Learning with Random Learning Rates

Léonard Blier[*][†][‡] leonardb@fb.com
Pierre Wolinski[†][‡] pierre.wolinski@u-psud.fr
Yann Ollivier[*] yol@fb.com

Abstract

Hyperparameter tuning is a bothersome step in the training of deep learning models. One of the most sensitive hyperparameters is the learning rate of the gradient descent. We present the *All Learning Rates At Once* (Alrao) optimization method for neural networks: each unit or feature in the network gets its own learning rate sampled from a random distribution spanning several orders of magnitude. This comes at practically no computational cost. Perhaps surprisingly, stochastic gradient descent (SGD) with Alrao performs close to SGD with an optimally tuned learning rate, for various architectures and problems. Alrao could save time when testing deep learning models: a range of models could be quickly assessed with Alrao, and the most promising models could then be trained more extensively. This text comes with a PyTorch implementation of the method, which can be plugged on an existing PyTorch model.

1 Introduction

Hyperparameter tuning is a notable source of computational cost with deep learning models [50]. One of the most critical hyperparameters is the learning rate of the gradient descent [42, 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 [14, 21, 28, 40]. Although popular optimizers like Adam [18] come with default hyperparameters, fine-tuning and scheduling of the Adam learning rate is still frequent [5], and we suspect the default setting might be somewhat specific to current problems and architecture sizes. Such hyperparameter tuning takes up a lot of engineering time. These and other issues largely prevent deep learning models from working out-of-the-box on new problems, or on a wide range of problems, without human intervention (AutoML setup, [8]).

We propose *All Learning Rates At Once* (Alrao), an alteration of standard optimization methods for deep learning models. Alrao uses multiple learning rates at the same time in the same network. By sampling one learning rate per feature, Alrao reaches performance close to the performance of the optimal learning rate, without having to try multiple learning rates. Alrao can be used on top of various optimization algorithms; we tested SGD and Adam [18]. Alrao with Adam typically led to strong overfit with good train but poor test performance (see Sec. 4), and our experimental results are obtained with Alrao on top of SGD.

Alrao could be useful when testing architectures: an architecture could first be trained with Alrao to obtain an approximation of the performance it would have with an optimal learning rate. Then it would be possible to select a subset of promising architectures based on Alrao, and search for the best learning rate on those architectures, fine-tuning with any optimizer.

Alrao increases the size of a model on the output layer, but not on the internal layers: this usually adds little computational cost unless most parameters occur on the output layer. This text comes along with a Pytorch implementation usable on a wide set of architectures.

[*]Facebook AI Research, Paris, France
[†]Université Paris Sud, INRIA, equipe TAU, Gif-sur-Yvette, France
[‡]Equal contribution

Related Work. Automatically using the "right" learning rate for each parameter was one motivation behind "adaptive" methods such as RMSProp [43], AdaGrad [6] or Adam [18]. Adam with its default setting is currently considered the default go-to method in many works [49], and we use it as a baseline. However, further global adjustement of the learning rate in Adam is common [26]. Many other heuristics for setting the learning rate have been proposed, e.g., [35]; most start with the idea of approximating a second-order Newton step to define an optimal learning rate [22].

Methods that directly set per-parameter learning rates are equivalent to preconditioning the gradient descent with a diagonal matrix. Asymptotically, an arguably optimal preconditioner is either the Hessian of the loss (Newton method) or the Fisher information matrix [1]. These can be viewed as setting a per-direction learning rate after redefining directions in parameter space. From this viewpoint, Alrao just replaces these preconditioners with a random diagonal matrix whose entries span several orders of magnitude.

Another approach to optimize the learning rate is to perform a gradient descent on the learning rate itself through the whole training procedure (for instance [29]). This can be applied online to avoid backpropagating through multiple training rounds [32]. This idea has a long history, see, e.g., [36] or [30] and the references therein.

The learning rate can also be treated within the framework of architecture search, which can explore both the architecture and learning rate at the same time (e.g., [34]). Existing methods range from reinforcement learning [50, 2] to bandits [25], evolutionary algorithms (e.g., [39, 16, 34]), Bayesian optimization [4] or differentiable architecture search [27]. These powerful methods are resource-intensive and do not allow for finding a good learning rate in a single run.

Motivation. Alrao was inspired by the intuition that not all units in a neural network end up being useful. Hopefully, in a large enough network, a sub-network made of units with a good learning rate could learn well, and hopefully the units with a wrong learning rate will just be ignored. (Units with a too large learning rate may produce large activation values, so this assumes the model has some form of protection against those, such as BatchNorm or sigmoid/tanh activations.)

Several lines of work support the idea that not all units of a network are useful or need to be trained. First, it is possible to *prune* a trained network without reducing the performance too much (e.g., [23, 9, 10, 37]). [24] even show that performance is reasonable if learning only within a very small-dimensional affine subspace of the parameters, *chosen in advance at random* rather than post-selected.

Second, training only some of the weights in a neural network while leaving the others at their initial values performs reasonably well (see experiments in Appendix F). So in Alrao, units with a very small learning rate should not hinder training.

Alrao is consistent with the *lottery ticket hypothesis*, which posits that "large networks that train successfully contain subnetworks that—when trained in isolation—converge in a comparable number of iterations to comparable accuracy" [7]. This subnetwork is the *lottery ticket winner*: the one which had the best initial values. Arguably, given the combinatorial number of subnetworks in a large network, with high probability one of them is able to learn alone, and will make the whole network converge.

Viewing the per-feature learning rates of Alrao as part of the initialization, this hypothesis suggests there might be enough sub-networks whose initialization leads to good convergence.

2 All Learning Rates At Once: Description

Alrao: principle. Alrao starts with a standard optimization method such as SGD, and a range of possible learning rates $(\eta_{\min}, \eta_{\max})$. Instead of using a single learning rate, we sample once and for all one learning rate for each *feature*, randomly sampled log-uniformly in $(\eta_{\min}, \eta_{\max})$. Then these learning rates are used in the usual optimization update:

$$\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \cdot \nabla_{\theta_{l,i}} \ell(\Phi_\theta(x), y) \tag{1}$$

where $\theta_{l,i}$ is the set of parameters used to compute the feature i of layer l from the activations of layer $l-1$ (the *incoming* weights of feature i). Thus we build "slow-learning" and "fast-learning" features, in the hope to get enough features in the "Goldilocks zone".

What constitutes a *feature* depends on the type of layers in the model. For example, in a fully connected layer, each component of a layer is considered as a feature: all incoming weights of the same unit share the same learning rate. On the other hand, in a convolutional layer we consider each convolution filter as constituting a feature: there is one learning rate per filter (or channel), thus keeping translation-invariance over the input image. In LSTMs, we apply the same learning rate to all components in each LSTM unit (thus in the implementation, the vector of learning rates is the same for input gates, for forget gates, etc.).

