Learning with Random Learning Rates

Léonard Blier[1,2] [✉], Pierre Wolinski[1], and Yann Ollivier[2]

[1] TAU, LRI, Inria, Université Paris Sud
`pierre.wolinski@u-psud.fr`
[2] Facebook AI Research
{`leonardb,yol`}`@fb.com`

Abstract. In neural network optimization, the learning rate of the gradient descent strongly affects performance. This prevents reliable out-of-the-box training of a model on a new problem. We propose the *All Learning Rates At Once* (Alrao) algorithm for deep learning architectures: each neuron or unit in the network gets its own learning rate, randomly sampled at startup from a distribution spanning several orders of magnitude. The network becomes a mixture of slow and fast learning units. Surprisingly, Alrao performs close to SGD with an optimally tuned learning rate, for various tasks and network architectures. In our experiments, all Alrao runs were able to learn well without any tuning.

1 Introduction

Deep learning models require delicate hyperparameter tuning [1]: when facing new data or new model architectures, finding a configuration that enables fast learning requires both expert knowledge and extensive testing. This prevents deep learning models from working out-of-the-box on new problems without human intervention (AutoML setup, [2]). One of the most critical hyperparameters is the learning rate of the gradient descent [3, p. 892]. With too large learning rates, the model does not learn; with too small learning rates, optimization is slow and can lead to local minima and poor generalization [4–7].

Efficient methods with no learning rate tuning are a necessary step towards more robust learning algorithms, ideally working out of the box. Many methods were designed to directly set optimal per-parameter learning rates [8–12], such as the popular Adam optimizer. The latter comes with default hyperparameters which reach good performance on many problems and architectures; yet fine-tuning and scheduling of its learning rate is still frequently needed [13], and the default setting is specific to current problems and architecture sizes. Indeed Adam's default hyperparameters fail in some natural setups (Section 6.2). This makes it unfit in an out-of-the-box scenario.

We propose *All Learning Rates At Once* (Alrao), a gradient descent method for deep learning models that leverages redundancy in the network. Alrao uses multiple learning rates at the same time in the same network, spread across several orders of magnitude. This creates a mixture of slow and fast learning units.

Alrao departs from the usual philosophy of trying to find the "right" learning rates; instead we take advantage of the overparameterization of network-based models to produce a diversity of behaviors from which good network outputs can be built. The width of the architecture may optionally be increased to get enough units within a suitable learning rate range, but surprisingly, performance was largely satisfying even without increasing width.

Our contributions are as follows:

– We introduce Alrao, a gradient descent method for deep learning models with no learning rate tuning, leveraging redundancy in deep learning models via a range of learning rates in the same network. Surprisingly, Alrao does manage to learn well over a range of problems from image classification, text prediction, and reinforcement learning.
– In our tests, Alrao's performance is always close to that of SGD with the optimal learning rate, without any tuning.
– Alrao combines performance with *robustness*: not a single run failed to learn with the default learning rate range we used. In contrast, our parameter-free baseline, Adam with default hyperparameters, is not reliable across the board.
– Alrao vindicates the role of redundancy in deep learning: having enough units with a suitable learning rate is sufficient for learning.

Acknowledgments. We would like to thank Corentin Tallec for his technical help and extensive remarks. We thank Olivier Teytaud for pointing useful references, Hervé Jégou for advice on the text, and Léon Bottou, Guillaume Charpiat, and Michèle Sebag for their remarks on our ideas.

2 Related Work

Redundancy in deep learning. Alrao specifically exploits the redundancy of units in network-like models. Several lines of work underline the importance of such redundancy in deep learning. For instance, dropout [14] relies on redundancy between units. Similarly, many units can be pruned after training without affecting accuracy [15–18]. Wider networks have been found to make training easier [19–21], even if not all units are useful a posteriori.

The *lottery ticket hypothesis* [22, 23] posits that "large networks that train successfully contain subnetworks that—when trained in isolation—converge in a comparable number of iterations to comparable accuracy'. This subnetwork is the *lottery ticket winner*: the one which had the best initial values. In this view, redundancy helps because a larger network has a larger probability to contain a suitable subnetwork. Alrao extends this principle to the learning rate.

Learning rate tuning. Automatically using the "right" learning rate for each parameter was one motivation behind "adaptive" methods such as RMSProp [8], AdaGrad [9] or Adam [10]. Adam with its default setting is currently considered the default method in many works [24]. However, further global adjustment of the Adam learning rate is common [25]. Other heuristics for setting the learning rate have been proposed [11]; these heuristics often start with the idea of approximating a second-order Newton step to define an optimal learning rate [12]. Indeed, asymptotically, an arguably optimal preconditioner is either the Hessian of the loss (Newton method) or the Fisher information matrix [26]. Another approach is to perform gradient descent on the learning rate itself through the whole training procedure [27–32]. Despite being around since the 80's [27], this has not been widely adopted, because of sensitivity to hyperparameters such as the meta-learning rate or the initial learning rate [33]. Of all these methods, Adam is probably the most widespread at present [24], and we use it as a baseline.

The learning rate can also be optimized within the framework of architecture or hyperparameter search, using methods from from reinforcement learning [1,34,35], evolutionary algorithms [36–38], Bayesian optimization [39], or differentiable architecture search [40]. Such methods are resource-intensive and do not allow for finding a good learning rate in a single run.

3 Motivation and Outline

We first introduce the general ideas behind Alrao. The detailed algorithm is explained in Section 4 and in Algorithm 1. We also release a Pytorch [41] implementation, including tutorials: `http://github.com/leonardblier/alrao`.

