TRAINABLE LEARNING RATE

Anonymous authors
Paper under double-blind review

ABSTRACT

Selecting an appropriate learning rate for efficiently training deep neural networks is a difficult process that can be affected by numerous parameters, such as the dataset, the model architecture or even the batch size. In this work, we propose an algorithm for automatically adjusting the learning rate during gradient descent. The rationale behind our approach is to train the learning rate along with the model weights, akin to line-search. Learning rate is optimized via a simple extra gradient descent step, justified by an analysis that exploits the structure of neural networks. We formulate first and second-order gradients with respect to learning rate as functions of consecutive weight gradients, leading to a cost-effective implementation. We also show that the scheme can be extended to accommodate for different learning rates per layer. Extensive experimental evaluation is conducted, validating the effectiveness of the proposed method for a plethora of different settings. The proposed method has proven to be robust to both the initial learning rate and the batch size, making it ideal for an off-the-shelf optimizing scheme.

1 INTRODUCTION

Deep learning ushered in a fascinating era where artificial intelligence applications become ever-increasingly more prevalent in every imaginable aspect of everyday life. Despite their success, an important practical shortcoming is that at the core of deep learning lies a very difficult computational optimization problem. Specifically, training requires optimizing a non-convex loss over a number of parameters that can easily reach the order of millions, finding a global optimum over which is known to be an NP-hard problem (Sun, 2016). Due to their complex loss structure, existing algorithms aim to discover well-performing local minima according to one of several gradient-based optimization schemes. The most characteristic example is the Stochastic Gradient Descent (SGD) algorithm.

Arguably, SGD is still widely used for training deep neural networks, decades after its inception. The efficiency of SGD is further supported by recent theoretical developments concerning its stability, as well as its convergence speed in the context of deep learning (Hardt et al., 2016; Allen-Zhu et al., 2019). Its simple rationale is to choose an improvement over current parameters along the path defined by the loss gradient (itself computed as an estimate over a minibatch of samples). Other gradient-based methods such as Adam or momentum-based methods share the logic of first choosing and then moving along a “good” search direction that is to be understood as an improvement over the direction defined by the gradient. The choice of the step-size (“learning rate”) is arguably perhaps the weakest point of these methods, and their performance is known to rely heavily on its choice. An inappropriate choice of learning rate value or learning rate schedule can easily lead to a suboptimal local minimum, leading in practice to inferior network efficiency.

To the end of dealing with the choice of learning rate, several works have explored whether a schedule that leads to theoretical guarantees can be obtained; we know that a constant step-size leads to convergence to a neighbourhood of the solution, and using a decreasing step-size can guarantee convergence to an exact optimum (Loizou et al., 2021). Alternate ways of dealing with choosing learning rate include using backtracking line-search methods (Nocedal & Wright, 2006; Royer & Wright, 2018; Vaswani et al., 2019). Line-search aims to optimize the section of the function that is defined by the current descent direction per iteration, and backtracking line-search provides a framework within which the current section can be optimized at the cost of a number of function evaluations. The function may be optimized either exactly or at least ensure a sufficient drop on the function value, subject to favourable constraints (Armijo-Goldstein conditions). However, these approches may entail requiring multiple function and gradient computations per search direction iterate, which translate to multiplying the total computational load.

In a nutshell, in this work we propose a method that deals with choosing the learning rate by treating it as a trainable parameter, motivated by line-search approaches. We build our method in the context of gradient descent for optimizing a neural network loss function. Contrary to standard backtracking line-search algorithms, which typically perform several steps within the same search direction using the backtracking algorithm with the Armijo-Goldstein conditions (Vaswani et al., 2019), we perform only a single "correction" step towards a better learning rate in a gradient descent scheme. The major advantage of our approach lies on the minimal computational requirements of such an update.

Our contribution can be summarized with respect to the following points: a) We show that we can construct a gradient-based method that requires neither specifying a fixed learning rate step, nor performing several search steps per line-search. Instead, *the learning rate is itself cast as a trainable parameter*. No manual tuning is required, and a simple update per iteration is used. The proposed update employs an explicit derivation of gradients with respect to learning rate. While related interpretations and formulations concerning meta-optimization and first-order gradient properties have partly appeared in earlier (Almeida et al., 1999; Schraudolph, 1999) and more recent works (Baydin et al., 2018; Metz et al., 2019), we introduce a novel second-order gradient derivation and analysis (following a Newton-Raphson rationale) and use both to motivate our method. The proposed second-order derivation leads to a simple formula of consecutive weight gradients, implemented as a constant-time operation on the backward pass, avoiding the computationally intensive and impractical Hessian derivation/approximation (w.r.t. weights), typically used in such settings. b) The proposed algorithm displays notable robustness to user-defined hyper-parameters of the training process, such as initial learning rate and batch size. c) Implementation-wise the algorithm can be coded very simply, it is cost-efficient, and does not require any critical hyperparameter to be set. d) We may naturally treat each layer separately, with a different trainable learning rate per layer.

2 Related Work

Treating step-sizes holistically as a decreasing series is a classic approach, which can conceptually intertwine with adaptive method principles. Since Robbins and Monro had proposed a step-size / learning rate schedule that should follow $\alpha_t = O(t^{-1})$, several works have explored and built on this idea (Gower et al., 2019; Schaul et al., 2013; Li & Orabona, 2019; Malitsky & Mishchenko, 2020). Popular schedule examples include cosine annealing (Loshchilov & Hutter, 2017), where learning rate is set to initially mildly decrease, drop fast then again mildly decrease again, (hopefully) when we are near a local minimum. Step-size can be chosen to adapt to measures that include current loss gradients, running statistics of the loss and its gradients, or other adaptive parameters. Polyak step-sizes propose a step-size that is proportional to the current difference to the global minimum, and favourable theoretical results exist concerning their speed (Barré et al., 2020; Loizou et al., 2021). Another notable adaptive method is Barzilai-Borwein, where a complete theory has been put forward (Barzilai & Borwein, 1988). Barzilai-Borwein is motivated by Quasi-Newton methods, and proved to work well in quadratic problems. Adam is a recent important algorithm in the genre of adaptive gradient methods, closely related to other popular algorithms such as RMSProp and Adagrad (Kingma & Ba, 2015; Duchi et al., 2011; Tieleman & Hinton, 2012). The search direction is adapted according to local geometry estimates that are computed as moving averages. Ideas from momentum-based methods are also integrated (Nesterov, 1983; Polyak, 1964). Collectively, the aforementioned methods share either one or all of these disadvantages: first, the proposed schedule is based on hyperparameters that themselves are not often straightforward to tune, or knowledge of information on the function, like the value of its global optimum (Polyak, 1964), the strong convexity parameter (O'Donoghue & Candes, 2015) or Hölderian error bounds (Barré et al., 2020); second, they can be difficult to use in practice; third, while these methods often come with a theoretical justification of the proposed schedule, there is still no guarantee for fast convergence for either of these methods, except for at best very tight constraints (Vaswani et al., 2019; Barré et al., 2020; Malitsky & Mishchenko, 2020).

Adaptive methods on a non-convex setting are markedly less studied, and their stochastic counterparts (used in practice) even more so (Almeida et al., 1999; Kresoja et al., 2017; Li & Orabona, 2019; Pesme et al., 2020). Inspired by batch normalization, a learning rate schedule that can start at a high value then adapt to current batch gradient is proposed in Wu et al. (2018a). The same rule is shown to behave well under both batch and stochastic settings. In Vaswani et al. (2019), Armijo line-search is used to determine the step-size in each SGD iterate. The method is used for over-parameterized models that satisfy the interpolation condition, and fast rates of convergence are demonstrated. Heuristics are proposed to use larger step-sizes. In Schaul et al. (2013), they use coordinate-wise adaptive learning rates that are constrained to maximize the decrease of the expected loss, assuming that the loss breaks down as separable quadratic functions. Variance reduction methods modify standard SGD intending to improve its convergence rate, albeit at the cost of not insignificant computational cost. Also, in an over-parameterized setting their fast convergence premise is practically not achieved (Defazio & Bottou, 2019; Vaswani et al., 2019).

Methods that tweak the learning rate based on the relation between the current and previous gradient(s), similarly to our approach, can be dated as far back as the work of Kesten (1958). Therein, it is proposed that the learning rate should increase if consecutive weight differences have the same sign, or decrease if they have opposite signs. Note also that in the same work they assume parameter-wise learning rates instead of a single global rate. A number of variations on this idea have been proposed (Saridis, 1970; Pflug, 1983; Sutton, 1986) before the Delta-Bar-Delta (DBD) method (Jacobs, 1988), which introduced heuristics based on the sign of the current and the exponential average over the previous gradients. Specifically, in the DBD method the learning rate is either increased by a constant, decreased by a percent of its current value or remain the same. Numerous related works have followed; for example, Minai & Williams (1990) presented an extended DBD, which integrated various extensions to the main algorithm, such as momentum, increasing learning rate by non-constant values, and enforcing a rate upper bound. Almeida et al. (1997, 1999) have experimented with finding correct heuristics in a stochastic setting. Following this line of work, Pesme et al. (2020) utilize a test based on the inner product of consecutive gradients to distinguish two phases during training: a transient phase during which iterates make fast progress towards the optimum, followed by a stationary phase during which iterates oscillate around the optimal point. In the recent work of Baydin et al. (2018), (first-order) gradients are computed over the learning rate and an update rule is defined using consecutive weight gradients. The authors dub this gradient a "hypergradient" and use it to update the learning rate as $\Delta\alpha_t = -\beta \nabla\mathcal{L}(\mathbf{w}_t)^\intercal \nabla\mathcal{L}(\mathbf{w}_{t-1})$ where β is the hypergradient learning rate. As we discuss elsewhere in this paper (sec. 3, 5, appendix D.4) such a meta-learning rate parameter can be difficult to set or tune. In (Metz et al., 2019), they analyze the gradient w.r.t. the learning rate in terms of previous T time-steps. Setting the number of unrolled steps to 1, they obtain the gradient as the negative inner product of the current and previous iteration.

3 Trainable Learning Rate

In this section, we propose and discuss two variants of an algorithm for automatically adjusting learning rate at each training iteration. The main idea relies on the following observation: Given the weights' gradients, one could estimate if an overestimation or an underestimation of learning rate could further reduce the task loss by casting the learning rate as an extra trainable parameter.

Problem Statement: The Gradient Descent (GD) framework. We develop our algorithm within the framework of gradient descent. Consider the GD update rule in its vanilla form:

$$w_{t+1} = w_t - \alpha \cdot \nabla\mathcal{L}(w_t) \tag{1}$$

where w_t are the model parameters at iteration t, $\mathcal{L}$ is the loss function and α is the learning rate hyperparameter. In vanilla GD, α is treated as a hyperparameter and does not contribute to the loss. We introduce an augmented loss term $\mathcal{L}_\alpha$, where we consider α as the (learnable) variable:

$$\mathcal{L}_\alpha(\alpha; w) = \mathcal{L}(w - \alpha \cdot \nabla\mathcal{L}(w)) \tag{2}$$

Intuitively, we can interpret $\mathcal{L}_\alpha$ as the loss value that we would have obtained if we followed a GD step of step-size α given parameter values w. Therefore, minimizing $\mathcal{L}_\alpha$ is equivalent to finding the optimal step-size for current w.