However, the update (1) cannot be used directly in the last layer. For instance, for regression there may be only one output feature. For classification, each feature in the final classification layer represents a single category, and so using different learning rates for these features would favor some categories during learning. Instead, on the output layer we chose to duplicate the layer using several learning rate values, and use a (Bayesian) model averaging method to obtain the overall network output (Fig. 1).

We set a learning rate *per feature*, rather than per parameter. Otherwise, every feature would have some parameters with large learning rates, and we would expect even a few large incoming weights to be able to derail a feature. So having diverging parameters within a feature is hurtful, while having diverging features in a layer is not necessarily hurtful since the next layer can choose to disregard them. Still, we tested this option; the results are compatible with this intuition (Appendix E).

Definitions and 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: $\frac{1}{N}\sum_i \ell(\Phi_\theta(x_i), y_i)$.

A deep learning model for classification Φ_θ is made of two parts: a *pre-classifier* $\phi_{\theta^{\mathrm{pc}}}$ which computes some quantities fed to a final *classifier layer* $C_{\theta^{\mathrm{c}}}$, namely, $\Phi_\theta(x) = C_{\theta^{\mathrm{cl}}}(\phi_{\theta^{\mathrm{pc}}}(x))$. The classifier layer $C_{\theta^{\mathrm{c}}}$ with K categories is defined by $C_{\theta^{\mathrm{c}}} = \mathrm{softmax} \circ \left(W^T x + b\right)$ with $\theta^{\mathrm{cl}} = (W, b)$, and $\mathrm{softmax}(x_1, ..., x_K)_k = e^{x_k}/(\sum_i e^{x_i})$ .The *pre-classifier* is a computational graph composed of any number of *layers*, and each layer is made of multiple *features*.

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

$$\log\text{-}\mathcal{U}(\eta; \eta_{\min}, \eta_{\max}) = \frac{\mathbb{1}_{\eta_{\min} \le \eta \le \eta_{\max}}}{\eta_{\max} - \eta_{\min}} \times \frac{1}{\eta} \tag{2}$$

Alrao for the pre-classifier: A random learning rate for each feature. In the pre-classifier, for each feature 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] Then the incoming parameters of each feature in the preclassifier are updated in the usual way with this learning rate (Eq. 4).

[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 noise.

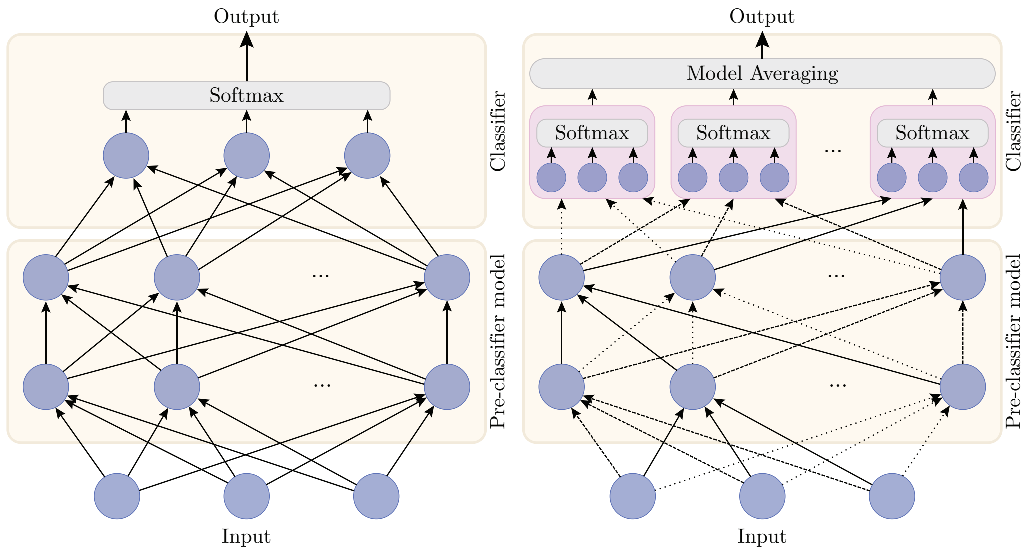


Figure 1: Left: a standard fully connected neural network for a classification task with three classes, made of a pre-classifier and a classifier layer. Right: Alrao version of the same network. The single classifier layer is replaced with a set of parallel copies of the original classifier, averaged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

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

$$C^{\text{Alrao}}_{\theta^{\text{cl}}}(z) := \sum_{j=1}^{N_{\text{cl}}} a_j \, C_{\theta^{\text{cl}}_j}(z), \qquad \Phi^{\text{Alrao}}_{\theta}(x) := C^{\text{Alrao}}_{\theta^{\text{cl}}}(\phi_{\theta^{\text{pc}}}(x)) \tag{3}$$

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

For each classifier $C_{\theta^{\text{cl}}_j}$, we set a learning rate $\log \eta_j = \log \eta_{\min} + \frac{j-1}{N_{\text{cl}}-1} \log(\eta_{\max}/\eta_{\min})$, so that the classifiers' learning rates are log-uniformly spread on the interval $(\eta_{\min}, \eta_{\max})$.

Thus, the original model $\Phi_\theta(x)$ leads to the Alrao model $\Phi^{\text{Alrao}}_{\theta}(x)$. Only the classifier layer is modified, the pre-classifier architecture being unchanged.

The Alrao update. Alg. 1 presents the full Alrao algorithm for use with SGD (other optimizers like Adam are treated similarly). The updates for the pre-classifier, classifier, and model averaging weights are as follows.

- The update rule for the pre-classifier is the usual SGD one, with per-feature learning rates. For each feature i in each layer l, its incoming parameters are updated as:

$$\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \cdot \nabla_{\theta_{l,i}} \ell(\Phi^{\text{Alrao}}_{\theta}(x), y) \tag{4}$$

- The parameters θ^{cl}_j of each classifier clone j on the classifier layer are updated as if this classifier alone was the only output of the model:

$$\theta^{\text{cl}}_j \leftarrow \theta^{\text{cl}}_j - \eta_j \cdot \nabla_{\theta^{\text{cl}}_j} \ell(C_{\theta^{\text{cl}}_j}(\phi_{\theta^{\text{pc}}}(x)), y) \tag{5}$$

Algorithm 1 Alrao-SGD for model $\Phi_\theta = C_{\theta^{\mathrm{cl}}} \circ \phi_{\theta^{\mathrm{pc}}}$ with N_{cl} classifiers and learning rates in $[\eta_{\min}, \eta_{\max}]$

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

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

- To set the weights a_j, several model averaging techniques are available, such as Bayesian Model Averaging [47]. We decided to use the *Switch* model averaging [44], a Bayesian method which is both simple, principled and very responsive to changes in performance of the various models. After each sample or mini-batch, the switch computes a modified posterior distribution (a_j) over the classifiers. This computation is directly taken from [44] and explained in Appendix A. The observed evolution of this posterior during training is commented in Appendix B.