Different learning rates for different units. Instead of using a single learning rate for the model, Alrao samples once and for all a learning rate for each *unit* in the network. These rates are taken from a log-uniform distribution in an interval $[\eta_{\min}; \eta_{\max}]$. The log-uniform distribution produces learning rates spread over several order of magnitudes, mimicking the log-uniform grids used in standard grid searches on the learning rate.

A *unit* corresponds for example to a feature or neuron for fully connected networks, or to a channel for convolutional networks. Thus we build "slow-learning" and "fast-learning" units. In contrast, with per-parameter learning rates, every unit would have a few incoming weights with very large learning rates, and possibly diverge.

Intuition. Alrao is inspired by the fact that not all units in a neural network end up being useful. Our idea is that in a large enough network with learning rates sampled randomly per unit, a sub-network made of units with a good learning rate will learn well, while the units with a wrong learning rate will produce useless values and just be ignored by the rest of the network. Units with too small learning rates will not learn anything and stay close to their initial values; this does not hurt training (indeed, even leaving some weights at their initial values, corresponding to a learning rate 0, does not hurt training). Units with a too large learning rate may produce large activation values, but those will be mitigated by subsequent normalizing mechanisms in the computational graph, such as sigmoid/tanh activations or BatchNorm.

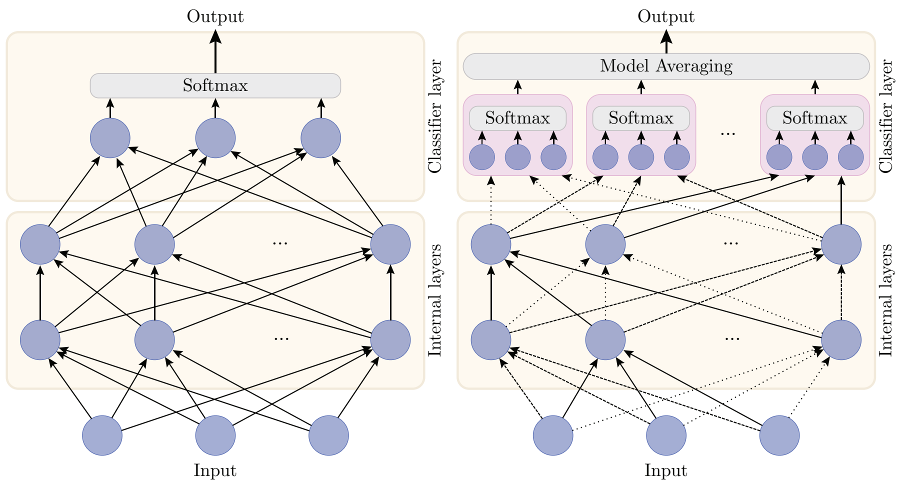


Fig. 1: Left: a standard fully connected neural network for a classification task with three classes, made of several internal layers and an output layer. Right: Alrao version of the same network. The single classifier layer is replaced with a set of parallel copies of the original classifier, averaged with a model averaging method. Each unit uses its own learning rate for its incoming weights (represented by different styles of arrows).

Alrao can be interpreted within the *lottery ticket hypothesis* [22]: viewing the per-unit learning rates of Alrao as part of the initialization, this hypothesis suggests that in a wide enough network, there will be a sub-network whose initialization (both values and learning rate) leads to good convergence.

Slow and fast learning units for the output layer. Sampling a learning rate per unit at random in the last layer would not make sense. For classification, each unit in the last layer represents a single category: using different learning rates for these units would favor some categories during learning. Moreover for scalar regression tasks there is only one output unit, thus we would be back to selecting a single learning rate.

The simplest way to obtain the best of several learning rates for the last layer, without relying on heuristics to guess an optimal value, is to use *model averaging* over several copies of the output layer (Fig. 1), each copy trained with its own learning rate from the interval $[\eta_{\min}; \eta_{\max}]$. All these untied copies of the output layer share the same Alrao internal layers (Fig. 1). This can be seen as a smooth form of model selection or grid-search over the output layer learning rate; actually, this part of the architecture can even be dropped after a few epochs, as the model averaging quickly concentrates on one model.

Increasing network width. With Alrao, neurons with unsuitable learning rates will not learn: those with too large learning rates might learn no useful signal, while those with too small learning rates will learn too slowly. Thus, Alrao may reduce the *effective width* of the network to only a fraction of the actual architecture width, depending on $[\eta_{\min}; \eta_{\max}]$. This may be compensated by multiplying the width of the network by a factor γ. Our first intuition was that $\gamma > 1$ would be necessary; still Alrao turns out to work well even without width augmentation.

4 All Learning Rates At Once: Description

4.1 Notation

We now describe Alrao more precisely for deep learning models with softmax output, on classification tasks; the case of regression is similar.

Let $\mathcal{D} = \{(x_1, y_1), ..., (x_N, y_N)\}$, with $y_i \in \{1, ..., K\}$, be a classification dataset. The goal is to predict the y_i given the x_i, using a deep learning model Φ_θ. For each input x, $\Phi_\theta(x)$ is a probability distribution over $\{1, ..., K\}$, and we want to minimize the categorical cross-entropy loss ℓ over the dataset: $\frac{1}{N} \sum_i \ell(\Phi_\theta(x_i), y_i)$.