Algorithm variant 1: Naïve GD-TLR. A straightforward idea is to apply GD over $\mathcal{L}_\alpha$:

$$\alpha_{t+1} = \alpha_t - \eta \cdot \nabla\mathcal{L}_\alpha(\alpha_t; w_t) \tag{3}$$

where hyperparameter η controls the updates of the initial learning rate α.

Algorithm 1 Naïve GD-TLR

```
Input: hyperparameter $\eta$, number of iterations $T$,
    initial weights $w_0$, initial learning rate $\alpha_0$
Output: optimized weights $w$
 1: Initialize $\mathbf{w}$ and $\alpha$ as $\mathbf{w}_0$ and $\alpha_0$
 2: for $t = 0$ to $T - 1$ do
 3:     First Pass: Compute $\nabla\mathcal{L}(w_t)$
 4:     Second Pass: Compute $\nabla\mathcal{L}_\alpha(\alpha_t; w_t)$
 5:     $\alpha_{t+1} \leftarrow \alpha_t - \eta \cdot \nabla\mathcal{L}_\alpha(\alpha_t; w_t)$
 6:     $w_{t+1} \leftarrow w_t - \alpha_{t+1} \cdot \nabla\mathcal{L}(w_t)$
 7: end for
```

Algorithm 2 Efficient GD-TLR

```
Input: number of iterations $T$, initial weights $w_0$, initial
    learning rate $\alpha_0$, hyperparameter $c$
Output: optimized weights $\mathbf{w}$
 1: Initialize $\mathbf{w}$ and $\alpha$ as $\mathbf{w}_0$ and $\alpha_0$
 2: Initialize $\mathbf{g}_{-1} = 0$
 3: for $t = 0$ to $T - 1$ do
 4:     Single Forward-Backward Pass:
        Compute $\mathbf{g}_t = \nabla\mathcal{L}(\mathbf{w}_t)$
 5:     Update $\alpha$ according to Eqs. 7 & 9:
        $\alpha_{t+1} \leftarrow \alpha_t\Big(1 + \min\big(\frac{\langle \mathbf{g}_t, \mathbf{g}_{t-1}\rangle}{\max(4\langle \mathbf{g}_t, \mathbf{g}_t - \mathbf{g}_{t-1}\rangle, 0)}, c\big)\Big)$
 6:     Update $\mathbf{w}$ according to Eq. 1:
        $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha_{t+1} \cdot \nabla\mathcal{L}(\mathbf{w}_t)$
 7:     Store $\mathbf{g}_t$ as $\mathbf{g}_{t-1}$
 8: end for
```

After having calculated gradients using back-propagation, we can update the weights along with the learning rate according to equations 1 and 2 respectively. Computation-wise, compared to vanilla gradient descent this algorithm requires an extra forward-backward pass per step in order to calculate the learning rate gradient. This algorithm is summarized in Algorithm 1.

Algorithm variant 2: Efficient GD-TLR. One weakness of the *Naïve* variant is that the addition of α requires an extra forward-backward pass in order to compute $\nabla\mathcal{L}_\alpha(\alpha_t; w_t)$. Also, the introduction of a GD step for α creates the extra hyperparameter η; it is desirable to consider dispensing with having to rely on this "meta"-learning rate. In the *Efficient* algorithm variant, we address both of these issues. Before we see how we can achieve these improvements, we need to compute and discuss augmented loss gradients $\nabla\mathcal{L}_\alpha$ and $\nabla^2\mathcal{L}_\alpha$, as their properties play a pivotal role in this direction.

First-Order Gradient and Insight: We make use of the structure of loss $\mathcal{L}$ and $\mathcal{L}_\alpha$ as functions over a neural network. Neural networks, in all their diversity of architectures, have structural traits in common. Specifically, neural networks consist of a cascade of non-linearities and linearities, with linear units being associated with the model parameters. For instance, a standard feed-forward network of N layers can be written as $\mathbf{y} = \phi^N \circ L^N \circ \phi^{N-1} \circ L^{N-1} \circ \ldots \phi^1 \circ L^1(\mathbf{x})$, with ϕ^k and L^k denoting the non-linear activation and linearity corresponding to layer k respectively. Linearities can be written in the simple form $L^k(\mathbf{x}) = \mathbf{W}^k\mathbf{x}$. [1] This expression applies not only for dense layers but for any layer as long as a linear relation holds; for example, convolution is well-known to be linear, as well as expressible as a matrix multiplication using circulant matrices (Jain, 1989; Sedghi et al., 2019). For our analysis, we consider the linearity of a *single layer* $\mathbf{y} = \mathbf{W}\mathbf{x}$, where $\mathbf{x}$ is the input coming from the previous layer, $\mathbf{W}$ is the matrix of parameters, $\mathbf{y}$ is the output, and we have dropped layer indices for clarity. Hat ($\hat{\mathbf{W}}, \hat{\mathbf{y}}$) denotes variables after a GD update (eq. 1). We can then write the updated $\hat{\mathbf{y}}$ as:

$$\hat{\mathbf{y}} = (\mathbf{W} - \alpha \cdot \frac{\partial\mathcal{L}}{\partial\mathbf{W}})\mathbf{x} = \hat{\mathbf{W}}\mathbf{x} \tag{4}$$

Assuming that α corresponds to a specific layer, the gradient of the augmented loss $\mathcal{L}_\alpha$ computes as:

$$\begin{aligned}\frac{\partial\mathcal{L}_\alpha}{\partial\alpha} &\overset{Eq.(2)}{=} \sum_i \frac{\partial\mathcal{L}}{\partial\hat{y}_i}\frac{\partial\hat{y}_i}{\partial\alpha} \overset{Eq.(4)}{=} \sum_i \frac{\partial\mathcal{L}}{\partial\hat{y}_i}\sum_j -\frac{\partial\mathcal{L}}{\partial w_{ij}}x_j = -\sum_i\sum_j\Big(\frac{\partial\mathcal{L}}{\partial\hat{y}_i}x_j\Big)\frac{\partial\mathcal{L}}{\partial w_{ij}} \\ &\overset{x_j=\partial\hat{y}_i/\partial\hat{w}_{ij}}{=} -\sum_i\sum_j\frac{\partial\mathcal{L}}{\partial\hat{w}_{ij}}\frac{\partial\mathcal{L}}{\partial w_{ij}} = -\langle\nabla\mathcal{L}(\hat{\mathbf{w}}), \nabla\mathcal{L}(\mathbf{w})\rangle\end{aligned} \tag{5}$$

where we use $x_j, \hat{y}_i, w_{ij}, \hat{w}_{ij}$ to denote elements of vectors $\mathbf{x}$, $\hat{\mathbf{y}}$ and matrices $\mathbf{W}$, $\hat{\mathbf{W}}$ respectively, set $\mathbf{w} = vec(\mathbf{W})$, $\hat{\mathbf{w}} = vec(\hat{\mathbf{W}})$ and $\langle\cdot,\cdot\rangle$ stands for the inner product.

[1] A bias term $\mathbf{b}$ can be easily incorporated in this form, by considering $\mathbf{W} \leftarrow \left(\begin{array}{c|c}\mathbf{W} & \mathbf{b}\\ 0 & 1\end{array}\right)$ and $\mathbf{x} \leftarrow \begin{pmatrix}\mathbf{x}\\1\end{pmatrix}$

This result can be straightforwardly extended to the network as a whole, regardless of the architecture, assuming a global learning rate α, which controls every weight tensor update:

$$\frac{\partial \mathcal{L}_a}{\partial \alpha} = \sum_{k=1}^{N} \langle \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}^k}, \frac{\partial \hat{\mathbf{w}}^k}{\partial \alpha} \rangle \overset{Eq.(4)}{=} -\sum_{k=1}^{N} \langle \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}^k}, \frac{\partial \mathcal{L}}{\partial \mathbf{w}^k} \rangle, \tag{6}$$

where we have used $\mathbf{w}^k$ for the weights of layer k. Overall, the learning rate gradient can be expressed as the inner product of consecutive gradients $\mathbf{g}_t = \nabla \mathcal{L}(\mathbf{w_t})$, where $\mathbf{w_t}$ is the vectorized form of all the network's weights at step t:

$$\frac{\partial \mathcal{L}_\alpha}{\partial \alpha}\Big|_{\alpha=\alpha_t} = -\langle \nabla \mathcal{L}(\mathbf{w_t}), \nabla \mathcal{L}(\mathbf{w_{t-1}}) \rangle = -\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle \tag{7}$$

The derived gradient has an intuitive interpretation:

- $\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle > 0$: If two consecutive weight updates move towards the same direction in order to be optimized, the learning rate should be increased.
- $\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle < 0$: On the other hand when the gradients have opposite directions, i.e. a local optimum is nearby and we skip it due to the "large" learning rate, the learning rate should be decreased in order to provide finer updates.
- $\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle = 0$: When the inner product is zero, either we reached a converged state ($||\mathbf{g}_t|| = 0$) or gradient directions are perpendicular. The later case corresponds to an optimal learning rate according to the line-search formulation (it holds that $\langle \nabla \mathcal{L}(\mathbf{w_t} - \alpha_\star \nabla \mathcal{L}(\mathbf{w_t})), \nabla \mathcal{L}(\mathbf{w_t}) \rangle = 0$, where $\alpha_\star$ is the optimal learning rate) and thus no change on the learning rate should be made.

Ultimately, the gradient derivation analysis links an intuitively heuristic rule with the optimization of learning rate (Jacobs, 1988; Almeida et al., 1997; Pesme et al., 2020). In practice, we encountered the same behavior as Pesme et al. (2020): The learning rate is initially increased ($\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle > 0$) until a maximum value and subsequently it decreases ($\langle \mathbf{g}_t, \mathbf{g_{t-1}} \rangle < 0$) in order to converge to a minimum.

Second-Order Gradient: A popular direction towards an adaptive and fast converging learning rate is adopting Newton-based methods, which include the derivation of second-order gradients. Despite the fact that such a second-order derivative would typically include demanding *Hessian* computations/approximations of the network's weights, the problem at hand has an intuitive analytical form, using only first-order weight gradients, as Eq. 8 suggests.