Implementation. We release along with this paper a Pytorch [33] implementation of this method. It can be used on an existing model with little modification. A short tutorial is given in Appendix H. The *features* (sets of weights which will share the same learning rate) need to be defined for each layer type: for now we have done this for linear, convolutional, and LSTMs layers.

3 Experiments

We tested Alrao on various convolutional networks for image classification (CIFAR10), and on LSTMs for text prediction. The baselines are SGD with an optimal learning rate, and Adam with its default setting, arguably the current default method [49].

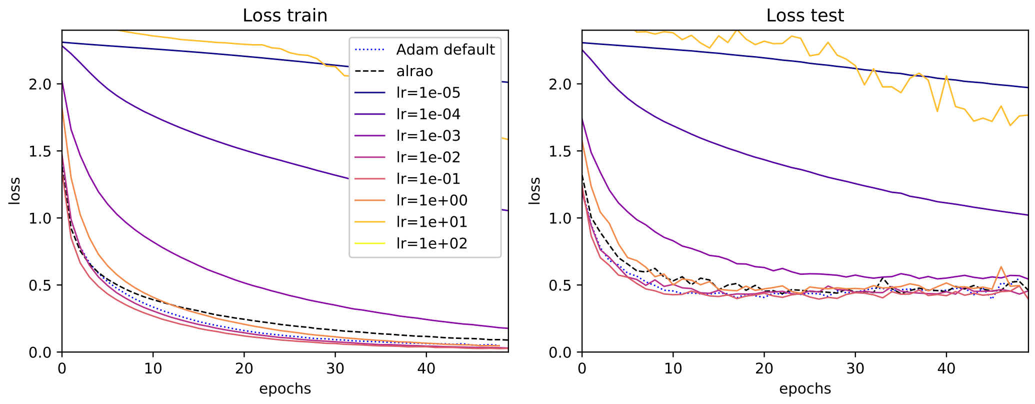
Image classification on CIFAR10. For image classification, we used the CIFAR10 dataset [20]. It is made of 50,000 training and 10,000 testing data; we split the training set into a smaller training set with 40,000 samples, and a validation set with 10,000 samples. For each architecture, training on the smaller training set 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. Training used data augmentation (random cropping and random horizontal flipping). The batch size is always 32. Each setting was run 10 times: the confidence intervals presented are the standard deviation over these runs.

We tested Alrao on three architectures known to perform well on this task: GoogLeNet [41], VGG19 [38] and MobileNet [13]. The exact implementation for each can be found in our code.

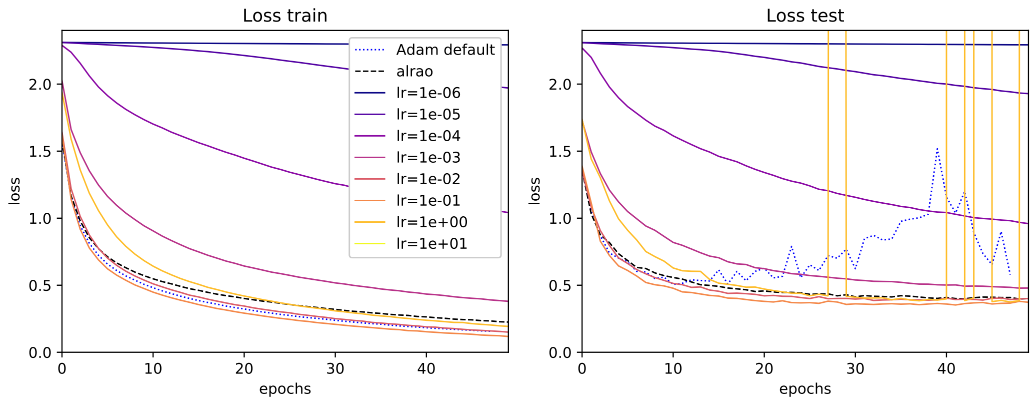
The Alrao learning rates were sampled log-uniformly from $\eta_{\min} = 10^{-5}$ to $\eta_{\max} = 10$. For the output layer we used 10 classifiers with switch model averaging (Appendix A); the learning rates of the output classifiers are deterministic and log-uniformly spread in $[\eta_{\min}, \eta_{\max}]$.

In addition, each model was trained with SGD for every learning rate in the set $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$. The best SGD learning rate is selected on the validation set, then reported in Table 1. We also compare to Adam with its default hyperparameters ($\eta = 10^{-3}, \beta_1 = 0.9, \beta_2 = 0.999$).

The results are presented in Table 1. Learning curves with various SGD learning rates, with Adam, and with Alrao are presented in Fig. 2. Fig. 3 tests the influence of $\eta_{\min}$ and $\eta_{\max}$.



(a) GoogLeNet



(b) MobileNetV2

Figure 2: Learning curves for SGD with various learning rates, Alrao-SGD, and Adam with its default setting, with the GoogLeNet and MobileNetV2 architecture. Left: training loss; right: test loss. While Alrao-SGD uses learning rates from the entire range, its performance is comparable to the optimal learning rate.

Recurrent learning on Penn Treebank. To test Alrao on a different kind of architecture, we used a recurrent neural network for text prediction on the Penn Treebank [31] dataset. The experimental procedure is the same, with $(\eta_{\min}, \eta_{\max}) = (0.001, 100)$ and 6

Table 1: Performance of Alrao-SGD, of SGD with optimal learning rate from $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$, and of Adam with its default setting. Three convolutional models are reported for image classifaction (CIFAR10) and one recurrent model for character prediction (Penn Treebank). For Alrao the learning rates lie in $[\eta_{\min}; \eta_{\max}] = [10^{-5}; 10]$ (CIFAR10) or $[10^{-3}; 10^2]$ (PTB). Each experiment is run 10 times (CIFAR10) or 5 times (PTB); the confidence intervals report the standard deviation over these runs.