We denote $\log\text{-}\mathcal{U}(\cdot; \eta_{\min}, \eta_{\max})$ the *log-uniform* probability distribution on an interval $[\eta_{\min}; \eta_{\max}]$. Namely, if $\eta \sim \log\text{-}\mathcal{U}(\cdot; \eta_{\min}, \eta_{\max})$, then $\log \eta$ is uniformly distributed between $\log \eta_{\min}$ and $\log \eta_{\max}$. Its density function is $\log\text{-}\mathcal{U}(\eta; \eta_{\min}, \eta_{\max}) = \frac{1}{\eta} \frac{\mathbb{1}_{\eta_{\min} \le \eta \le \eta_{\max}}}{\log(\eta_{\max}) - \log(\eta_{\min})}$.

Algorithm 1 Alrao-SGD for model $\Phi_\theta = C_{\theta^{\text{out}}} \circ \phi_{\theta^{\text{r}}}$ with N_{out} classifiers and learning rates in $[\eta_{\min}; \eta_{\max}]$

```
1: $a_j \leftarrow 1/N_{\text{out}}$ for each $1 \le j \le N_{\text{out}}$    ▷ Initialize the $N_{\text{out}}$ model averaging weights $a_j$
2: $\Phi_\theta^{\text{Alrao}}(x) := \sum_{j=1}^{N_{\text{out}}} a_j \, C_{\theta_j^{\text{out}}}(\phi_{\theta^{\text{int}}}(x))$    ▷ Define the Alrao architecture
3: for all layers $l$, for all unit $i$ in layer $l$ do
4:     Sample $\eta_{l,i} \sim \log\text{-}\mathcal{U}(.; \eta_{\min}, \eta_{\max})$.    ▷ Sample a learning rate for each unit
5: for all Classifiers $j$, $1 \le j \le N_{\text{out}}$ do
6:     Define $\log \eta_j = \log \eta_{\min} + \frac{j-1}{N_{\text{out}}-1} \log \frac{\eta_{\max}}{\eta_{\min}}$.    ▷ Set a learning rate for each classifier
7: while Stopping criterion is False do
8:     $z_t \leftarrow \phi_{\theta^{\text{int}}}(x_t)$    ▷ Store the output of the last internal layer
9:     for all layers $l$, for all unit $i$ in layer $l$ do
10:        $\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \cdot \nabla_{\theta_{l,i}} \ell(\Phi_\theta^{\text{Alrao}}(x_t), y_t)$    ▷ Update the repr. netw. weights
11:    for all Classifier $j$ do
12:        $\theta_j^{\text{out}} \leftarrow \theta_j^{\text{out}} - \eta_j \cdot \nabla_{\theta_j^{\text{out}}} \ell(C_{\theta_j^{\text{out}}}(z_t), y_t)$    ▷ Update the classifiers' weights
13:    $a \leftarrow \texttt{ModelAveraging}(a, (C_{\theta_i^{\text{out}}}(z_t))_i, y_t)$    ▷ Update the model averaging weights.
14:    $t \leftarrow t + 1 \bmod N$
```

4.2 Alrao Architecture

Multiple Alrao output layers. A deep learning model Φ_θ for classification can be decomposed into two parts: first, *internal layers* compute some function $z = \phi_{\theta^{\text{int}}}(x)$ of the inputs x, fed to a final *output (classifier) layer* $C_{\theta^{\text{out}}}$, so that the overall network output is $\Phi_\theta(x) := C_{\theta^{\text{out}}}(\phi_{\theta^{\text{int}}}(x))$. For a classification task with K categories, the output layer $C_{\theta^{\text{out}}}$ is defined by $C_{\theta^{\text{out}}}(z) := \text{softmax} \circ \left(W^T z + b\right)$ with $\theta^{\text{out}} := (W, b)$, and $\text{softmax}(u_1, ..., u_K)_k := e^{u_k} / (\sum_i e^{u_i})$.

In Alrao, we build multiple copies of the original output layer, with different learning rates for each, and then use a model averaging method among them. The averaged classifier and the overall Alrao model are:

$$C^{\text{Alrao}}_{\theta^{\text{out}}}(z) := \sum_{j=1}^{N_{\text{out}}} a_j \, C_{\theta^{\text{out}}_j}(z), \qquad \Phi^{\text{Alrao}}_{\theta}(x) := C^{\text{Alrao}}_{\theta^{\text{out}}}(\phi_{\theta^{\text{int}}}(x)) \tag{1}$$

where the $C_{\theta^{\text{out}}_j}$ are copies of the original classifier layer, with non-tied parameters, and $\theta^{\text{out}} := (\theta^{\text{out}}_1, ..., \theta^{\text{out}}_{N_{\text{out}}})$. The a_j are the parameters of the model averaging, with $0 \leq a_j \leq 1$ and $\sum_j a_j = 1$. The a_j are not updated by gradient descent, but via a model averaging method from the literature (see below).

Increasing the width of internal layers. As explained in Section 3, we may compensate the effective width reduction in Alrao by multiplying the width of the network by a factor γ. This means multiplying the number of units (or filters for a convolutional layer) of all internal layers by γ.

4.3 Alrao Update for the Internal Layers: A Random Learning Rate for Each Unit

In the internal layers, for each unit i in each layer l, a learning rate $\eta_{l,i}$ is sampled from the probability distribution $\log\text{-}\mathcal{U}(.; \eta_{\min}, \eta_{\max})$, once and for all at the beginning of training. [1]

The incoming parameters of each unit in the internal layers are updated in the usual SGD way, only with per-unit learning rates (Eq. 2): for each unit i in each layer l, its incoming parameters are updated as:

$$\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \cdot \nabla_{\theta_{l,i}} \ell(\Phi^{\text{Alrao}}_{\theta}(x), y) \tag{2}$$

where $\Phi^{\text{Alrao}}_{\theta}$ is the Alrao loss (1) defined above.