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha^2}\Big|_{\alpha=\alpha_t} = \frac{4}{\alpha_t} \langle \mathbf{g}_t, \mathbf{g}_t - \mathbf{g}_{t-1} \rangle = \frac{4}{\alpha_t} (\|\mathbf{g}_t\|^2 - \langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle) \tag{8}$$

A detailed description of the steps towards attaining Eq. 8 can be found in Appendix A. In order to attain a minimum, the second derivative should be positive. Nonetheless, Equation 8 can take negative values, when $\|\mathbf{g}_t\|^2 < \langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle$. Moreover, as this derivative is the denominator to the Newton-Raphson formulation, an impractical update direction which tends to infinity would arise when it takes values close to zero. To overcome this, we design the following rule for η:

$$\eta_t = \frac{\alpha_t}{\max(4\langle \mathbf{g}_t, \mathbf{g}_t - \mathbf{g}_{t-1} \rangle, c^{-1} \langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle)}, \tag{9}$$

where we form an overall bound on the update of α: $\alpha_{t+1} \leq (1+c)\alpha_t$ which imposes smoother behavior without any exploding increases of rate (hyperparameter $c > 0$ controls the bound). Nonetheless, even small bound values could lead to large increases over the learning rate, if needed, after a few iterations due to its exponential nature. Therefore, this is not a critical hyperparameter to be defined and has minor effect over the behavior of the algorithm. For the rest of the paper we select c as $c = 1/4$ (which implies $\alpha_{t+1} \leq 1.25\alpha_t$) and the effect of this parameter to the algorithm performance is included in the supplementary material.

Using the aforementioned η_t is equivalent to performing a Newton step (under the bounding condition), that is in the direction $-\nabla^2 \mathcal{L}(\alpha)^{-1} \nabla \mathcal{L}(\alpha)$. The step length is effectively specified by the second-order gradient (Nocedal & Wright, 2006). Moreover, this result is in line with the intuition of taking larger steps when the rate a is large and smaller steps when a is small.

Single-pass Algorithm: Finally, we can express the update equation for learning rate (Eq. 3) in a more coherent and cost-effective formulation using Equations 7 and 9. These equations rely on the values of the gradients $\mathbf{g}_{t-1}$ and $\mathbf{g}_t$, generated in two consecutive steps of the gradient descent process. Thus the second step of Algorithm 1, which corresponds to an extra forward-backward pass, can be omitted. We combine this consideration along with the update of α in Eq. 9, leading to the Efficient GD-TLR variant, summarized in Algorithm 2.

The major advantage of the proposed method (*Efficient* variant), especially compared to backtracking line-search algorithms, is its trivial computational overhead. The extra update of a can be performed using only pre-calculated values attained by classical GD (consecutive gradients). Implementation-wise, we should additionally store the calculated gradient at the previous step, as a temporary variable, in order to compute the inner product of consecutive gradients.

Per-layer learning rate: The reported algorithm refers to a global learning rate α, for which every weight from every layer contributes to its update. Nonetheless, the gradient derivation of Eqs. 5 and 9 hints towards an effortless definition of different learning rates per layer, or even per tensor (e.g., different rates for weights and biases). See Sec. D.2 in the Appendix for a more detailed discussion.

4 SGD Extension and Technical Considerations

The intuitive interpretation of Eq. 7 relies on the Gradient Descent formulation, where gradients are computed over the entire set of data. However, its stochastic counterpart, SGD, computes (sub)gradients for limited (usually non-overlapping) subsets of data, commonly referred to as batches. Therefore, consecutive gradients, as Eq. 7 dictates, would correspond to different data. The inner product of such consecutive gradients could be negative for the majority of batch pairs (different optimization directions for different data), leading to a rapidly diminishing learning rate.

To address this problem, we follow the SGD logic of expecting well-performing gradients, in average, after several iterations. To this end, we accumulate the calculated gradients across several consecutive iterations. These accumulated gradients can approximate a gradient direction close to the ideal gradient which corresponds to the entire set. In order to provide finer control over the whole process, we update the learning rate after a predefined number of iterations corresponding to a specific percentage p of the entire set. Therefore, at step t, the weight gradients can be computed as follows:

$$\mathbf{g}_t \approx \sum_{k=t-pN}^{t} g(\mathbf{w}_k; x_k) = \sum_{k=t-pN}^{t} (\mathbf{w}_k - \mathbf{w}_{k-1})/\alpha_t = \frac{\mathbf{w}_t - \mathbf{w}_{t-pN}}{a_t} \tag{10}$$

where x_k is the batch input at step k and the $g(\mathbf{w}_k; x_k)$ denotes the subgradient derivation given the batch x_k. Moreover, N are the steps needed to iterate through the entire dataset (corresponding to an epoch) and thus pN are the number of steps required between accumulating the gradients and updating the learning rate α. The impact of this user-defined hyperparameter p, which controls the frequency of learning rate updates is examined in Section D.1. Frequent updates would lead to a fast diminishing rate that may result to a suboptimal solution; in contrast, updating at the end of each epoch would slow down the whole procedure. To this end, we consider that $p = 0.33$ is a good trade-off and thus this is the default update frequency p for the rest of the paper.

5 Experimental Evaluation

Experimental Setup: We select four datasets on the image classification task: MNIST, CIFAR10/100 (Krizhevsky & Hinton, 2009) and ImageNet (Russakovsky et al., 2015). For MNIST, we use a multi-layer perceptron (MLP) with a hidden layer of width 1000, as in Vaswani et al. (2019). For CIFAR10 and CIFAR100, we selected the Wide-ResNet (WRNET) architectures (Zagoruyko & Komodakis, 2016) which can be modified easily, allowing us to study the proposed algorithm across various network width and depth settings (this exploration, along with experiments with the DenseNet architecture of Huang et al. can be found in the appendices). Finally, the ResNet50 architecture (He et al., 2016) is selected for the ImageNet setting. All networks are trained using standard cross entropy loss and run on one GeForce RTX 2080Ti GPU. Next, we first perform ablation studies over various configurations, and then compare our optimizer (the *efficient* variant of Alg.2) against other successful adaptive optimizers. Specifically, we evaluate both the global learning rate and the per-layer proposed alternatives, dubbed as *sgd_tlr* and *sgd_tlr_pl* respectively.

Robustness to hyperparameters: We distinguish two user-defined hyperparameters that may significantly affect the performance of an optimizer: the learning rate and the batch size. Note that these two parameters are codependent, since an increase in batch size should correspond to an increased learning rate in order to maintain a well-performing training scheme (Smith et al., 2018). Ideally, our algorithm should have similar performance regardless of the batch size (*bs*) or the initial learning rate α_0. Figure 1 results show that indeed our method is robust, with the loss and accuracy curves converging considerably fast into a common behavior for all the considered settings of MNIST and CIFAR10/100 (the behavior of the extreme case of batch size 64 and initial rate 0.1 in the MNIST setting is explained in Section D.5). Considering the learning rate progress, we observe the same behavior across all settings: the rate initially increases in order to accelerate the procedure and then decreases in an exponential manner. Moreover, the larger the batch size is, the higher the rate gets. Section D of Appendix includes detailed figures with respect to different learning rates and batch sizes, along with different optimizers/schedulers for comparison.

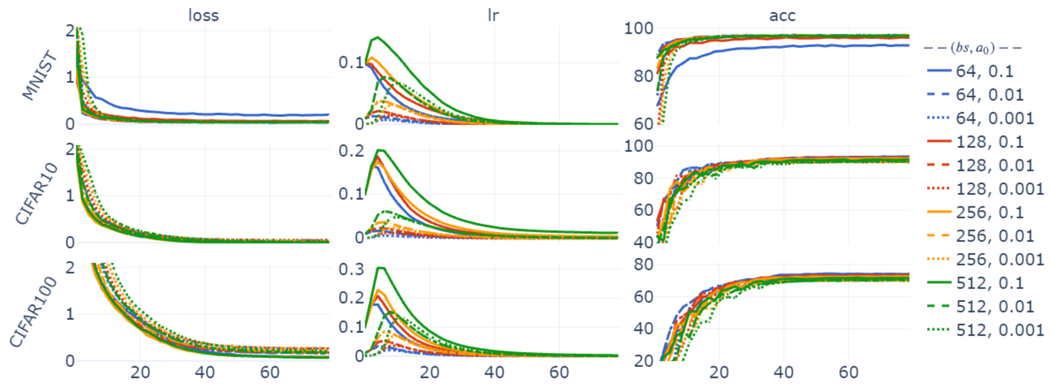


Figure 1: Performance of the proposed algorithm for different initial learning rates ($\alpha_0 = \{0.001, 0.01, 0.1\}$) and batch sizes ($bs = \{64, 128, 256, 512\}$). The following curves are provided: training loss (left), evolution of learning rate (middle) and test accuracy (right).

Comparison to related work on meta-optimization: First, we explore the case of the similarly motivated work of Baydin et al. (2018), an extension of the earlier work of Almeida et al. (1997), which shares an identical formulation up to the first-order derivation of learning rate gradient. Nonetheless, as we have already mentioned, such a formulation includes a meta-learning rate parameter that should be carefully defined and may be critically sensitive. Indeed, this sensitivity is evident in the comparison of Figure 2, where different values of this meta-learning rate β were considered. On the contrary, the proposed method provides a stable behavior, consistently outperforming the first-order approach of Baydin et al.. Observe that different values of β lead to vastly different behaviors, while the batch size also significantly affects the performance of the optimizer. Specifically, when using a very low value of β (10^{-5}) the algorithm of Baydin et al. practically gets a linearly decreasing learning rate which resembles typical scheduling tactics, while for large values (10^{-3}) the learning rate oscillates intensely. For intermediate values (10^{-4}) the optimizer may be trapped to an under-performing optima. Overall, this experiment highlights the importance of the proposed methodology compared to considering only a first-order gradient approach. Note that the meta-learning rate denoted as β in the paper of Baydin et al. (2018) is equivalent to our η of Eq. 3 and the effect of η, when no second-order derivative is used, under our framework is explored in the appendices (Section D.4).

State-of-the-art Comparison: Next, we consider the CIFAR100+WRNET setting for various batch sizes ($\{64, 128, 256, 512\}$). The results are summarized in Figure 3, where we also included state-of-the-art adaptive optimizers: Adam (Kingma & Ba, 2015), SLS (based on the line-search formulation) (Vaswani et al., 2019) and SPS (based on the Polyak step size) (Loizou et al., 2021). We also consider the popular multistep scheduler with SGD, denoted as *sgd_mstep*, where rates are decayed by 0.1 at 50% and 75% of the total number of epochs (with an initial learning rate of 0.1, a typical well-performing setting for CIFAR). Note that the curves denote the mean training loss and mean accuracy across three runs. Along with the mean we also show the standard deviation (shaded).