Model	SGD with optimal LR			Adam - Default		Alrao-SGD	
	LR	Loss	Acc (%)	Loss	Acc (%)	Loss	Acc (%)
MobileNet	1*e*-1	0.37 ± 0.01	90.2 ± 0.3	1.01 ± 0.95	78 ± 11	0.42 ± 0.02	88.1 ± 0.6
GoogLeNet	1*e*-2	0.45 ± 0.05	89.6 ± 1.0	0.47 ± 0.04	89.8 ± 0.4	0.47 ± 0.03	88.9 ± 0.8
VGG19	1*e*-1	0.42 ± 0.02	89.5 ± 0.2	0.43 ± 0.02	88.9 ± 0.4	0.45 ± 0.03	87.5 ± 0.4
LSTM (PTB)	1	1.566 ± 0.003	66.1 ± 0.1	1.587 ± 0.005	65.6 ± 0.1	1.67 ± 0.01	64.1 ± 0.2

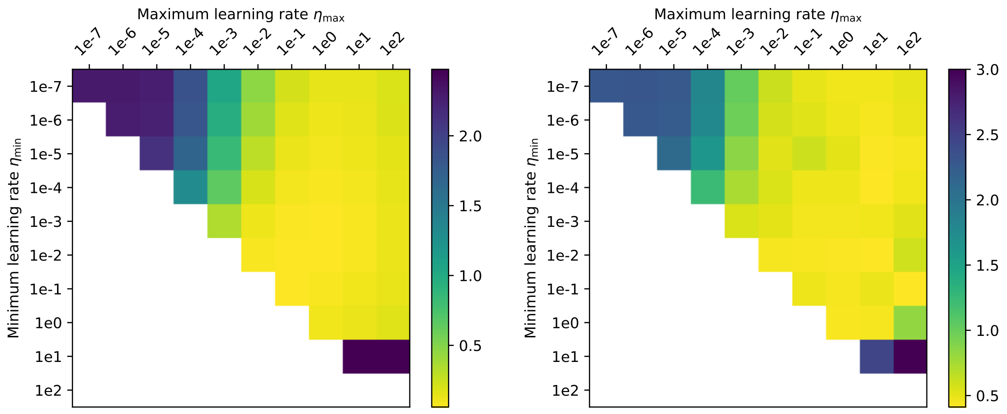


Figure 3: Performance of Alrao with a GoogLeNet model, depending on the interval $(\eta_{\min}, \eta_{\max})$. Left: loss on the train set; right: on the test set. Each point with coordinates $(\eta_{\min}, \eta_{\max})$ above the diagonal represents the loss after 30 epochs for Alrao with this interval. Points (η, η) on the diagonal represent standard SGD with learning rate η after 50 epochs. Standard SGD with $\eta = 10^2$ is left blank to due numerical divergence (NaN). Alrao works as soon as $(\eta_{\min}, \eta_{\max})$ contains at least one suitable learning rate.

output classifiers for Alrao. The results appear in Table 1, where the loss is given in bits per character and the accuracy is the proportion of correct character predictions.

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 more output parameters than character prediction; see Section 4 and Appendix D.

The model is a two-layer LSTM [12] with an embedding size of 100 and with 100 hidden features. 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 [48] with truncation every 70 characters.

Comments. As expected, Alrao performs slightly worse than the best learning rate. Still, even with wide intervals $(\eta_{\min}, \eta_{\max})$, Alrao comes reasonably close to the best learning rate, across all setups; hence Alrao's possible use as a quick assessment method. Although Adam with its default parameters almost matches optimal SGD, this is not always the case, for example with the MobileNet model (Fig.2b). This confirms a known risk of overfit with Adam [49]. In our setup, Alrao seems to be a more stable default method.

Our results, with either SGD, Adam, or SGD-Alrao, are somewhat below the art: in part this is because we train on only 40,000 CIFAR samples, and do not use stepsize schedules.

4 Limitations, further remarks, and future directions

Increased number of parameters for the classification layer. Alrao modifies the output layer of the optimized model. The number of parameters for the classification layer is multiplied by the number of classifier copies used (the number of parameters in the pre-classifier is unchanged). On CIFAR10 (10 classes), the number of parameters increased by less than 5% for the models used. On Penn Treebank, the number of parameters increased by 15% in our setup (working at the character level); working at word level it would have increased threefold (Appendix D).

This is clearly a limitation for models with most parameters in the classifier layer. For output-layer-heavy models, this can be mitigated by handling the copies of the classifiers on distinct computing units: in Alrao these copies work in parallel given the pre-classifier.

Still, models dealing with a very large number of output classes usually rely on other parameterizations than a direct softmax, such as a hierarchical softmax (see references in [15]); Alrao could be used in conjunction with such methods.

Adding two hyperparameters. 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. In our experiments, convergence happens as soon as $(\eta_{\min}; \eta_{\max})$ contains a reasonable learning rate (Fig. 3).

A wide range of values of $(\eta_{\min}; \eta_{\max})$ will contain one good learning rate and achieve close-to-optimal performance (Fig. 3). Typically, we 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.

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 learning rate. For instance, standard SGD fails due to numerical issues for $\eta = 100$ while Alrao with $\eta_{\max} = 100$ works with any $\eta_{\min} \leq 1$ (Fig. 3), and is thus stable to relatively large learning rates. We would still expect numerical issues with very large $\eta_{\max}$, but this has not been observed in our experiments.

Alrao with Adam. Alrao is much less reliable with Adam than with SGD. Surprisingly, this occurs mostly for test performance, which can even diverge, while training curves mostly look good (Appendix C). We have no definitive explanation for this at present. It might be that changing the learning rate in Adam also requires changing the momentum parameters in a correlated way. It might be that Alrao does not work on Adam because Adam is more sensitive to its hyperparameters. The stark train/test discrepancy might also suggest that Alrao-Adam performs well as a pure optimization method but exacerbates the underlying risk of overfit of Adam [49, 17].

Increasing network size. With Alrao, neurons with unsuitable learning rates will not learn: those with a too large learning rate might learn nothing, while those with too small learning rates will learn too slowly to be used. Thus, Alrao may reduce the *effective size* of the network to only a fraction of the actual architecture size, depending on $(\eta_{\min}, \eta_{\max})$.

Our first intuition was that increasing the width of the network was going to be necessary with Alrao, to avoid wasting too many units. In a fully connected network, the number of weights is quadratic in the width, so increasing width (beyond a factor three in our experiments) can be bothersome. Comparisons of Alrao with increased width are reported in

Appendix G. Width is indeed a limiting factor for the models considered, even without Alrao (Appendix G). But to our surprise, Alrao worked well even without width augmentation.

Other optimization algorithms, other hyperparameters, learning rate schedulers... Using a learning rate schedule instead of a fixed learning rate is often effective [3]. We did not use learning rate schedulers here; this may partially explain why the results in Table 1 are worse than the state-of-the-art. Nothing prevents using such a scheduler within Alrao, e.g., by dividing all Alrao learning rates by a time-dependent constant; we did not experiment with this yet.

The idea behind Alrao could be used on other hyperparameters as well, such as momentum. However, if more hyperparameters are initialized randomly for each feature, the fraction of features having all their hyperparameters in the "Goldilocks zone" will quickly decrease.

5 Conclusion

Applying stochastic gradient descent with random learning rates for different features is surprisingly resilient in our experiments, and provides performance close enough to SGD with an optimal learning rate, as soon as the range of random learning rates contains a suitable one. This could save time when testing deep learning models, opening the door to more out-of-the-box uses of deep learning.