What constitutes a *unit* depends on the type of layers in the model. In a fully connected layer, each component of a layer is considered as a unit for Alrao: all incoming weights of the same unit share the same Alrao learning rate. On the other hand, in a convolutional layer we consider each convolution filter

[1] With learning rates resampled at each time, each step would be, in expectation, an ordinary SGD step with learning rate $\mathbb{E}\eta_{l,i}$, thus just yielding an ordinary SGD trajectory with more variance.

as constituting a unit: there is one learning rate per filter (or channel), thus preserving translation-invariance over the input image. In LSTMs, we apply the same learning rate to all components in each LSTM cell (thus the vector of learning rates is the same for input gates, for forget gates, etc.).

We set a learning rate *per unit*, rather than per parameter. Otherwise, every unit would have some parameters with large learning rates, and we would expect even a few large incoming weights to be able to derail a unit. Having diverging parameters within every unit is hurtful, while having diverging units in a layer is not necessarily hurtful since the next layer can learn to disregard them.

4.4 Alrao Update for the Output Layer: Model Averaging from Output Layers Trained with Different Learning Rates

Learning the output layers. The j-th copy $C_{\theta_j^{\mathrm{out}}}$ of the classifier layer is attributed a learning rate η_j defined by $\log \eta_j := \log \eta_{\min} + \frac{j-1}{N_{\mathrm{out}}-1} \log\left(\frac{\eta_{\max}}{\eta_{\min}}\right)$, so that the classifiers' learning rates are log-uniformly spread on the interval $[\eta_{\min}; \eta_{\max}]$. Then the parameters θ_j^{out} of each classifier j are updated as if this classifier alone was the only output of the model:

$$\theta_j^{\mathrm{out}} \leftarrow \theta_j^{\mathrm{out}} - \eta_j \cdot \nabla_{\theta_j^{\mathrm{out}}} \ell(C_{\theta_j^{\mathrm{out}}}(\phi_{\theta^{\mathrm{int}}}(x)), y), \tag{3}$$

(still sharing the same internal layers $\phi_{\theta^{\mathrm{int}}}$). This ensures that classifiers with low weights a_j still learn, and is consistent with model averaging philosophy. Algorithmically this requires differentiating the loss N_{out} times with respect to the last layer, but no additional backpropagations through the internal layers.

Model averaging. To set the weights a_j, several model averaging techniques are available, such as Bayesian Model Averaging [42]. We use the *Switch* model averaging [43], a Bayesian method which is both simple, principled, and very responsive to changes in performance of the various models. After each mini-batch, the switch computes a modified posterior distribution (a_j) over the classifiers. This computation is directly taken from [43].

Additional experiments show that the model averaging method acts like a smooth model selection procedure: after only a few hundreds gradient steps, a single output layer is selected, with its parameter a_j very close to 1. Actually, Alrao's performance is unchanged if the extraneous output layer copies are thrown away when the posterior weight a_j of one of the copies gets close to 1.

5 Experimental Setup

We tested Alrao on various convolutional networks for image classification (Imagenet and CIFAR10), on LSTMs for text prediction, and on reinforcement learning problems. We always use the same learning rate interval $[10^{-5}; 10]$, corresponding to the values we would have tested in a grid search, and 10 Alrao output layer copies, for every task.

Table 1: Performance of Alrao, SGD with tuned learning rate, and Adam with its default setting. Three convolutional models are reported for image classification on CIFAR10, three others for ImageNet, one recurrent model for character prediction (Penn Treebank), and two experiments on RL problems. Four of the image classification architectures are further tested with a width multiplication factor $\gamma = 3$. Alrao learning rates are taken in a wide, a priori reasonable interval $[\eta_{\min}; \eta_{\max}] = [10^{-5}; 10]$, and the optimal learning rate for SGD is chosen in the set $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$. Each experiment is run 10 times (CIFAR10 and RL), 5 times (PTB) or 1 time (ImageNet); the confidence intervals report the standard deviation over these runs. For RL tasks, the return has to be maximized, not minimized.

Model	SGD with optimal LR			Adam - Default		Alrao	
	LR	Loss	Top1 (%)	Loss	Top1 (%)	Loss	Top1 (%)
CIFAR10							
MobileNet	0.1	$0.37 \pm .01$	$90.2 \pm .3$	$1.01 \pm .95$	78 ± 11	$0.42 \pm .02$	$88.1 \pm .6$
MobileNet, $\gamma = 3$	0.1	$0.33 \pm .01$	$90.3 \pm .5$	$0.32 \pm .02$	$90.8 \pm .4$	$0.35 \pm .01$	$89.0 \pm .6$
GoogLeNet	0.01	$0.45 \pm .05$	$89.6 \pm 1.$	$0.47 \pm .04$	$89.8 \pm .4$	$0.47 \pm .03$	$88.9 \pm .8$
GoogLeNet, $\gamma = 3$	0.1	$0.34 \pm .02$	$90.5 \pm .8$	$0.41 \pm .02$	$88.6 \pm .6$	$0.37 \pm .01$	$89.8 \pm .8$
VGG19	0.1	$0.42 \pm .02$	$89.5 \pm .2$	$0.43 \pm .02$	$88.9 \pm .4$	$0.45 \pm .03$	$87.5 \pm .4$
VGG19, $\gamma = 3$	0.1	$0.35 \pm .01$	$90.0 \pm .6$	$0.37 \pm .01$	$89.5 \pm .8$	$0.381 \pm .004$	$88.4 \pm .7$
ImageNet							
AlexNet	0.01	2.15	53.2	6.91	0.10	2.56	43.2
Densenet121	1	1.35	69.7	1.39	67.9	1.41	67.3
ResNet50	1	1.49	67.4	1.39	67.1	1.42	67.5
ResNet50, $\gamma = 3$	-	-	-	1.99	60.8	1.33	70.9
Penn Treebank							
LSTM	1	$1.566 \pm .003$	$66.1 \pm .1$	$1.587 \pm .005$	$65.6 \pm .1$	$1.706 \pm .004$	$63.4 \pm .1$
RL		Return		Return		Return	
Pendulum	0.0001	-372 ± 24		-414 ± 64		-371 ± 36	
LunarLander	0.1	188 ± 23		155 ± 23		186 ± 45	

We compare Alrao to SGD with an optimal learning rate selected in the set $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$, and, as a tuning-free baseline, to Adam with its default setting ($\eta = 10^{-3}, \beta_1 = 0.9, \beta_2 = 0.999$), arguably the current default method [24].