The following observations can be made: 1) The proposed approach achieves fast convergence to well-performing minima for all of the batch sizes considered, supporting our claim for an automated

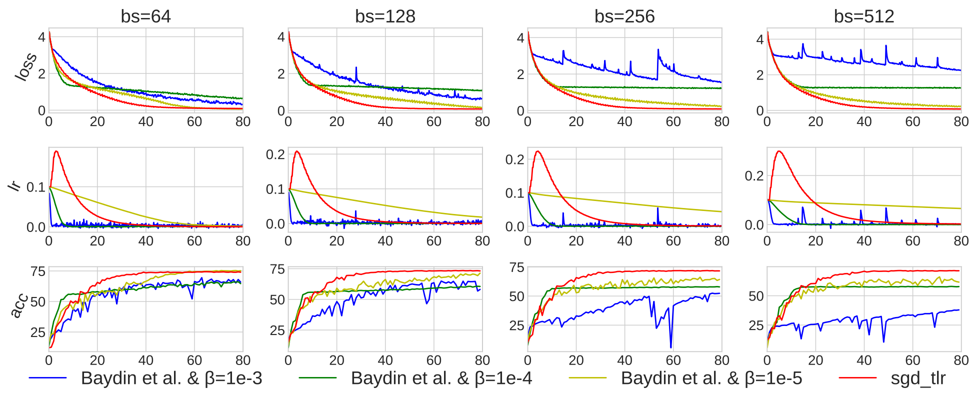


Figure 2: Comparison with the closely-related work of Baydin et al. (2018) for different batch sizes under the CIFAR100+WRNET_16_4 setting. Different values of the hyperparameter β were considered (akin to the η parameter of our Algorithm 1).

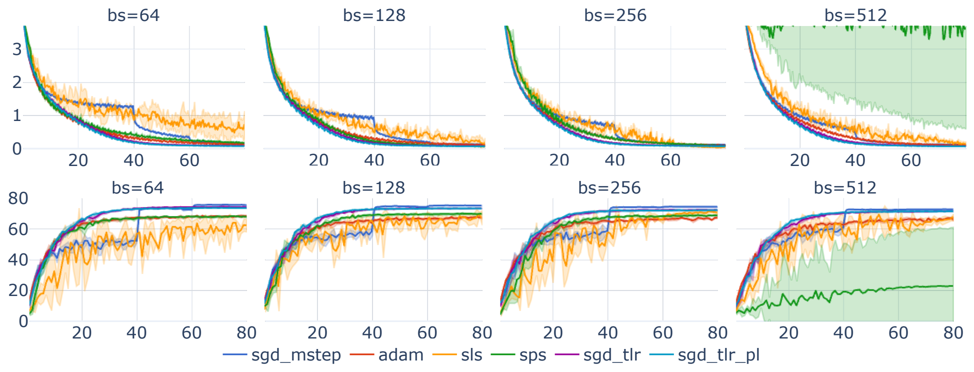


Figure 3: Loss (upper) and accuracy (bottom) curves of different optimizers for the setting CIFAR100 + WRNET_16_4, while selecting different batch sizes (from 64 to 512).

learning procedure without the need for empirical definition of a set of hyperparameters. 2) *sgd_mstep* provides well-performing models at the cost of manually setting when the rate should be decreased (and how much). 3) SPS and SLS are sensitive to extreme cases of batch size, as shown by the large standard deviation (assuming batch size values of 64 for SLS and 512 for SPS). On the contrary, the other methods present negligible deviation across different runs with the same settings. 4) Both proposed variants (*sgd_tlr* and *sgd_tlr_pl*) have similar performance. This is especially interesting, since the per-layer alternative provides very different step size behavior across layers (see Section D.2). 5) SPS behaves similarly to Adam, providing sub-par performance. Specifically, according to the accuracy curves, we observe generalization problems (Keskar & Socher, 2017).

The effectiveness of the proposed method is further validated on the large-scale ImageNet setting, as shown in Figure 4. The overall number of epochs were 80 and the batch size was set to 96 for all experiments due to GPU memory constraints. SLS approach diverges in this setting and thus is not included in the figure. Once again, SPS has identical behavior to Adam. Even though these two approaches exhibit convergence to slightly better overall loss, they have sub-optimal performance on the test set, with reduced accuracy, indicating a generalization problem. In contrast, our approach efficiently converges into a well-performing optimum, providing the best accuracy results.

Time Requirements: All compared optimizers, except SLS, have practically non-existent computational overhead. Specifically, the proposed algorithm only requires an inner product and a norm computation at each step. For comparison, Adam requires similar computation overhead for computing the squared gradient term. In the following, we report indicative results of time comparisons for CIFAR100 and the WRNET_16_4 architecture, measuring the required time for performing a training epoch (sec): SGD 24.26 / SGD-TLR 24.27 / Adam 24.28 / SPS 24.88 / SLS 30.25. As we can see, the optimizer overhead is practically non-existent for TLR (and for Adam/SPS) since the

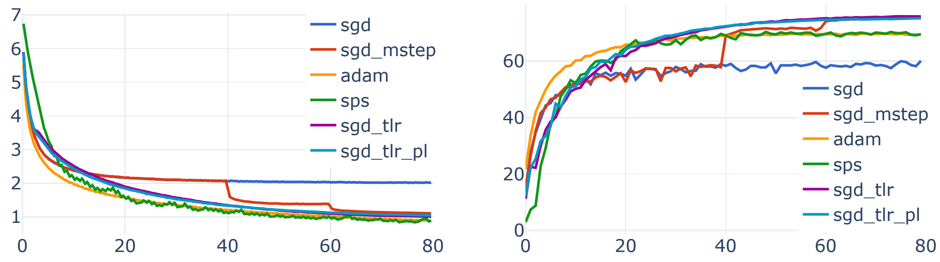


Figure 4: Loss (left) and accuracy (right) curves of different optimizers for the setting ImageNet + ResNet50, with batch size set to 96.

computations required in a training epoch, when considering a deep architecture, far exceed the trivial overhead introduced by the aforementioned optimizers. SLS, on the other hand, includes an iterative procedure (backtracking) at each training step which introduces a notable delay.

Per-layer Necessity: To fully grasp the capabilities and limitations of our method, we train an *Object Detection* system consisted of a Single Shot Multibox Detector (SSD) (Liu et al., 2016) and a pre-trained VGG backbone, applied on the Pascal VOC dataset [2]. The training loss curve can be seen in Figure 5, where the proposed alternative *sgd_tlr_pl* displays superior performance. However, the global rate TLR algorithm (*sgd_tlr*) leads to exploding gradients early on and thus is not reported. This behavior can be explained by the inconsistency of expected rates between the pre-trained part and the SSD head. In other words, the SSD head requires large increases of the rate, in contrast to the small required changes on the pre-trained backbone. Thus, this experiment showcases the effectiveness of the per-layer TLR alternative, where a finer control over the learning rate is desired.

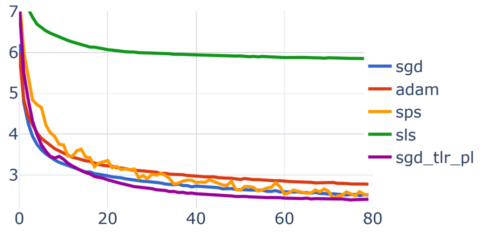


Figure 5: Loss curve of different optimizers for the Object Detection setting. The reported loss is the multibox loss as described in the work of Liu et al. (2016).

Final Remark: The proposed algorithm cannot provide a convergence guarantee for non-convex loss functions, where the increase of learning rate may lead to exploding gradients (e.g. object detection setting for the *sgd_tlr*). Moreover, the algorithm could be trapped at local minima if the learning rate decreases fast enough (a common problem with such adaptive optimizers, dubbed as “short-horizon bias” by Wu et al. (2018b)). Nonetheless, the proposed method can achieve accelerated convergence to the proximity of a well-performing optimum regardless of the setting or the hyperparameters, reporting minor sensitivity to such selections, making it an ideal component for more complex heuristic schedulers, such as restarting schemes.

6 Conclusions

We addressed the problem of automatically adjusting the learning rate of Gradient Descent, proposing an efficient algorithm with minor computational overhead compared to vanilla GD, that treats learning rate as an extra trainable parameter. Furthermore, we extended this idea to handle distinct learning rates at each layer, allowing a finer control over the descent acceleration. Experimental evaluation over a plethora of settings support the effectiveness of the proposed algorithm, which rapidly converges to well-performing local minima. This work paves the way towards the following research steps: analyze the conditions where such an algorithm can guarantee convergence over non-convex functions, formally define a SGD variant and examine how this algorithm can be extended as a meta-algorithm capable of using search directions other than the steepest descent direction (e.g. use alongside Adam).

[2]taken from *https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection*

REFERENCES

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International Conference on Machine Learning*, pp. 242–252. PMLR, 2019.

Luís Almeida, Thibault Langlois, D. Amaral José, and Rua Alves Redol. On-line step size adaptation. In *INESC. 9 Rua Alves Redol, 1000*. Citeseer, 1997.

Luís Almeida, Thibault Langlois, D. Amaral José, and Alexander Plakhov. Parameter adaptation in stochastic optimization. In *Publications of the Newton Institute*, pp. 111–134. 1999.

Mathieu Barré, Adrien Taylor, and Alexandre d'Aspremont. Complexity guarantees for Polyak steps with momentum. In *Conference on Learning Theory*, pp. 452–478. PMLR, 2020.

Jonathan Barzilai and Jonathan M. Borwein. Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148, 1988.

Atılım Günes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *Proceedings of the International Conference on Learning Representations*, 2018.

Aaron Defazio and Léon Bottou. On the ineffectiveness of variance reduced optimization for deep learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Caroline Etienne, Guillaume Fidanza, Andrei Petrovskii, Laurence Devillers, and Benoit Schmauch. CNN+LSTM architecture for speech emotion recognition with data augmentation. *arXiv preprint arXiv:1802.05630*, 2018.

Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, and Peter Richtárik. SGD: General analysis and improved rates. In *International Conference on Machine Learning*, pp. 5200–5209. PMLR, 2019.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pp. 1225–1234. PMLR, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, Inc., 1989.

Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from Adam to SGD. *arXiv preprint arXiv:1712.07628*, 2017.

Harry Kesten. Accelerated stochastic approximation. *The Annals of Mathematical Statistics*, pp. 41–59, 1958.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations*, 2015.

Milena Kresoja, Zorana Lužanin, and Irena Stojkovska. Adaptive stochastic approximation algorithm. *Numerical Algorithms*, 76(4):917–937, 2017.

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

Xiaoyu Li and Francesco Orabona. On the convergence of stochastic gradient descent with adaptive stepsizes. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 983–992. PMLR, 2019.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision*, pp. 21–37. Springer, 2016.

Nicolas Loizou, Sharan Vaswani, Issam Hadj Laradji, and Simon Lacoste-Julien. Stochastic Polyak step-size for SGD: An adaptive learning rate for fast convergence. In *International Conference on Artificial Intelligence and Statistics*, pp. 1306–1314. PMLR, 2021.

I. Loshchilov and F. Hutter. SGDR: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations*, 2017.

Yura Malitsky and Konstantin Mishchenko. Adaptive Gradient Descent without descent. In *International Conference on Machine Learning*, pp. 6702–6712. PMLR, 2020.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pp. 4556–4565. PMLR, 2019.

Ali A. Minai and Ronald D. Williams. Back-propagation heuristics: a study of the extended Delta-Bar-Delta algorithm. In *1990 IJCNN International Joint Conference on Neural Networks*, pp. 595–600. IEEE, 1990.