Acknowledgments

We would like to thank Corentin Tallec for his technical help, and his many remarks and advice. We thank Olivier Teytaud for pointing useful references.

References

[1] S.-i. Amari. Natural gradient works efficiently in learning. *Neural Comput.*, 10:251–276, February 1998.

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[3] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[4] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[5] M. Denkowski and G. Neubig. Stronger baselines for trustable results in neural machine translation. *arXiv preprint arXiv:1706.09733*, 2017.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[7] J. Frankle and M. Carbin. The Lottery Ticket Hypothesis: Finding Small, Trainable Neural Networks. *arXiv preprint arXiv:1704.04861*, mar 2018.

[8] I. Guyon, I. Chaabane, H. J. Escalante, S. Escalera, D. Jajetic, J. R. Lloyd, N. Macià, B. Ray, L. Romaszko, M. Sebag, et al. A brief review of the ChaLearn AutoML challenge: any-time any-dataset learning without human intervention. In *Workshop on Automatic Machine Learning*, pages 21–30, 2016.

[9] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149*, 2015.

[10] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both Weights and Connections for Efficient Neural Networks. In *Advances in Neural Information Processing Systems*, 2015.

[11] M. Herbster and M. K. Warmuth. Tracking the best expert. *Machine learning*, 32(2):151–178, 1998.

[12] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[14] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[15] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[16] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015.

[17] N. S. Keskar and R. Socher. Improving generalization performance by switching from Adam to SGD. *arXiv preprint arXiv:1712.07628*, 2017.

[18] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, 2015.

[19] W. Koolen and S. De Rooij. Combining expert advice efficiently. *arXiv preprint arXiv:0802.2015*, 2008.

[20] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009.

[21] K. Kurita. Learning Rate Tuning in Deep Learning: A Practical Guide — Machine Learning Explained, 2018.

[22] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998.

[23] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[24] C. Li, H. Farkhoor, R. Liu, and J. Yosinski. Measuring the Intrinsic Dimension of Objective Landscapes. *arXiv preprint arXiv:1804.08838*, apr 2018.

[25] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[26] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

[27] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[28] D. Mack. How to pick the best learning rate for your machine learning project, 2016.

[29] D. Maclaurin, D. Duvenaud, and R. Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[30] A. R. Mahmood, R. S. Sutton, T. Degris, and P. M. Pilarski. Tuning-free step-size adaptation. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 2121–2124. IEEE, 2012.

[31] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[32] P.-Y. Massé and Y. Ollivier. Speed learning on the fly. *arXiv preprint arXiv:1511.02540*, 2015.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[34] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[35] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

[36] N. N. Schraudolph. Local gain adaptation in stochastic gradient descent. 1999.

[37] A. See, M.-T. Luong, and C. D. Manning. Compression of Neural Machine Translation Models via Pruning. *arXiv preprint arXiv:1606.09274*, 2016.

[38] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[39] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[40] P. Surmenok. Estimating an Optimal Learning Rate For a Deep Neural Network, 2017.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[42] S. Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic Press, 2015.

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

[44] T. Van Erven, P. Grünwald, and S. De Rooij. 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):361–417, 2012.

[45] T. Van Erven, S. D. Rooij, and P. Grünwald. Catching up faster in Bayesian model selection and model averaging. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 417–424. Curran Associates, Inc., 2008.

[46] P. A. Volf and F. M. Willems. Switching between two universal source coding algorithms. In *Data Compression Conference, 1998. DCC'98. Proceedings*, pages 491–500. IEEE, 1998.

[47] L. Wasserman. Bayesian Model Selection and Model Averaging. *Journal of Mathematical Psychology*, 44, 2000.

[48] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[49] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pages 4148–4158, 2017.

[50] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A Model Averaging with the Switch

As explained is Section 2, we use a model averaging method on the classifiers of the output layer. We could have used the Bayesian Model Averaging method [47]. But one of its main weaknesses is the *catch-up phenomenon* [44]: plain Bayesian posteriors are slow to react when the relative performance of models changes over time. Typically, for instance, some larger-dimensional models need more training data to reach good performance: at the time they become better than lower-dimensional models for predicting current data, their Bayesian posterior is so bad that they are not used right away (their posterior needs to "catch up" on their bad initial performance). This leads to very conservative model averaging methods.

The solution from [44] against the catch-up phenomenon is to *switch* between models. It is based on previous methods for prediction with expert advice (see for instance [11, 46] and the references in [19, 44]), and is well rooted in information theory. The switch method maintains a Bayesian posterior distribution, not over the set of models, but over the set of *switching strategies* between models. Intuitively, the model selected can be adapted online to the number of samples seen.

We now give a quick overview of the switch method from [44]: this is how the model averaging weights a_j are chosen in Alrao.

Assume that we have a set of prediction strategies $\mathcal{M} = \{p^j, j \in \mathcal{I}\}$. We define the set of *switch sequences*, $\mathbb{S} = \{((t_1, j_1), ..., (t_L, j_L)), 1 = t_1 < t_2 < ... < t_L \ , \ j \in \mathcal{I}\}$. Let $s = ((t_1, j_1), ..., (t_L, j_L))$ be a switch sequence. The associated prediction strategy $p_s(y_{1:n}|x_{1:n})$ uses model p^{j_i} on the time interval $[t_i; t_{i+1})$, namely

$$p_s(y_{1:i+1}|x_{1:i+1}, y_{1:i}) = p^{K_i}(y_{i+1}|x_{1:i+1}, y_{1:i}) \tag{6}$$

where K_i is such that $K_i = j_l$ for $t_l \leq i < t_{l+1}$. We fix a prior distribution π over switching sequences. In this work, $\mathcal{I} = \{1, ..., N_C\}$ the prior is, for a switch sequence $s = ((t_1, j_1), ..., (t_L, j_L))$:

$$\pi(s) = \pi_L(L)\pi_K(j_1)\prod_{i=2}^{L}\pi_T(t_i|t_i > t_{i-1})\pi_K(j_i) \tag{7}$$

with $\pi_L(L) = \frac{\theta^L}{1-\theta}$ a geometric distribution over the switch sequences lengths, $\pi_K(j) = \frac{1}{N_C}$ the uniform distribution over the models (here the classifiers) and $\pi_T(t) = \frac{1}{t(t+1)}$.

This defines a Bayesian mixture distribution:

$$p_{sw}(y_{1:T}|x_{1:T}) = \sum_{s \in \mathbb{S}} \pi(s) p_s(y_{1:T}|x_{1:T}) \tag{8}$$

Then, the model averaging weight a_j for the classifier j after seeing T samples is the posterior of the switch distribution: $\pi(K_{T+1} = j|y_{1:T}, x_{1:T})$.

$$a_j = p_{sw}(K_{T+1} = j|y_{1:T}, x_{1:T}) = \frac{p_{sw}(y_{1:T}, K_{T+1} = j|x_{1:T})}{p_{sw}(y_{1:T}|x_{1:T})} \tag{9}$$

These weights can be computed online exactly in a quick and simple way [44], thanks to dynamic programming methods from hidden Markov models.

