10)

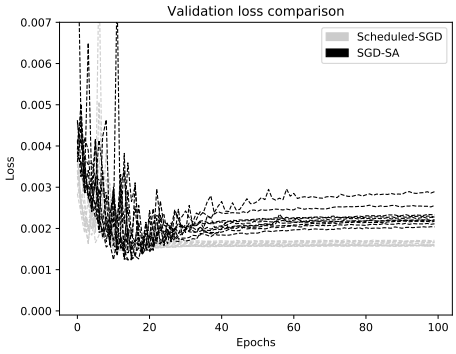
Our first order of business is to evaluate the convergence property of `SGD-SA` on the training set—after all, this is the optimization task that SA faces directly. In Figure 2 we plot the average probability $prob$ (clipped to 1) of accepting a move at Step 12, as well as the training-set accuracy as a function of the epochs. Subfigure 2a shows that the probability of accepting a move is almost one in the first epochs, even if the amount of worsening is typically quite large in this phase. Later on, the probability becomes smaller and smaller, and only very small worsenings are more likely to be accepted. As a result, large learning rates are automatically discarded in the last iterations. Subfigure 2b is quite interesting: even in our simple implementation, `Scheduled-SGD` quickly converges to the best-possible value of one for accuracy, and the plots for the various seeds (gray lines) are almost overlapping—thus confirming that the random seed has negligible effects of `Scheduled-SGD`. As to `SGD-SA` (black lines), its convergence to accuracy one is slower than `Scheduled-SGD`, and different seeds lead to substantially different curves—a consequence of the discrete random decisions taken along the search path.

Plots in Figures 3 and 4 show the performance (on both the training and validation sets) of `Scheduled-SGD` and `SGD-SA` when using the ResNet34 and VGG16 architectures, respectively. As expected, the search path of `SGD-SA` is more diversified (leading to accuracy drops in the first epochs) abut the final solutions tend to generalize better than `Scheduled-SGD` (as witnessed by the performance on the validation set).

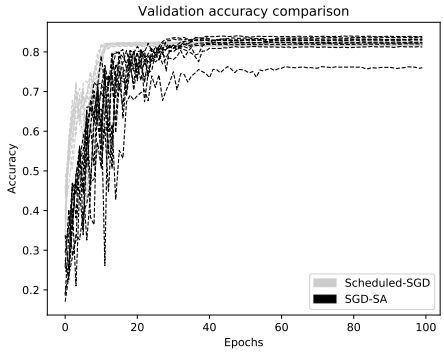
Table 1 gives more detailed results for each seed, and reports the final validation accuracy and loss reached by `Scheduled-SGD` and `SGD-SA`. The results show that, for all seeds, `SGD-SA` always produces a significantly better (lower) validation loss than `Scheduled-SGD`. As to validation accuracy, `SGD-SA` outperforms `Scheduled-SGD` for all seeds but seeds 3, 4 and 6 for ResNet34. In particular, `SGD-SA` leads to a significantly better (1-2%) validation accuracy than `Scheduled-SGD` if the best run for the 10 seeds is considered.

Method	Seed	VGG16		ResNet34	
		Loss	Accuracy	Loss	Accuracy
Scheduled-SGD	0	0.001640	85.27	0.001519	82.18
	1	0.001564	84.94	0.001472	82.58
	2	0.001642	84.84	0.001467	82.27
	3	0.001662	84.93	0.001468	82.37
	4	0.001628	84.92	0.001602	81.69
	5	0.001677	85.37	0.001558	81.80
	6	0.001505	84.91	0.001480	82.24
	7	0.001480	85.28	0.001532	82.07
	8	0.001623	85.26	0.001574	81.52
	9	0.001680	85.41	0.001499	82.41
`SGD-SA`	0	0.001127	86.44	0.001306	82.55
	1	0.001206	86.18	0.001231	84.11
	2	0.001121	86.04	0.001238	83.32
	3	0.001133	86.76	0.001457	81.39
	4	0.001278	85.17	0.001585	76.31
	5	0.001112	86.30	0.001276	83.74
	6	0.001233	85.71	0.001405	82.07
	7	0.001130	86.59	0.001261	82.57
	8	0.001167	86.14	0.001407	83.12
	9	0.001084	86.28	0.001240	83.19
Best `Scheduled-SGD`		0.001480	85.41	0.001467	82.58
Best `SGD-SA`		**0.001084**	**86.76**	**0.001240**	**84.11**

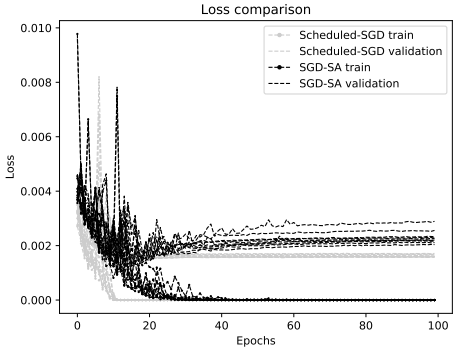
Table 1: Final validation accuracy and loss, seed by seed.