Yuri Nesterov. A method of solving a convex programming problem with convergence rate o (1/kˆ 2) o (1/k2). In *Sov. Math. Dokl*, volume 27, 1983.

Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

Brendan O'Donoghue and Emmanuel Candes. Adaptive restart for accelerated gradient schemes. *Foundations of computational mathematics*, 15(3):715–732, 2015.

Scott Pesme, Aymeric Dieuleveut, and Nicolas Flammarion. On convergence-diagnostic based step sizes for stochastic gradient descent. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 7641–7651, 2020.

Georg C. Pflug. On the determination of the step size in stochastic quasigradient methods. 1983.

Elijah Polak and Gerard Ribière. Note sur la convergence de méthodes de directions conjuguées. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 3(R1):35–43, 1969.

Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Boris T. Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4):94–112, 1969.

Youngmin Ro and Jin Young Choi. AutoLR: Layer-wise pruning and auto-tuning of learning rates in fine-tuning of deep networks. In *In proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

Clément W. Royer and Stephen J. Wright. Complexity analysis of second-order line-search algorithms for smooth nonconvex optimization. *SIAM Journal on Optimization*, 28(2):1448–1477, 2018.

O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

George N. Saridis. Learning applied to successive approximation algorithms. *IEEE Transactions on systems science and cybernetics*, 6(2):97–103, 1970.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pp. 343–351. PMLR, 2013.

Nicol N. Schraudolph. Local gain adaptation in stochastic gradient descent. Technical Report IDSIA-09-99, IDSIA, 1999.

Hanie Sedghi, Vineet Gupta, and Philip M. Long. The singular values of convolutional layers. In *Proceedings of the International Conference on Learning Representations*, 2019.

Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *Proceedings of the International Conference on Learning Representations*, 2018.

Ju Sun. *When are nonconvex optimization problems not scary?* PhD thesis, Columbia University, 2016.

Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pp. 823–832, 1986.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-RMSProp, Coursera: Neural Networks for machine learning. *University of Toronto, Technical Report*, 2012.

Sharan Vaswani, Aaron Mishkin, Issam Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. Painless stochastic gradient: Interpolation, line-search, and convergence rates. *arXiv preprint arXiv:1905.09997*, 2019.

Xiaoxia Wu, Rachel Ward, and Léon Bottou. Wngrad: Learn the learning rate in gradient descent. *arXiv preprint arXiv:1803.02865*, 2018a.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. In *Proceedings of the International Conference on Learning Representations*, 2018b.

Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. In *Proceedings of the International Conference on Learning Representations*, 2020.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In Edwin R. Hancock Richard C. Wilson and William A. P. Smith (eds.), *Proceedings of the British Machine Vision Conference (BMVC)*, 2016.

Jian Zhang and Ioannis Mitliagkas. Yellowfin and the art of momentum tuning. In A. Talwalkar, V. Smith, and M. Zaharia (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 289–308, 2019.

Appendices

A Second-order Derivative with respect to Learning Rate α

Given a single linear layer, we have proven that the first-order derivative with respect to learning rate can be written as follows, only depending on weight gradients:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = -\sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_{ij}} \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{11}$$

For the sake of brevity we use a flattened version of the weight tensors and their gradients. Thus Eq. 11 can be re-written as:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = -\sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i} \frac{\partial \mathcal{L}}{\partial w_i} \tag{12}$$

The derivative of this formula is computed as follows:

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial \alpha^2} &= \frac{\partial\big(-\sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial \mathcal{L}}{\partial w_i}\big)}{\partial \alpha} = -\sum_i \frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \hat{w}_i}\frac{\partial \mathcal{L}}{\partial w_i} - \sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial \alpha \partial w_i} \\
&= -\sum_i \frac{\partial\big(\frac{\partial \mathcal{L}}{\partial \alpha}\big)}{\partial \hat{w}_i}\frac{\partial \mathcal{L}}{\partial w_i} - \sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial\big(\frac{\partial \mathcal{L}}{\partial \alpha}\big)}{\partial w_i} \\
&\overset{(12)}{=} \sum_i \frac{\partial\big(\sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j}\big)}{\partial \hat{w}_i}\frac{\partial \mathcal{L}}{\partial w_i} + \sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial\big(\sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j}\big)}{\partial w_i} \\
&= \sum_i \Big(\sum_j \frac{\partial^2 \mathcal{L}}{\partial \hat{w}_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} + \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_j}\frac{\partial^2 \mathcal{L}}{\partial \hat{w}_i \partial w_j}\Big)\frac{\partial \mathcal{L}}{\partial w_i} \\
&+ \sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\Big(\sum_j \frac{\partial^2 \mathcal{L}}{\partial w_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} + \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_j}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}\Big) \\
&= \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial w_i}\frac{\partial^2 \mathcal{L}}{\partial \hat{w}_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} + \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial w_i}\frac{\partial^2 \mathcal{L}}{\partial \hat{w}_i \partial w_j}\frac{\partial \mathcal{L}}{\partial \hat{w}_j} \\
&+ \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} + \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}\frac{\partial \mathcal{L}}{\partial \hat{w}_j}
\end{aligned} \tag{13}$$

Let's focus on the first term of $\frac{\partial^2 \mathcal{L}}{\partial \alpha^2}$ and utilize consecutive chain rule replacements:

$$\begin{aligned}
&\sum_i \sum_j \frac{\partial \mathcal{L}}{\partial w_i}\frac{\partial^2 \mathcal{L}}{\partial \hat{w}_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} = \sum_i \sum_j \Big[\frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial \hat{w}_i}{\partial w_i}\Big]\frac{\partial\Big(\frac{\partial \mathcal{L}}{\partial \hat{w}_j}\Big)}{\partial \hat{w}_i}\frac{\partial \mathcal{L}}{\partial w_j} \\
&= \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\Big[\frac{\partial\Big(\frac{\partial \mathcal{L}}{\partial \hat{w}_j}\Big)}{\partial \hat{w}_i}\frac{\partial \hat{w}_i}{\partial w_i}\Big]\frac{\partial \mathcal{L}}{\partial w_j} = \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial \hat{w}_j}\frac{\partial \mathcal{L}}{\partial w_j} \\
&= \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial\Big(\frac{\partial \mathcal{L}}{\partial w_i}\Big)}{\partial \hat{w}_j}\Big[\frac{\partial \mathcal{L}}{\partial \hat{w}_j}\frac{\partial \hat{w}_j}{\partial w_j}\Big] = \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\Big[\frac{\partial\Big(\frac{\partial \mathcal{L}}{\partial w_i}\Big)}{\partial \hat{w}_j}\frac{\partial \hat{w}_j}{\partial w_j}\Big]\frac{\partial \mathcal{L}}{\partial \hat{w}_j} \\
&= \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}\frac{\partial \mathcal{L}}{\partial \hat{w}_j}
\end{aligned} \tag{14}$$

Using similar arguments all terms of the final form of Eq. 13 are equal and thus Eq. 13 can be formulated as:

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha^2} = 4\sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}\frac{\partial \mathcal{L}}{\partial \hat{w}_j} \tag{15}$$

The gradient derivation leads to the definition of the scalar second-order derivative of learning rate using the Hessian matrix elements of $\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}$. However, calculating the Hessian matrix of the weights is a very computationally intense procedure and almost impractical when larger number of weights are considered.

To address this, we rely on the initial definition of $\hat{w}$: $\hat{w}_i = w_i - \alpha \frac{\partial \mathcal{L}}{\partial w_i}$. The derivative of this equation with respect to to w_j results in the following definition:

$$\frac{\partial \hat{w}_i}{\partial w_j} = \delta_{ij} - \alpha \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j} \Rightarrow \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j} = \alpha^{-1}\Big(\delta_{ij} - \frac{\partial \hat{w}_i}{\partial w_j}\Big), \tag{16}$$

where δ_{ij} is the Kronecker delta operator.

Therefore, in order to avoid computing second derivatives over the weights, we make use of Eq. 16 as follows:

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \alpha^2} &= 4\sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i} \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j} \frac{\partial \mathcal{L}}{\partial \hat{w}_j} \overset{(16)}{=} \frac{4}{\alpha} \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial \hat{w}_i} \Big(\delta_{ij} - \frac{\partial \hat{w}_i}{\partial w_j}\Big) \frac{\partial \mathcal{L}}{\partial \hat{w}_j} \\ &= \frac{4}{\alpha} \sum_i \Big[\Big(\frac{\partial \mathcal{L}}{\partial \hat{w}_i}\Big)^2 - \frac{\partial \mathcal{L}}{\partial w_i} \frac{\partial \mathcal{L}}{\partial \hat{w}_i}\Big] = \frac{4}{\alpha} \sum_i \frac{\partial \mathcal{L}}{\partial \hat{w}_i} \Big(\frac{\partial \mathcal{L}}{\partial \hat{w}_i} - \frac{\partial \mathcal{L}}{\partial w_i}\Big)\end{aligned}$$

This final form contains only the first-order gradients of two consecutive network updates and therefore we can calculate the second-order derivative with respect to α with trivial computational overhead.

B Connection to Two-Point Algorithms

Our proposed update rule on learning rate α makes use of Newton-Raphson approach, providing a simple connection to second-order derivative methods. Tackling the problem from a different viewpoint and specifically if a quasi-Newton approach is assumed for the gradient descent scheme and the approximation of the Hessian matrix is set as $\alpha_t \mathbf{I}$, the adaptive Barzilai-Borwein (BB) step (Barzilai & Borwein, 1988) is obtained:

$$\alpha_t = \frac{\mathbf{s}_{t-1}^\intercal \mathbf{y}_{t-1}}{\mathbf{y}_{t-1}^\intercal \mathbf{y}_{t-1}} \tag{17}$$

where $\mathbf{s}_{t-1} = \mathbf{w}_t - \mathbf{w}_{t-1}$ and $\mathbf{y}_{t-1} = \nabla \mathcal{L}(\mathbf{w}_t) - \nabla \mathcal{L}(\mathbf{w}_{t-1})$. This adaptive BB step shares similarities with the proposed Eq. 9, in the sense of utilizing consecutive gradients in order to find an improved adaptive step size, capable of further decreasing the loss function. Alas, convergence guarantees for the Barzilai-Borwein equations exist only for quadratic problems.