The results are presented in Table 1. Fig. 2 presents learning curves for AlexNet and Resnet50 on ImageNet.

5.1 Image Classification on ImageNet and CIFAR10

For image classification, we used the ImageNet [44] and CIFAR10 [45] datasets. The ImageNet dataset is made of 1,283,166 training and 60,000 testing data; we split the training set into a smaller training set and a validation set with 60,000 samples. We do the same on CIFAR10: the 50,000 training samples are split into 40,000 training samples and 10,000 validation samples.

For each architecture, training was stopped when the validation loss had not improved for 20 epochs. The epoch with best validation loss was selected and the corresponding model tested on the test set. The inputs are normalized, and training used data augmentation: random cropping and random horizontal flipping. For CIFAR10, each setting was run 10 times: the confidence intervals presented are the standard deviation over these runs. For ImageNet, because of high computation time, we performed only a single run per experiment.

We tested Alrao on several standard architectures. On ImageNet, we tested Resnet50 [46], Densenet121 [47], and Alexnet [48], using the default Pytorch implementation. On CIFAR10, we tested GoogLeNet [49], VGG19 [50], and MobileNet [51], as implemented in [52]. We also tested wider architectures, with a width multiplication factor $\gamma = 3$. On the largest model, Resnet50 on ImageNet with triple width, systematic SGD learning rate grid search was not performed due to the excessive computational burden, hence the omitted value in Tab. 1.

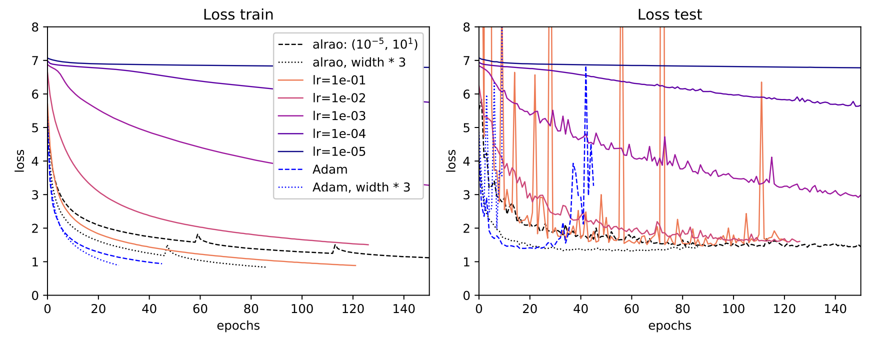
5.2 Other Tasks: Text Prediction, Reinforcement Learning

Text prediction on Penn TreeBank. To test Alrao on other kinds of tasks, we first used a recurrent neural network for text prediction on the Penn Treebank (PTB) [53] dataset. The Alrao experimental procedure is the same as above.

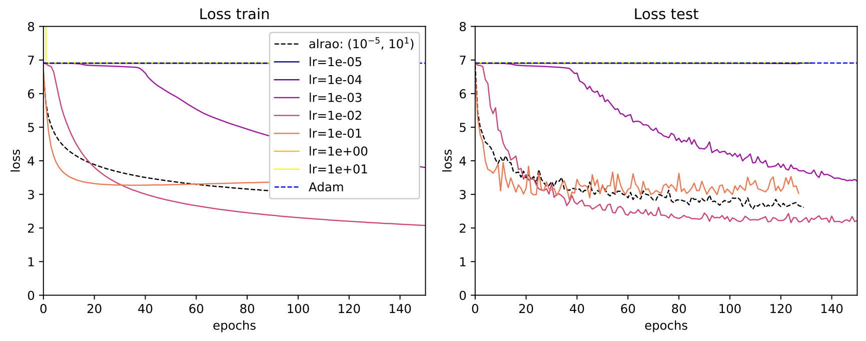
The loss in Table 1 is given in bits per character and the accuracy is the proportion of correct character predictions. The model is a two-layer LSTM [54] with an embedding size of 100, and 100 hidden units. A dropout layer with rate 0.2 is included before the decoder. The training set is divided into 20 minibatchs. Gradients are computed via truncated backprop through time [55] with truncation every 70 characters.

The model was trained for *character* prediction rather than word prediction. This is technically easier for Alrao implementation: since Alrao uses copies of the output layer, memory issues arise for models with most parameters on the output layer. Word prediction (10,000 classes on PTB) requires many more output parameters than character prediction; see Section 7.

Reinforcement learning tasks. Next, we tested Alrao on two standard reinforcement learning problems: the Pendulum and Lunar Lander environments from OpenAI Gym [56]. We use standard deep Q-learning [57]. The Q-network is a standard MLP with 2 hidden layers. The experimental setting is the same as above, with regressors instead of classifiers on the output layer. For each environment, we select the best epoch on validation runs, and then report the return of the selected model on new test runs in that environment.