The implementation of the switch used in Alrao exactly follows the pseudo-code from [45], with hyperparameter $\theta = 0.999$ (allowing for many switches a priori). It can be found in the accompanying online code.

B Evolution of the Posterior

The evolution of the model averaging weights can be observed during training. In Figure 4, we can see their evolution during the training of the GoogLeNet model with Alrao, 10 classifiers, with $\eta_{\min} = 10^{-5}$ and $\eta_{\max} = 10^{1}$.

We can make several observations. First, after only a few gradient descent steps, the model averaging weights corresponding to the three classifiers with the largest learning rates go to zero. This means that their parameters are moving too fast, and their loss is getting very large.

Next, for a short time, a classifier with a moderately large learning rate gets the largest posterior weight, presumably because it is the first to learn a useful model.

Finally, after the model has seen approximately 4,000 samples, a classifier with a slightly smaller learning rate is assigned a posterior weight a_j close to 1, while all the others go to 0. This means that after a number of gradient steps, the model averaging method acts like a model selection method.

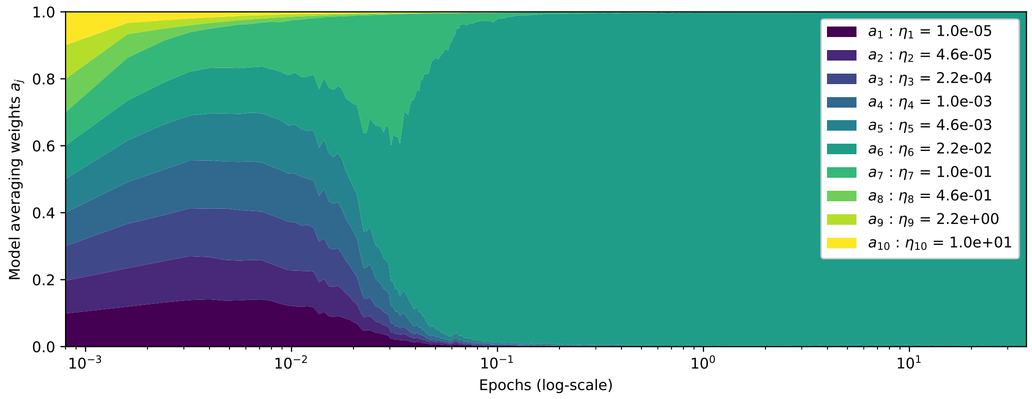


Figure 4: Model averaging weights during training. During the training of the GoogLeNet model with Alrao, 10 classifiers, with $\eta_{\min} = 10^{-5}$ and $\eta_{\max} = 10^{1}$, we represent the evolution of the model averaging weights a_j, depending on the corresponding classifier's learning rate.

C Alrao-Adam

In Figure 5, we report our experiments with Alrao-Adam. As explained in Section 4, Alrao is much less reliable with Adam than with SGD.

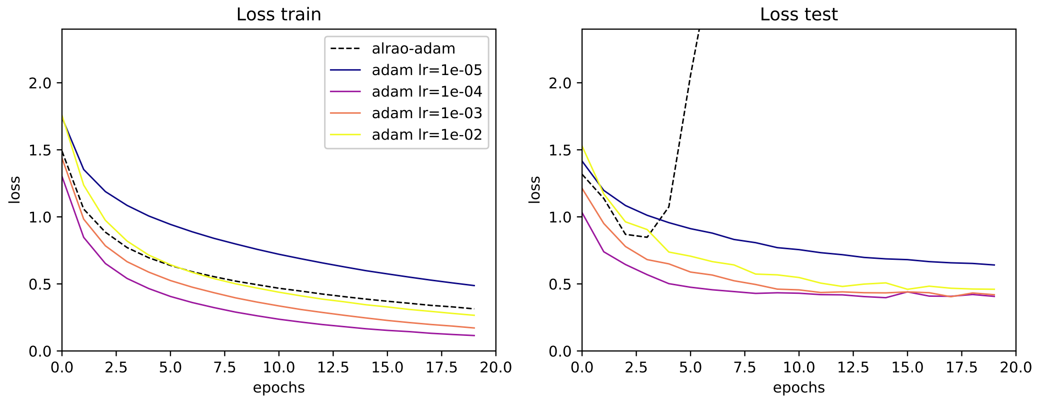
This is especially true for the test performance, which can even diverge while training performance remains either good or acceptable (Fig. 5). Thus Alrao-Adam seems to send the model into atypical regions of the search space.

D Number of Parameters

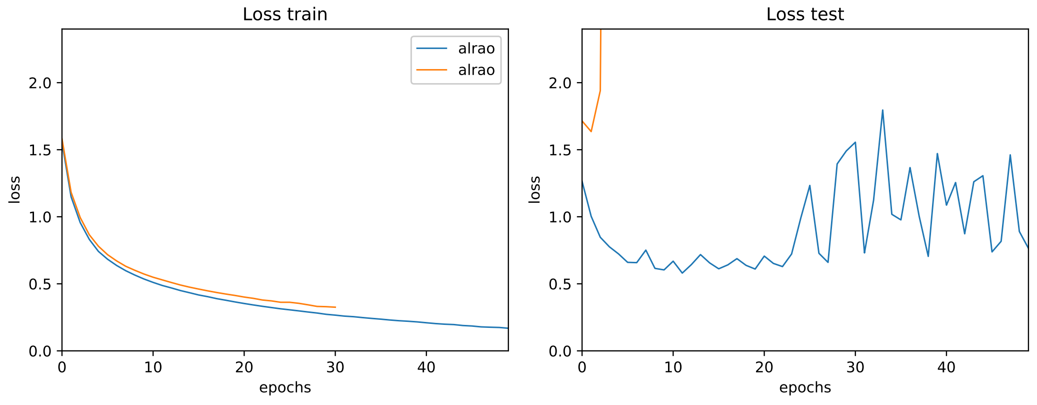
As explained in Section 4, Alrao increases the number of parameters of a model, due to output layer copies. The additional number of parameters is approximately equal to $(N_{\text{cl}} - 1) \times K \times d$ where N_{cl} is the number of classifier copies used in Alrao, d is the dimension of the output of the pre-classifier, and K is the number of classes in the classification task (assuming a standard softmax output; classification with many classes often uses other kinds of output parameterization instead).