Tackling the problem from a momentum-based viewpoint, we see similar traits to our core idea. Take for example the heavy ball method (Polyak's method (Polyak, 1964)) which can be summarized as follows: if the current gradient step has the same direction as the previous step, then move a little further in that direction; If the current and previous gradients have opposite directions, move less far. Interestingly enough, the PRP (Polak & Ribière (1969) and Polyak (1969)) method, proposed as a conjugate gradient approach, contains a momentum-like term very similar to the derived one in our approach (although inversed):

$$\beta^{PRP} = \frac{\|\nabla \mathcal{L}(\mathbf{w}_t)\|^2 - \nabla \mathcal{L}(\mathbf{w}_t)^\intercal \nabla \mathcal{L}(\mathbf{w}_{t-1})}{\|\nabla \mathcal{L}(\mathbf{w}_{t-1})\|^2}$$

C Connection to Schedulers

The proposed algorithm is a GD-based optimizer with adaptive step size, following a large line of research where the learning rate is automatically adjusted according to a well-defined optimization problem (Vaswani et al., 2019; Loizou et al., 2021). A characteristic example of such algorithms is line-search optimization, which was the main motivation of this work. Notably, their behavior resembles the effect of a scheduler, where learning rate is adjusted according to a set of rules. Nonetheless, adaptive optimizers search for an “optimal” way to adaptively set the learning rate, while schedulers are empirical, typically user-tuned, algorithms which rely on well-founded theoretical motivations.

Despite the differences in formulating these two families of algorithms, namely adaptive optimizers and schedulers, their end goal is the same. Constructing a scheduler with the same learning rate behavior would lead to the same performance, while fine-tuning the scheduler could lead to even better performance. However, the main advantage of our method is exactly the ability to automatically extract an underlying scheduler policy on the fly, regardless of the network or the dataset, without the need to define a set of sensitive hyper-parameters. For example, one may simulate the typical behavior of TLR by using a scheduler consisted of warm-up phase with linear increase of learning rate and followed by an exponentially decay of learning rate (a widely used scheduling scheme). Nonetheless, such a scheduler requires the following parameters to be set: initial learning rate, maximum learning rate, epoch of terminating warm-up phase, exponential decay parameter. In contrast, the proposed method, or even similar adaptive step-size algorithms, decide on the learning rate behavior according to an "internal" optimization criterion without carefully setting any hyper-parameter.

We should stress that we do not consider our approach as a replacement of existing optimizers/schedulers, rather than a complementary module that can greatly increase convergence rate towards well-performing optima as supported by the experimental evaluation. Nonetheless, due to the “aggressive” control over learning rate, the proposed method could theoretically be trapped in a sub-optimal local minimum (Wu et al. (2018b)). One may address this problem by restarting the whole procedure when learning rate has become less than a predefined value (Loshchilov & Hutter, 2017). In other words, one can design a heuristic scheduler on top of our approach. The initial learning rate, after a restart, can be selected either according to a predefined value or even from using each time values from the previous cycle (e.g. the maximum rate a attained). The aforementioned ideas hint towards possible future research directions of the proposed algorithm under the context of a complex scheduling approach.

D Additional Experiments

First, we provide additional details about the experimental setup:

- experiments were conducted for 80 epochs (except the upcoming Transformer example, where 100 epochs were used). We consider that this number of epochs can give us indicative curves for the behavior of different optimizers.
- All SGD-based optimizers (regardless of whether they use a scheduler) use weight decay and momentum parameters equal to $5e^{-4}$ and 0.8 respectively. This applies also on the proposed algorithms *sgd_tlr* and *sgd_tlr_pl*, where gradients are calculated according to the formula derived in Section 5:

$$\mathbf{g}_t \approx \frac{\mathbf{w}_t - \mathbf{w}_{t-pN}}{a_t}$$

- The frequency update hyperparameter, denoted as a percentage p, of proposed algorithm is set to $1/3$ unless stated otherwise.

D.1 Impact of update frequency hyperparameter p

An intrinsic hyperparameter of our method related to the definition of the SGD variant of Section 4 is the user-defined update frequency p. The update frequency corresponds to the number of steps between different learning rate updates and is defined as a percentage of the dataset size. We can see how p may affect performance: Frequent updates would lead to a fast diminishing rate that may result to a suboptimal solution, while subgradients do not faithfully represent the data (see Section D.1); in contrast, updating at the end of each epoch would slow down the whole procedure since weights are updated much more frequently compared to the learning rate. To determine which percentage corresponds to an optimal trade-off between speed and quality of convergence, we report in Figure 6 the impact of the update frequency on a WRNET-16-4 trained on the CIFAR100 dataset (we note that MNIST is a rather simple dataset and the update frequency p had minor effect on the loss curves). We observe that updating once per epoch (100% percentage) leads to a slower convergence, even though the attained solution at the end is well-performing. On the other hand, frequent rate updates (e.g. 10%), hastily minimize the learning rate and, as a result, we get trapped on sub-optimal solutions. Values such as 25% and 33% provide a decent trade-off between the aforementioned extreme situations.

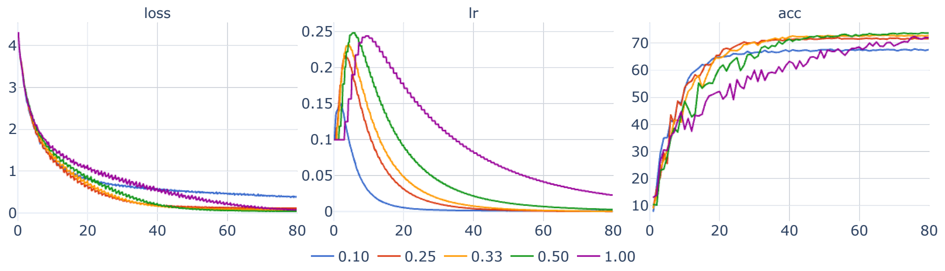


Figure 6: Impact of update frequency for the setting CIFAR100 + WRNET_16_4, with batch size equal to 256. Update frequency is provided as a percentage of the dataset size. The following curves are provided: training loss (left), evolution of learning rate (middle) and test accuracy (right).

Remark: Learning rate α or the meta-rate η are very sensitive and critical hyper-parameters which can easily lead to divergence if not set right. One the contrary, this intrinsic hyper-parameter p of our method is robust and provides a typical tradeoff of precision/speed, without leading to extreme cases (e.g. divergence). Moreover, it can be intuitively set to a modest value (e.g. 0.33 or 0.5 / $p = 1$ corresponds to full batch-size) which provides a decent tradeoff, without any further need for adjustment. In other words, the choice of p is not critical for the performance of the optimizer, whereas rate α (for typical SGD) or even η (meta learning rate of Eq.6) are very sensitive and should be chosen very carefully (as explored in Section D.4 and as Figure 2 shows).

D.2 Granularity of Control

The capability of independently controlling different rates α_l at each layer has not been explored as extensively as other techniques in the literature, while theoretically it can be beneficial (Schaul et al., 2013; You et al., 2017; Etienne et al., 2018; You et al., 2020; Ro & Choi, 2021); different types of layers (convolutional vs linear) or layers corresponding to different depth of the network (input layer vs output layer), may display varying levels of sensitivity and thus a per-layer granularity could lead to faster convergence. We should note that the per-layer rationale cannot be supported by backtracking line-search algorithms, which assume single-valued learning rates (preforming an extra forward step for every layer's learning rate would be computationally impractical).

We assume that each weight tensor (convolutional, linear or bias) has a unique, independent learning rate, which is initialized according to a global initial rate for the whole model. In theory, this alternative can provide finer level of control and subsequently lead to better solutions. In practice, we observed the exact same behavior with the global TLR algorithm in terms of the reported loss and accuracy. Nonetheless, the per-layer TLR algorithm produces vastly different rate curves for each layer, as Figure 7 suggests. Notably, layers/tensors closer to the output corresponds to rate curves that increase considerably faster.

D.3 Impact of bound c

After deriving a Newton-based update rule for learning rate α, we introduced a bound c (Eq. 9) with a two-fold importance: 1) properly define a positive second-order derivative for the Newton-Raphson method and 2) impose a maximum increase over α. In this section we evaluate the performance of the proposed algorithm under different bounds c, as shown in Figure 8. Only 40 epochs are depicted in order to better depict the initial behavior.

The following observations can be made:
• Considering the MNIST setting, we observe that the algorithm diverges for large bound such as $c = 5$ or $c = 10$. This supports the need for a stricter bound.
• The choice of bound is practically irrelevant when the initial learning rate is carefully chosen to

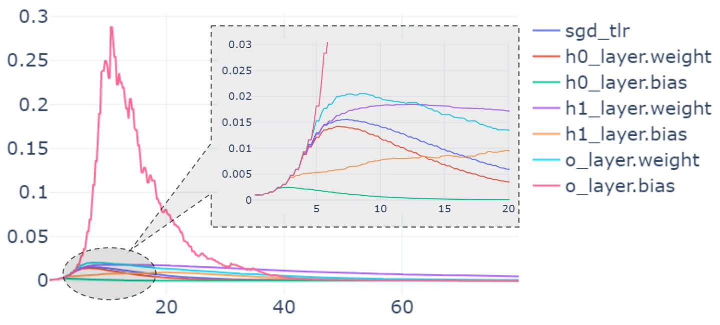


Figure 7: Per-layer learning rate curves for the MNIST+MLP setting. The global learning rate curve of the initial algorithm *sgd_tlr* is also included.

"favor" the setting at hand (e.g. $lr = 0.1$ for CIFAR datasets).
• When initial learning rate is chosen to be too far from a favorable value (e.g. $lr = 0.001$ for CIFAR100), a stricter bound slows down the decrease of loss, as expected. Nonetheless, the overall behavior of the proposed algorithm is the same.

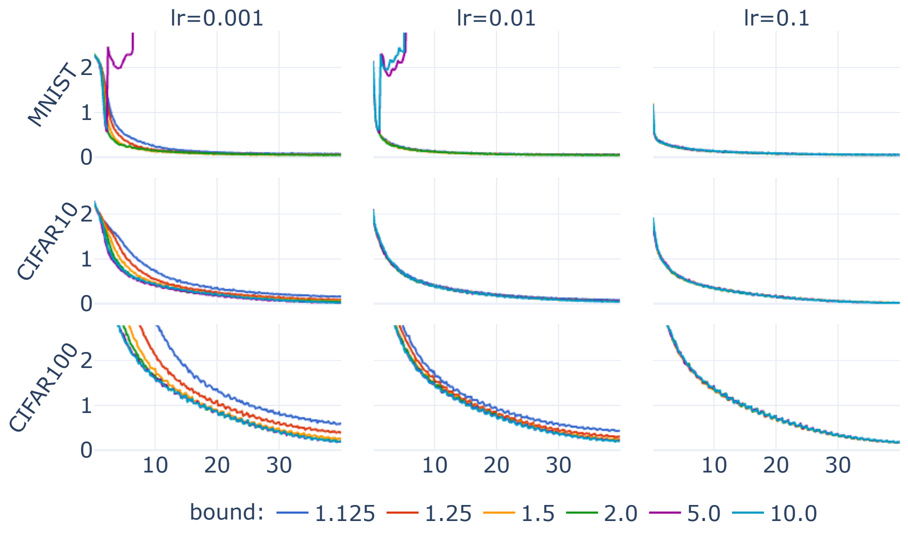


Figure 8: Exploration of different bounds c evaluated for both CIFAR10 and CIFAR100 datasets with the WRNET_16_4 architecture (256 batch size and 0.1 initial learning rate). Only the first 40 epochs are reported in order to have a finer depiction of the differences.