(a) Resnet50 trained on ImageNet.



(b) AlexNet trained on ImageNet

Fig. 2: Learning curves for Alrao, SGD with various learning rates, and Adam with its default setting, on ImageNet. Left: training loss; right: test loss. Curves are interrupted by the early stopping criterion. Alrao's performance is comparable to the optimal SGD learning rate.

6 Performance and Robustness of Alrao

6.1 Alrao Compared to SGD with Optimal Learning Rate

First, Alrao does manage to learn; this was not obvious a priori.

Second, SGD with an optimally tuned learning rate usually performs better than Alrao. This can be expected when comparing a tuning-free method with a method that tunes the hyperparameter in hindsight.

Still, the difference between Alrao and optimally-tuned SGD is reasonably small across every setup, even with wide intervals $[\eta_{\min}; \eta_{\max}]$, with a somewhat larger gap in one case (AlexNet on ImageNet). Notably, this occurs even though SGD achieves good performance only for a few learning rates within the interval $[\eta_{\min}; \eta_{\max}]$. With $\eta_{\min} = 10^{-5}$ and $\eta_{\max} = 10$, among the 7 SGD learning rates tested ($10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$, and 10), only three are able to learn with

AlexNet, and only one is better than Alrao (Fig. 2b); with ResNet50, only three are able to learn well, and only two of them achieve performance similar to Alrao (Fig. 2a); on the Pendulum environment, only two are able to learn well, only one of which converges as fast as Alrao.

Thus, surprisingly, Alrao manages to learn at a nearly optimal rate, even though most units in the network have learning rates unsuited for SGD.

6.2 Robustness of Alrao, and Comparison to Default Adam

Overall, Alrao learns reliably in every setup in Table 1. Moreover, this is quite stable over the course of learning: Alrao curves shadow optimal SGD curves over time (Fig. 2).

Often, Adam with its default parameters almost matches optimal SGD, but this is not always the case. Over the 13 setups in Table 1, default Adam gives a significantly poor performance in three cases. One of those is a pure optimization issue: with AlexNet on ImageNet, optimization does not start with the default parameters (Fig. 2b). The other two cases are due to strong overfit despite good train performance: MobileNet on CIFAR and ResNet with increased width on ImageNet.

In two further cases, Adam achieves good validation performance in Table 1, but actually overfits shortly after its peak score: ResNet (Fig. 2a) and DenseNet, [24, 58].

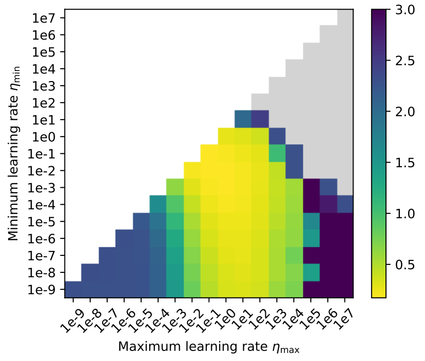
Overall, default Adam tends to give slightly better results than Alrao when it works, but does not learn reliably with its default hyperparameters. It can exhibit two kinds of lack of robustness: optimization failure, and overfit or non-robustness over the course of learning. On the other hand, every single run of Alrao reached reasonably close-to-optimal performance. Alrao also performs steadily over the course of learning (Fig. 2).

6.3 Sensitivity Study to $[\eta_{\min}; \eta_{\max}]$

We claim to remove a hyperparameter, the learning rate, but replace it with two hyperparameters $\eta_{\min}$ and $\eta_{\max}$. Formally, this is true. But a systematic study of the impact of these two hyperparameters (Fig. 3) shows that the sensitivity to $\eta_{\min}$ and $\eta_{\max}$ is much lower than the original sensitivity to the learning rate.

To assess this, we tested every combination of $\eta_{\min}$ and $\eta_{\max}$ in a grid from 10^{-9} to 10^{7} on GoogLeNet for CIFAR10 (left plot in Fig. 3, with SGD on the diagonal). The largest satisfactory learning rate for SGD is 1 (diagonal on Fig. 3). Unsurprisingly, if all the learning rates in Alrao are too large, or all too small, then Alrao fails (rightmost and leftmost zones in Fig. 3). Extremely large learning rates diverge numerically, both for SGD and Alrao.

On the other hand, Alrao converges as soon as $[\eta_{\min}; \eta_{\max}]$ contains a reasonable learning rate (central zone Fig. 3), even with values of $\eta_{\max}$ for which SGD fails. A wide range of choices for $[\eta_{\min}; \eta_{\max}]$ will contain one good learning rate and achieve close-to-optimal performance. Thus, as a general rule, we



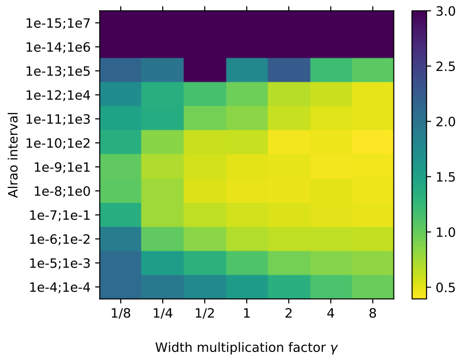


Fig. 3: Influence of $[\eta_{\min}; \eta_{\max}]$ and of network width on Alrao performance, with GoogLeNet on CIFAR10. Results are reported after 15 epochs, and averaged on three runs. Left plot: each point with coordinates $[\eta_{\min}; \eta_{\max}]$ below the diagonal represents the loss for Alrao with this interval. Points (η, η) on the diagonal represent standard SGD with learning rate η. Grey squares represent numerical divergence (NaN). Alrao works as soon as $[\eta_{\min}; \eta_{\max}]$ contains at least one suitable learning rate. Right plot: varying network width.

recommend to just use an interval containing all the learning rates that would have been tested in a grid search, e.g., 10^{-5} to 10.