The number of parameters for the models used, with and without Alrao, are in Table 2. We used 10 classifiers in Alrao for convolutional neural networks, and 6 classifiers for LSTMs. Using Alrao for classification tasks with many classes, such as word prediction (10,000 classes on PTB), increases the number of parameters noticeably.

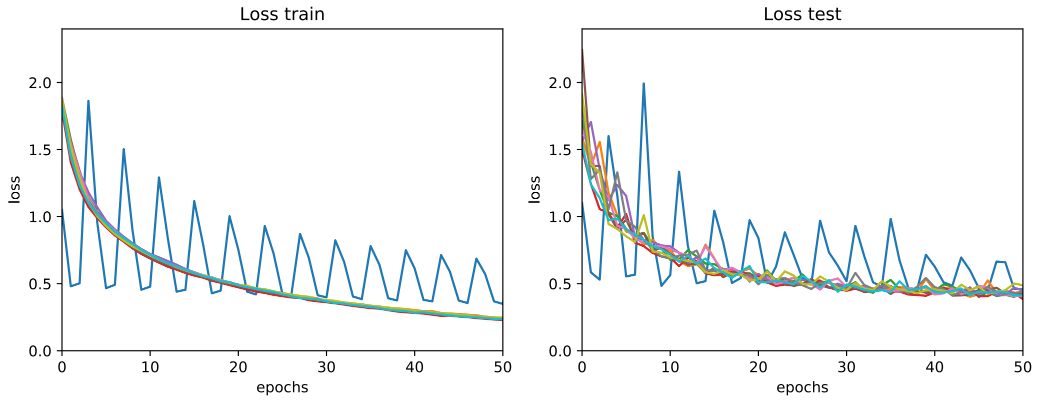
For those model with significant parameter increase, the various classifier copies may be done on parallel GPUs.



(a) Alrao-Adam on GoogLeNet: Alrao-Adam compared with standard Adam with various learning rates. Alrao uses 10 classifiers and learning rates in the interval $[10^{-6}, 1]$. Each plot is averaged on 10 experiments. We observe that optimization with Alrao-Adam is efficient, since train loss is comparable to the usual Adam methods. But the model starkly overfits, as the test loss diverges.



(b) Alrao-Adam on MobileNet: Alrao-Adam with two different learning rate intervals, with 10 classifiers. Each plot is averaged on 10 experiments. Exactly as with GoogLeNet model, optimization itself is efficient (for both intervals). For the interval with the smallest $\eta_{\max}$, the test loss does not converge and is very unstable. For the interval with the largest $\eta_{\max}$, the test loss diverges.



(c) Alrao-Adam on VGG19: Alrao-Adam on the interval $[10^{-6}, 1]$, with 10 classifiers. The 10 plots are 10 runs of the same experiments. While 9 of them do converge and generalize, the last one exhibits wide oscillations, both in train and test.

Figure 5: Alrao-Adam: Experiments on the VGG19, GoogLeNet and MobileNet networks.

Table 2: Comparison between the number of parameters in models used without and with Alrao. LSTM (C) is a simple LSTM cell used for character prediction while LSTM (W) is the same cell used for word prediction.

Model	Number of parameters	
	Without Alao	With Alrao
GoogLeNet	6.166M	6.258M
VGG	20.041M	20.087M
MobileNet	2.297M	2.412M
LSTM (C)	0.172M	0.197M
LSTM (W)	2.171M	7.221M

E Other Ways of Sampling the Learning Rates

In Alrao we sample a learning rate for each feature. Intuitively, each feature (or neuron) is a computation unit of its own, using a number of inputs from the previous layer. If we assume that there is a "right" learning rate for learning new features based on information from the previous layer, then we should try a learning rate per feature; some features will be useless, while others will be used further down in the network.

An obvious variant is to set a random learning rate per weight, instead of for all incoming weights of a given feature. However, this runs the risk that *every* feature in a layer will have a few incoming weights with a large rate, so intuitively every feature is at risk of diverging. This is why we favored per-feature rates.

Still, we tested sampling a learning rate for each weight in the pre-classifier (while keeping the same Alrao method for the classifier layer).

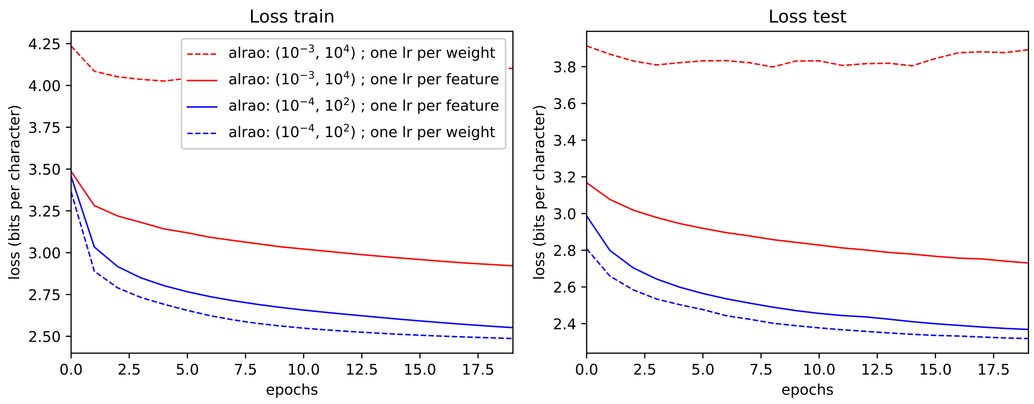


Figure 6: Loss for various intervals $(\eta_{\min}, \eta_{\max})$, as a function of the sampling method for the learning rates, per feature or per weight. The model is a two-layer LSTM trained for 20 epochs only, for character prediction. Each curves represents 10 runs. (Losses are much higher than the results reported in Table 1 because the full training for Table 1 takes approximately 300 epochs.)

In our experiments on LSTMs, per-weight learning rates sometimes perform well but are less stable and more sensitive to the interval $(\eta_{\min}, \eta_{\max})$: for some intervals $(\eta_{\min}, \eta_{\max})$ with very large $\eta_{\max}$, results with per-weight learning rates are a lot worse than with per-feature learning rates. This is consistent with the intuition above.

F Learning a Fraction of the Features

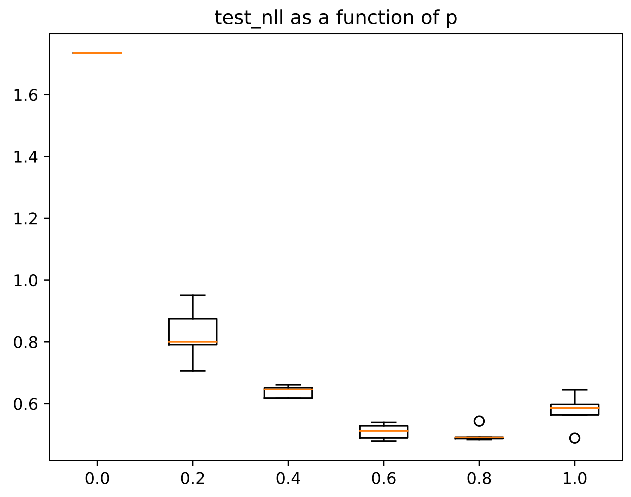


Figure 7: Loss of a model where only a random fraction p of the features are trained, and the others left at their initial value, as a function of p. The architecture is GoogLeNet, trained on CIFAR10.