Remark: Assuming a convex function $\mathcal{L}$, it holds $\langle \mathbf{g}_t, \mathbf{w}_{t-1} - \mathbf{w}_t \rangle \leq \mathcal{L}(\mathbf{w}_{t-1}) - \mathcal{L}(\mathbf{w}_t)$ and thus:

$$\mathcal{L}(\mathbf{w}_t) \leq \mathcal{L}(\mathbf{w}_{t-1}) - \alpha_t \langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle$$

The assumption of a convex loss function leads to improved (minimized) loss values at each step when $\langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle > 0$. In other words, given a convex function and the proposed learning rate update rule, which relies on the intrinsic property $\langle \mathbf{g}_t, \mathbf{g}_{t-1} \rangle$, the proposed algorithm cannot diverge when learning rate is increasing, corresponding to positive values of the inner product of consecutive gradients. Nonetheless, we do not expect such convex functions in practice (e.g. loss functions of DNNs). This is the reason that we used a rather restricting upper bound on the possible increase of α, namely $\alpha_t \leq 1.25\alpha_{t-1}$. We assume that for ill-conditioned non-convex functions a sudden increase in learning rate may lead to exploding gradients and eventually to divergence.

D.4 Effect of Using a Fixed η

The proposed algorithm uses a second-order derivative to overcome the need of an extra meta-learning rate hyperparameter η. In this section, we assume that learning rate is calculated as

$\alpha_{t+1} = max(0, \alpha_t + \eta\langle \mathbf{g}_t, \mathbf{g}_{t-1}\rangle)$, aiming to examine the behavior of a fixed meta-learning rate alternative. For a set of different η values, we evaluated the aforementioned variant of the proposed algorithm for the typical classification settings. The results are depicted in Figure 9. One can observe that small η values leading to a behavior akin to SGD with fixed learning rate, since rate does not change fast enough. On the other hand, large η values lead to oscillations. This can be explained as follows: learning rate is zeroed while gradients have non-negligible values and therefore the gradient at the next step is the same as the previous. This means that the term $\langle \mathbf{g}_t, \mathbf{g}_{t-1}\rangle$ becomes positive and rate should be abruptly increased. Overall, we can see that selecting an appropriate meta-learning rate can be as tedious as selecting the typical learning rate.

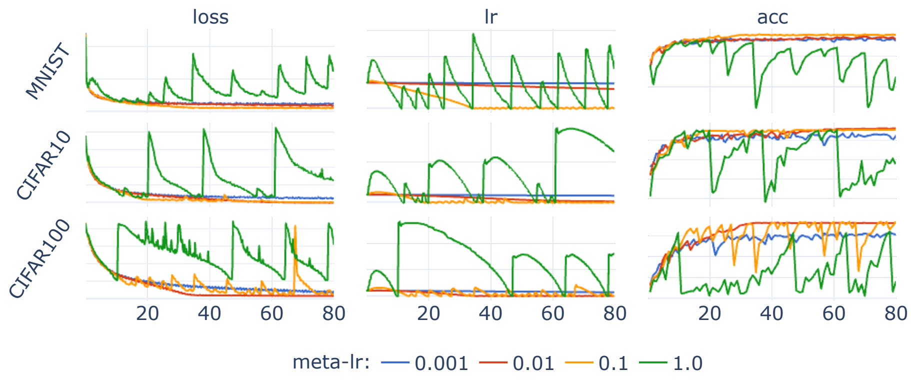


Figure 9: Evaluation of the algorithm's behavior when using a fixed meta-learning rate η. Experiments were conducted on the settings MNIST+MLP, CIFAR10+WRNET_16_4 and CIFAR100+WRNET_16_4 (256 batch size and 0.1 initial learning rate).

D.5 Detailed Experiments on MNIST, CIFAR10 and CIFAR100

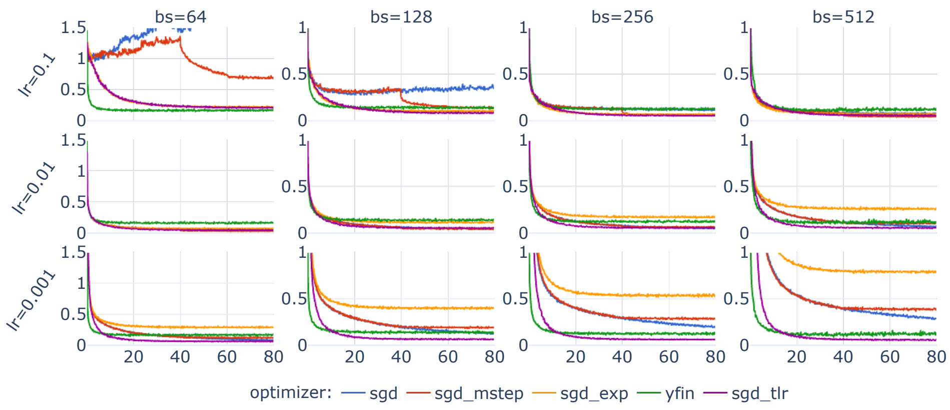


Figure 10: Loss curves for different initial learning rates and batch sizes (MNIST+MLP).

Initial Learning Rate vs Batch Size: Figures 10,11 and 12 depict the impact of different initial learning rates α_0 and batch sizes to our optimizer *sgd_tlr*. Each pair of initial learning rate and batch size corresponds to a different curve, evaluated on all possible classification settings (both MNIST and CIFAR datasets). To fully grasp the effectiveness of our method, we also report results using an SGD optimizer along with two possible schedulers: 1) *sgd_mstep*: multistep (rates are decayed by 0.1 at 50% and 75% of the epochs) and 2) *sgd_exp:* exponential decay scheme. Furthermore, we are also reporting results using the modern momentum-based optimizer YellowFin (Zhang & Mitliagkas, 2019) (denoted as *yfin*), known to be robust with respect to the initial learning rate.

The following observations can be made:
• The proposed *sgd_tlr* constantly outperforms SGD and its scheduler variants, which may perform poorly if we do not choose an appropriate learning rate and batch size. Note that the exponential

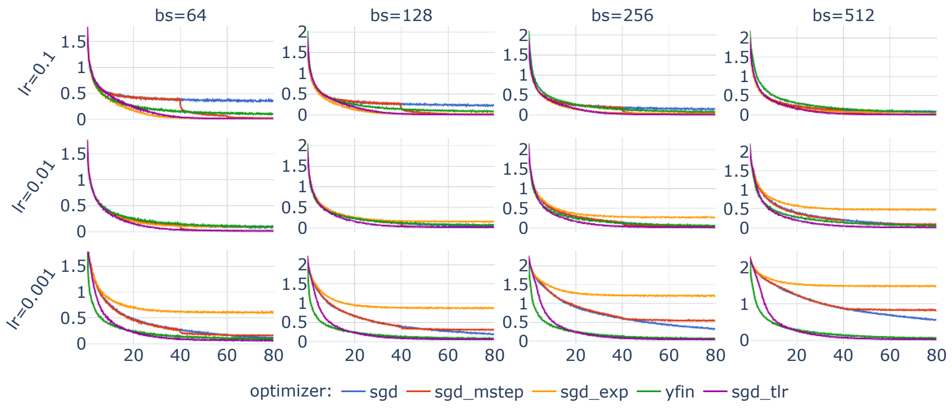


Figure 11: Loss curves for different initial learning rates and batch sizes (CIFAR10+WRNET_16_4).

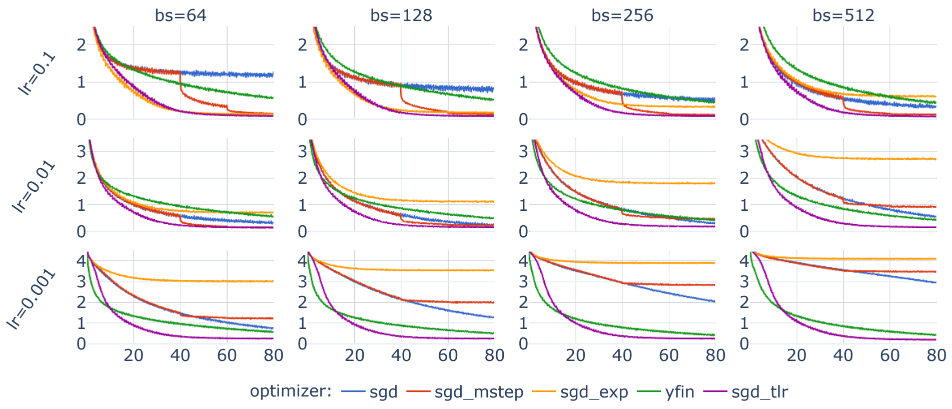


Figure 12: Loss curves for different initial learning rates and batch sizes (CIFAR100+WRNET_16_4).

decay scheduler has similar performance to the proposed optimizer for the “extreme” case of initial learning rate equal to 0.1 and batch size equal to 64. This was expected, since learning rate should be decreased immediately and the proposed update rule for decreasing learning rate corresponds to an exponential scheme.

• Considering MNIST, for the “extreme” case of initial learning rate equal to 0.1 and batch size equal to 64, the proposed algorithm converges to a sub-optimal minimum, with increased loss compared to other settings. Nonetheless, we can observe that other SGD variants fail completely to converge to such minima, supporting our claim for improved performance. Note that YellowFin optimizer exhibits excellent performance at this setting.
• YellowFin optimizer (*yfin*) displays increased sensitivity to the initial learning rate hyperparameter compared to our approach, even though it is characterized by robust behavior in general. Especially for the CIFAR100 case, YellowFin presents sub-par performance compared to our proposed optimizer.

Batch Size Sensitivity and SoA Adaptive Optimizers: In Figure 13 we provide the complementary plot for exploring different different batch sizes for CIFAR10 dataset, while comparing against modern optimizers (main paper contains only the case of CIFAR100). We observe similar behavior to the CIFAR100 setting, where the proposed algorithm *sgd_tlr* and its variant *sgd_tlr_pl* converge rapidly to a local minimum, outperforming the other approaches for the majority of the settings. Note that both *sls* and *sps* methods displayed sensitivity for the “extreme” case of batch size (specifically for 64 and 512) for the CIFAR100 setting. Nevertheless, for the CIFAR10 case, both algorithms have better behavior, without any sign of divergence.