For a fixed network size, one might expect Alrao to perform worse with large intervals $[\eta_{\min}; \eta_{\max}]$, as most units would become useless. On the other hand, in a larger network, many units would have extreme learning rates, which might disturb learning. We tested how increasing or decreasing network width changes Alrao's sensitivity to $[\eta_{\min}; \eta_{\max}]$ (right plot of Fig. 3 for Alrao). The sensitivity of Alrao to $[\eta_{\min}; \eta_{\max}]$ decreases markedly with network width. For instance, a wide interval $[\eta_{\min}; \eta_{\max}] = [10^{-12}; 10^{4}]$ works reasonably well with an 8-fold network, even though most units receive unsuitable learning rates.

So, even if the choice of $\eta_{\min}$ and $\eta_{\max}$ is important, the results are much more stable to varying these two hyperparameters than to the original learning rate, especially with large networks.

7 Discussion, Limitations, and Perspectives

Alrao specifically exploits redundancy between units in deep learning models, relying on the overall network approach of combining a large number of units built for diversity of behavior. Alrao would not make sense in a classical convex optimization setting. That Alrao works at all is already informative about some phenomena at play in deep neural networks.

Alrao can make lengthy SGD learning rate sweeps unnecessary on large models, such as the triple-width ResNet50 for ImageNet above. Incidentally, in our experiments, wider networks provided increased performance both for SGD and

Alrao (Table 1 and Fig. 3): network size is still a limiting factor for the models used, independently of the algorithm.

Increased number of parameters for the classification layer. Since Alrao modifies the output layer of the optimized model, the number of parameters in the classification layer is multiplied by the number of classifier copies. (The number of parameters in the internal layers is unchanged.) This is a limitation for models with most parameters in the classifier layer.

On CIFAR10 (10 classes), the number of parameters increases by less than 5% for the models used. On ImageNet (1000 classes), it increases by 50–100% depending on the architecture. On Penn Treebank, the number of parameters increased by 26% in our setup (at character level); working at word level it would have increased fivefold.

This can be mitigated by handling the copies of the classifiers on distinct computing units: in Alrao these copies work in parallel given the internal layers. Moreover, the additional output layer copies may be thrown away early in training. Finally, models with a large number of output classes usually rely on other parameterizations than a direct softmax, such as a hierarchical softmax (see references in [59]); Alrao can be used in conjunction with such methods.

Multiple output layer copies and expressiveness. Using several copies of the output layer in Alrao formally provides more expressiveness to the model, as it creates a larger architecture with more parameters. We performed two control experiments to check that Alrao's performance does not just stem from this. First, we performed ablation of the output layer copies in Alrao after one epoch, only keeping the copy with the highest model averaging weight a_i: the learning curves are identical. Second, we trained default Adam using copies of the output layer (all with the same Adam default learning rate): the learning curves are identical to Adam on the unmodified architecture. Thus, the copies of the output layer do not bring any useful added expressiveness.

Learning rate schedules, other optimizers, other hyperparameters... Learning rate schedules are often effective [60]. We did not use them here: this may partially explain why the results in Table 1 are worse than the state-of-the-art. One might have hoped that the diversity of learning rates in Alrao would effortlessly bring it to par with step size schedules, but the results above do not support this. Still, nothing prevents using a scheduler together with Alrao, e.g., by dividing all Alrao learning rates by a time-dependent constant.

The Alrao idea can also be used with other optimizers than SGD, such as Adam. We tested combining Alrao and Adam, and found the combination less reliable than standard Alrao: curves on the training set mostly look good, but the method quickly overfits.

The Alrao idea could be used on other hyperparameters as well, such as momentum. However, with more hyperparameters initialized randomly for each unit, the fraction of units having suitable values for all their hyperparameters simultaneously will quickly decrease.

8 Conclusion

Applying stochastic gradient descent with multiple learning rates for different units is surprisingly resilient in our experiments, and provides performance close to SGD with an optimal learning rate, as soon as the range of random learning rates is not excessive. Alrao could save time when testing deep learning models, opening the door to more out-of-the-box uses of deep learning.

References

1. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
2. Guyon, I., Chaabane, I., Escalante, H.J., Escalera, S., Jajetic, D., Lloyd, J.R., Macià, N., Ray, B., Romaszko, L., Sebag, M., et al.: A brief review of the ChaLearn AutoML challenge: any-time any-dataset learning without human intervention. In: Workshop on Automatic Machine Learning. (2016) 21–30
3. Theodoridis, S.: Machine learning: a Bayesian and optimization perspective. Academic Press (2015)
4. Jastrzebski, S., Kenton, Z., Arpit, D., Ballas, N., Fischer, A., Bengio, Y., Storkey, A.: Three factors influencing minima in sgd. arXiv preprint arXiv:1711.04623 (2017)
5. Kurita, K.: Learning Rate Tuning in Deep Learning: A Practical Guide — Machine Learning Explained (2018)
6. Mack, D.: How to pick the best learning rate for your machine learning project (2016)
7. Surmenok, P.: Estimating an Optimal Learning Rate For a Deep Neural Network (2017)
8. Tieleman, T., Hinton, G.: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning **4**(2) (2012) 26–31
9. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. JMLR **12** (2011) 2121–2159
10. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. In: International Conference on Learning Representations. (2015)
11. Schaul, T., Zhang, S., LeCun, Y.: No more pesky learning rates. In: International Conference on Machine Learning. (2013) 343–351
12. LeCun, Y., Bottou, L., Orr, G.B., Müller, K.R.: Efficient backprop. In: Neural Networks: Tricks of the Trade. Springer (1998) 9–50
13. Denkowski, M., Neubig, G.: Stronger baselines for trustable results in neural machine translation. arXiv preprint arXiv:1706.09733 (2017)
14. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15** (2014) 1929–1958
15. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In Touretzky, D.S., ed.: NIPS 2. Morgan-Kaufmann (1990) 598–605
16. Han, S., Mao, H., Dally, W.J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv preprint arXiv:1510.00149 (2015)

17. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both Weights and Connections for Efficient Neural Networks. In: NIPS. (2015)
18. See, A., Luong, M.T., Manning, C.D.: Compression of neural machine translation models via pruning. CoNLL 2016 (2016) 291
19. Bengio, Y., Roux, N.L., Vincent, P., Delalleau, O., Marcotte, P.: Convex neural networks. In Weiss, Y., Schölkopf, B., Platt, J.C., eds.: Advances in Neural Information Processing Systems 18. MIT Press (2006) 123–130
20. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
21. Zhang, C., Bengio, S., Hardt, M., Recht, B., Vinyals, O.: Understanding deep learning requires rethinking generalization. (2017)
22. Frankle, J., Carbin, M.: The Lottery Ticket Hypothesis: Finding Small, Trainable Neural Networks. arXiv preprint arXiv:1704.04861 (mar 2018)
23. Frankle, J., Dziugaite, G.K., Roy, D.M., Carbin, M.: The lottery ticket hypothesis at scale (2019)
24. Wilson, A.C., Roelofs, R., Stern, M., Srebro, N., Recht, B.: The marginal value of adaptive gradient methods in machine learning. In: NIPS. (2017) 4148–4158
25. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision (ECCV). (2018) 19–34
26. Amari, S.i.: Natural gradient works efficiently in learning. Neural Comput. **10** (February 1998) 251–276
27. Jacobs, R.A.: Increased rates of convergence through learning rate adaptation. Neural networks **1**(4) (1988) 295–307
28. Schraudolph, N.N.: Local gain adaptation in stochastic gradient descent. (1999)
29. Mahmood, A.R., Sutton, R.S., Degris, T., Pilarski, P.M.: Tuning-free step-size adaptation. In: Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on, IEEE (2012) 2121–2124
30. Maclaurin, D., Duvenaud, D., Adams, R.: Gradient-based hyperparameter optimization through reversible learning. In: International Conference on Machine Learning. (2015) 2113–2122
31. Massé, P.Y., Ollivier, Y.: Speed learning on the fly. arXiv preprint arXiv:1511.02540 (2015)
32. Baydin, A.G., Cornish, R., Rubio, D.M., Schmidt, M., Wood, F.: Online learning rate adaptation with hypergradient descent. In: International Conference on Learning Representations. (2018)
33. Erraqabi, A., Le Roux, N.: Combining adaptive algorithms and hypergradient method: a performance and robustness study. (2018)
34. Baker, B., Gupta, O., Naik, N., Raskar, R.: Designing neural network architectures using reinforcement learning. arXiv preprint arXiv:1611.02167 (2016)
35. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. JMLR **18**(1) (2017) 6765–6816
36. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary computation **10**(2) (2002) 99–127
37. Jozefowicz, R., Zaremba, W., Sutskever, I.: An empirical exploration of recurrent network architectures. In: International Conference on Machine Learning. (2015) 2342–2350
38. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: Proceedings of the 34th

International Conference on Machine Learning-Volume 70, JMLR. org (2017) 2902–2911
39. Bergstra, J., Yamins, D., Cox, D.D.: Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. (2013)
40. Liu, H., Simonyan, K., Yang, Y.: Darts: Differentiable architecture search. arXiv preprint arXiv:1806.09055 (2018)
41. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS-W. (2017)
42. Wasserman, L.: Bayesian Model Selection and Model Averaging. Journal of Mathematical Psychology **44** (2000)
43. Van Erven, T., Grünwald, P., De Rooij, S.: Catching up faster by switching sooner: A predictive approach to adaptive estimation with an application to the AIC-BIC dilemma. Journal of the Royal Statistical Society: Series B (Statistical Methodology) **74**(3) (2012) 361–417
44. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09. (2009)
45. Krizhevsky, A.: Learning Multiple Layers of Features from Tiny Images. (2009)
46. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: ICCV. (2016) 770–778
47. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: CVPR. Volume 1. (2017) 3
48. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997 (2014)
49. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: ICCV. (2015) 1–9
50. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR **abs/1409.1556** (2014)
51. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
52. Kianglu: pytorch-cifar (2018)
53. Marcus, M.P., Marcinkiewicz, M.A., Santorini, B.: Building a large annotated corpus of english: The penn treebank. Comput. Linguist. **19**(2) (June 1993) 313–330
54. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8) (1997) 1735–1780
55. Werbos, P.J.: Backpropagation through time: what it does and how to do it. Proceedings of the IEEE **78**(10) (1990) 1550–1560
56. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
57. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540) (2015) 529
58. Keskar, N.S., Socher, R.: Improving generalization performance by switching from Adam to SGD. arXiv preprint arXiv:1712.07628 (2017)
59. Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., Wu, Y.: Exploring the limits of language modeling. arXiv preprint arXiv:1602.02410 (2016)
60. Bengio, Y.: Practical recommendations for gradient-based training of deep architectures. In: Neural networks: Tricks of the trade. Springer (2012) 437–478