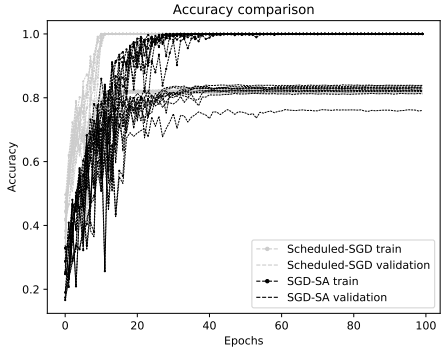
(a) Validation loss



(b) Validation accuracy

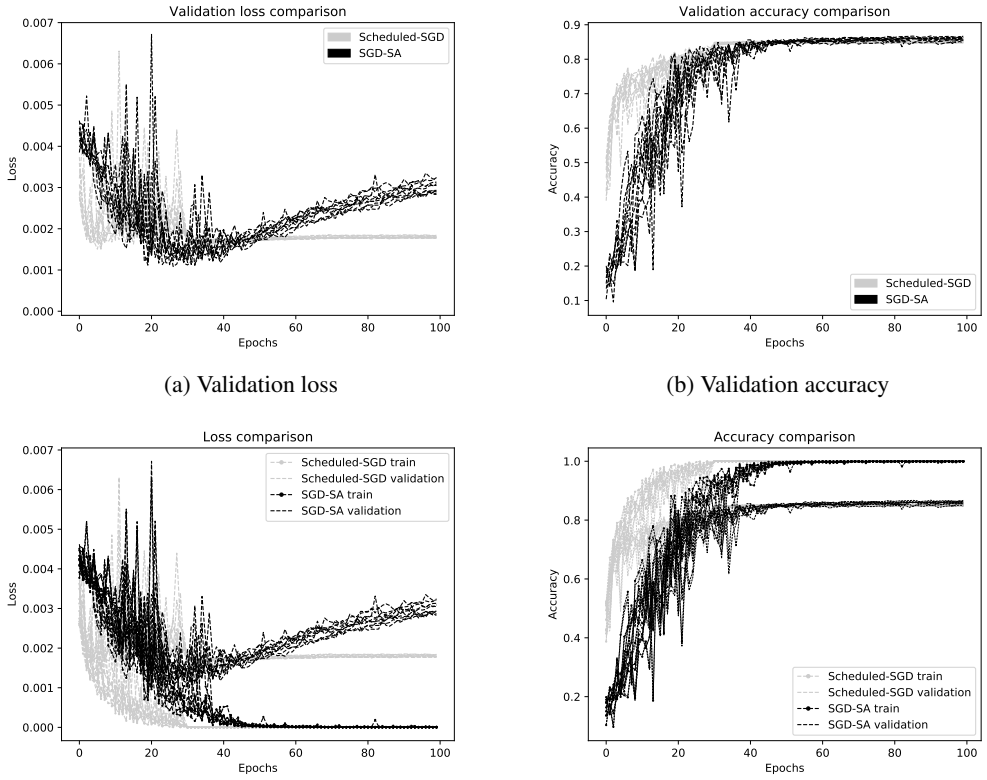


(c) Loss comparison



(d) Accuracy comparison

Figure 3: ResNet34 on CIFAR-10



(a) Validation loss

(b) Validation accuracy

(c) Loss comparison

(d) Accuracy comparison

Figure 4: VGG16 on CIFAR-10

5 Conclusions and future work

We have proposed a new metaheuristic training scheme that combines Stochastic Gradient Descent and Discrete Optimization in an unconventional way.

Our idea is to define a discrete neighborhood of the current solution containing a number of "potentially good moves" that exploit gradient information, and to search this neighborhood by using a classical metaheuristic scheme borrowed from Discrete Optimization. In the present paper, we have investigated the use of a simple Simulated Annealing metaheuristic that accepts/rejects a candidate new solution in the neighborhood with a probability that depends both on the new solution quality and on a parameter (the temperature) which is varied over time. We have used this scheme as an automatic way to perform hyper-parameter tuning within a single training execution, and have shown its potentials on a classical test problem (CIFAR-10 image classification using VGG16/ResNet34 deep neural networks).

In a follow-up research we plan to investigate the use of two different objective functions at training time: one differentiable to compute the gradient (and hence a set of potentially good moves), and one completely generic (possibly black-box) for the Simulated Annealing acceptance/rejection test—the latter intended to favor simple/robust solutions that are likely to generalize well.

Replacing Simulated Annealing with other Discrete Optimization metaheuristics (tabu search, variable neighborhood search, genetic algorithms, etc.) is also an interesting topic that deserves future research.

Acknowledgments

Work supported by MiUR, Italy (project PRIN 2015 on "Nonlinear and Combinatorial Aspects of Complex Networks"). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

References

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[2] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, December 2015.

[4] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

[5] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 (Canadian Institute for Advanced Research). `http://www.cs.toronto.edu/~kriz/cifar.html`.

[6] Sergio Ledesma, Miguel Torres, Donato Hernández, Gabriel Aviña, and Guadalupe García. Temperature cycling on simulated annealing for neural network learning. In Alexander Gelbukh and Ángel Fernando Kuri Morales, editors, *MICAI 2007: Advances in Artificial Intelligence*, pages 161–171, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Nicholas Constantine Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[8] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[9] Randall Sexton, Robert Dorsey, and John Johnson. Beyond backpropagation: Using simulated annealing for training neural networks. *Journal of End User Computing*, 11, 07 1999.

[10] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv e-prints*, September 2014.

[11] Leslie N. Smith. Cyclical learning rates for training neural networks, 2015. cite arxiv:1506.01186Comment: Presented at WACV 2017; see https://github.com/bckenstler/CLR for instructions to implement CLR in Keras.

[12] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, abs/1803.09820, 2018.

[13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[14] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.