Considering Different Architectures: Next, we consider the CIFAR10/100+WRNET setting for various depth and width configurations (denoted as WRNET_{depth}_{width_factor} - for more details, see the work of Zagoruyko & Komodakis). The results are summarized in Figures 14

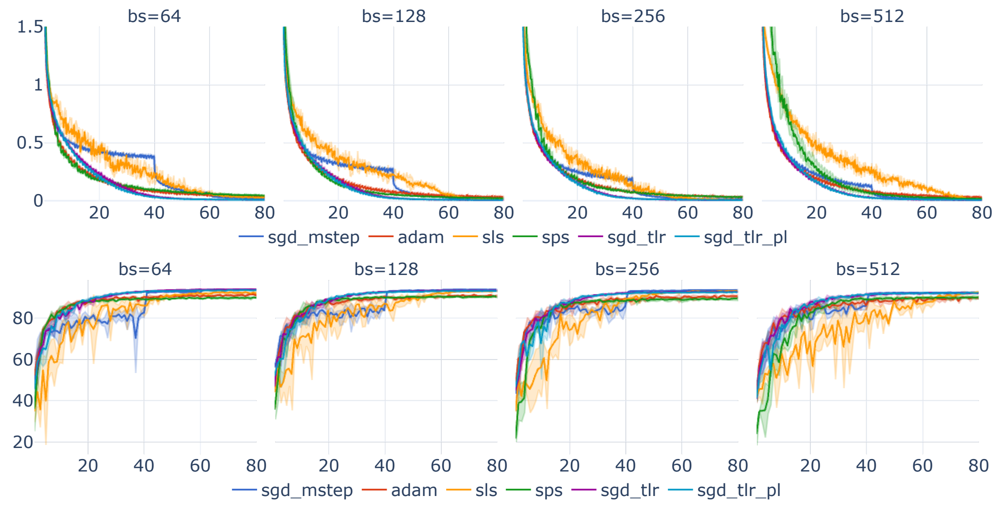


Figure 13: Loss (upper) and accuracy (bottom) curves of different optimizers for the setting CIFAR10 + WRNET_16_4, while selecting different batch sizes (from 64 to 512).

and 15, for CIFAR10 and CIFAR100 respectively. We observe that the proposed method generalizes well along all architectural variations, providing superior performance compared to the considered adaptive optimizers. Specifically, SLS displays a sensitivity issue with a turbulent behavior in the starting epochs, though it converges towards the end of the training process. SPS and Adam provide similar sub-par performance, achieving slower convergence to a suboptimal solution, which does not generalizes well according to the accuracy curves, after 80 epochs. Note that SLS has better generalization behavior compared to Adam and SPS.

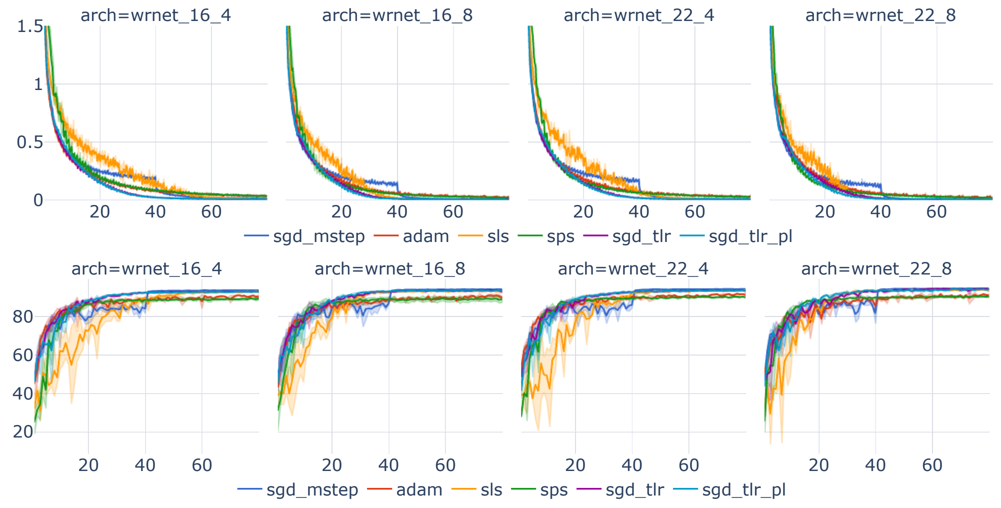


Figure 14: Loss (upper) and accuracy (bottom) curves of different optimizers for the setting CIFAR10 + WRNET_{d}_{w}. Batch size is fixed to 256.

We also evaluated the proposed algorithm by training a modified DenseNet-121 architecture (Huang et al., 2017) on the CIFAR100 dataset, reducing the growth rate to 12 and replacing the first layer with a convolution layer with kernel size 3, stride 1, and padding 1, to account for the image size in the CIFAR dataset. The results are shown on Figure 16, where competing optimizers are also reported. As we can see, the proposed algorithm also performs well under this setting, outperforming the compared optimizers. Note that the SGD along with the multi-step scheduler report the highest

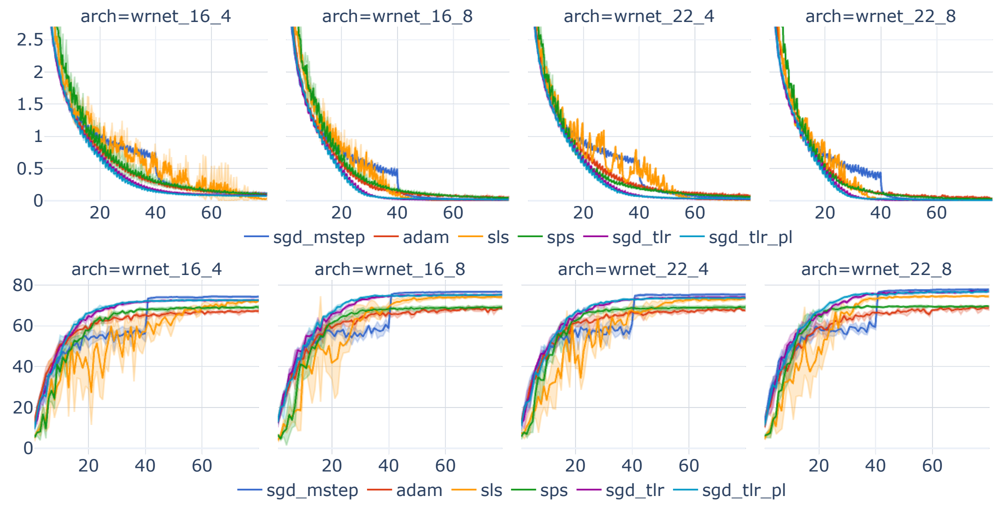
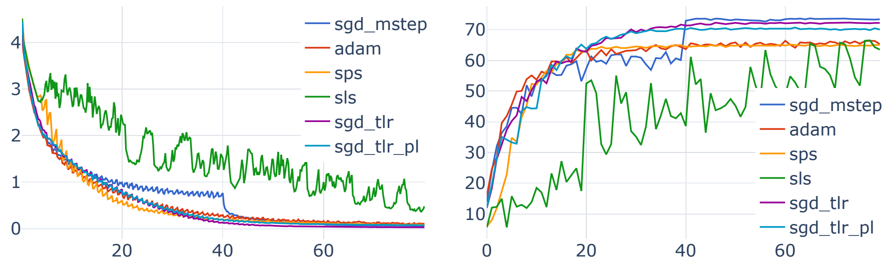


Figure 15: Loss (upper) and accuracy (bottom) curves of different optimizers for the setting CIFAR100 + WRNET_{d}_{w}. Batch size is fixed to 256.



(a) Loss Curves (b) Accuracy Curves

Figure 16: Evaluating a DenseNet architecture on CIFAR100 dataset (256 batch size and $\alpha_0 = 0.1$)

accuracy despite having increased loss compared to other optimizers. Nonetheless, the proposed algorithms present similar accuracy, while converging much faster to a well-performing minimum.

Finally, we evaluated our method on a Seq2Seq *Transformer* model with 8 heads and 3 encoder and decoder layers used for machine translation from German to English (Multi30k dataset) [3]. The corresponding training loss curve can be found in Figure 17, where it is evident that both proposed TLR algorithms converge to a minimum faster than the other reported methods. This experiment highlights the capability of the proposed method to effectively function across different tasks and architectural modules.

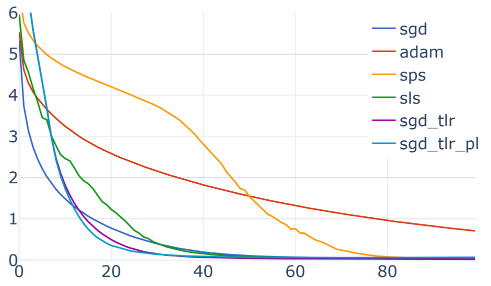


Figure 17: Loss curves (cross entropy) of different optimizers for the machine translation task with a transformer architecture.

[3]Taken from `https://github.com/pytorch/tutorials`

D.6 IMAGENET CONVERGENCE

In this section, we further investigate the reported results of the ImageNet+ResNet50 setting, as shown in Figure 4, with respect to convergence speed and generalization. The maximum attained accuracy was 75.9% and was achieved by the proposed *sgd_tlr* method, with both *sgd_mstep* and the proposed alternative *sgd_tlr_pl* having similar performance. Note that a carefully tuned scheduler or a longer training process could have achieved higher accuracy, but this strays from the goal of our paper. Nonetheless, a performance of 75.9% could be considered as competitive to the state-of-the-art results (Pytorch pre-trained ResNet50 model has an accuracy of 76.13%)[4]. In order to take a closer, more quantitative, look to the results of Figure 4, we report the epoch that an optimizer achieved a specific accuracy threshold in Table 1. To draw a better picture of the convergence behavior, we reported the attained epoch at different percentage of the maximum achieved accuracy. The maximum accuracy was reported near the end of the training process (epoch 77), but interestingly enough the proposed methods can achieve up to 95% of this maximum accuracy, namely around Top-1 72.1% accuracy, for half the epochs. The *sgd_mstep* achieves similar performance to the proposed method in the end, but its behavior and convergence speed is governed by the scheduling setup, offering no notable insights. *Adam* shows a fast converging initial phase, supporting its theoretical fast converging property, but seems to be stuck to a suboptimal solution, indicating generalization problems (combining Adam with a state-of-the-art scheduler may resolve this issue, but such a solution does not align with the research directions of this paper). Similar performance has been observed for the *sps* adaptive step-size algorithm. On the contrary, the proposed method variants (*sgd_tlr* and *sgd_tlr_pl*) show better generalization abilities, attaining as good results as the widely-used multi-step scheduler, while converging to such a result notably fast, namely achieving over 70% accuracy under 35 epochs.

	percentage (%) of maximum accuracy							
method	80	85	90	92	94	96	98	100
adam	13	19	35	-	-	-	-	-
sps	20	21	34	44	-	-	-	-
sgd_mstep	41	41	41	43	49	61	63	-
sgd_tlr	19	24	31	34	38	44	53	77
sgd_tlr_pl	18	22	28	32	37	43	58	-

Table 1: Epoch of achieved accuracy threshold across different optimizers for the ImageNet+ResNet50 setting. Accuracy thresholds are reported as percentages of the maximum reported accuracy of 75.9%. The symbol '-' denotes that the requested accuracy never achieved under the 80 epochs of training.

[4]https://pytorch.org/vision/stable/models.html