As explained in the introduction, several works support the idea that not all units are useful when learning a deep learning model. Additional results supporting this hypothesis are presented in Figure 7. We trained a GoogLeNet architecture on CIFAR10 with standard SGD with learning rate η_0, but learned only a random fraction p of the features (chosen at startup), and kept the others at their initial value. This is equivalent to sampling each learning rate η from the probability distribution $P(\eta = \eta_0) = p$ and $P(\eta = 0) = 1 - p$.

We observe that even with a fraction of the weights not being learned, the model's performance is close to its performance when fully trained.

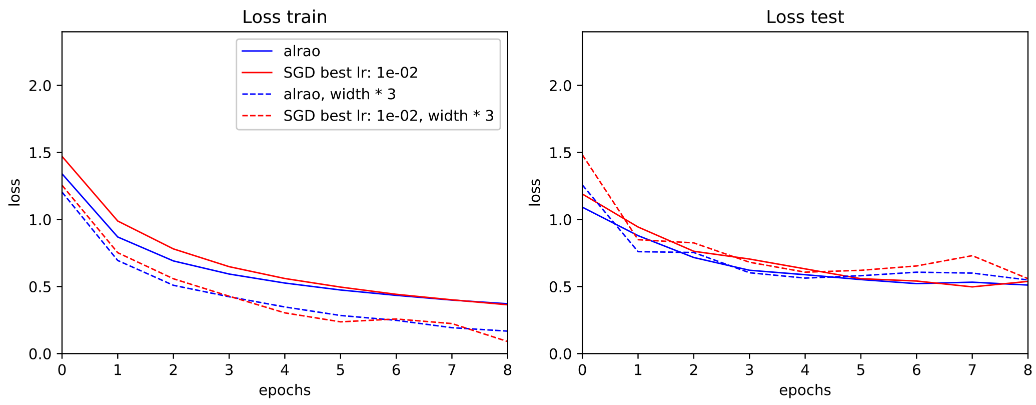
When training a model with Alrao, many features might not learn at all, due to too small learning rates. But Alrao is still able to reach good results. This could be explained by the resilience of neural networks to partial training.

G Increasing Network Size

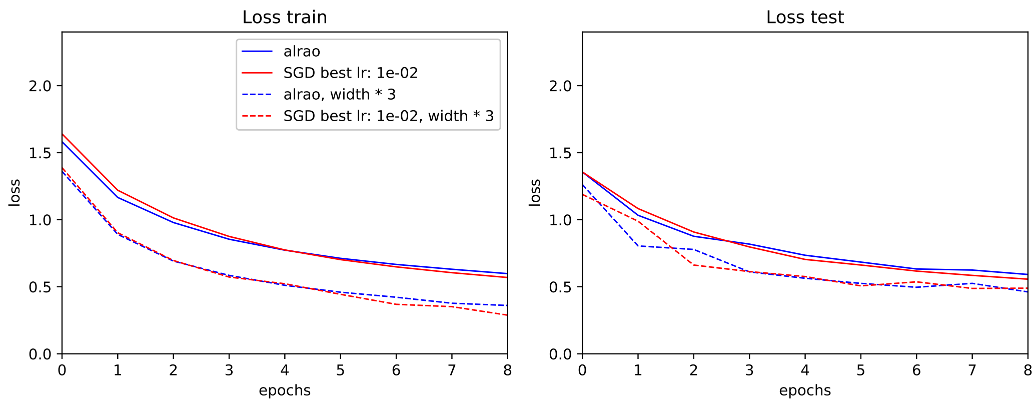
As explained in Section 4, learning with Alrao reduces the *effective size* of the network to only a fraction of the actual architecture size, depending on $(\eta_{\min}, \eta_{\max})$. We first tought that increasing the width of each layer was going to be necessary in order to use Alrao. However, our experiments show that this is not necessary.

Alrao and SGD experiments with increased width are reported in Figure 8. As expected, Alrao with increased width has better performance, since the *effective size* increases. However, increasing the width also improves performance of standard SGD, by roughly the same amount.

Thus, width is still a limiting factor both for GoogLeNet and MobileNet. This shows that Alrao can perform well even when network size is a limiting factor; this runs contrary to our initial intuition that Alrao would require very large networks in order to have enough features with suitable learning rates.



(a) GoogLeNet



(b) MobileNet

Figure 8: Increasing network width. We compare the performance of the GoogLeNet and MobileNet models, to the same models with 3 times as many units in each layer, both for standard SGD and for Alrao.

H Tutorial

In this section, we briefly show how Alrao can be used in practice on an already implemented method in Pytorch. The code will be available soon.

The first step is to build the preclassifier. Here, we use the VGG19 architecture. The model is built without a classifier. Nothing else is required for Alrao at this step.

```
class VGG(nn.Module):
    def __init__(self, cfg):
        super(VGG, self).__init__()
        self.features = self._make_layers(cfg)
        # The dimension of the preclassier's output need to be specified.
        self.linearinputdim = 512

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        # The model do not contain a classifier layer.
        return out

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                           nn.BatchNorm2d(x),
                           nn.ReLU(inplace=True)]
                in_channels = x
        layers += [nn.AvgPool2d(kernel_size=1, stride=1)]
        return nn.Sequential(*layers)

preclassifier = VGG([64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', \
        512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'])
```

Then, we can build the Alrao-model with this preclassifier, sample the learning rates for the model, and define the Alrao optimizer

```
# We define the interval in which the learning rates are sampled
minlr = 10 ** (-5)
maxlr = 10 ** 1

# nb_classifiers is the number of classifiers averaged by Alrao.
nb_classifiers = 10
nb_categories = 10

net = AlraoModel(preclassifier, nb_categories, preclassifier.linearinputdim, nb_classifiers)

# We spread the classifiers learning rates log-uniformly on the interval.
classifiers_lr = [np.exp(np.log(minlr) + \
    k /(nb_classifiers-1) * (np.log(maxlr) - np.log(minlr)) \
    ) for k in range(nb_classifiers)]

# We define the sampler for the preclassifier's features.
lr_sampler = lr_sampler_generic(minlr, maxlr)
lr_preclassifier = generator_randomlr_neurons(net.preclassifier, lr_sampler)

# We define the optimizer
optimizer = SGDAlrao(net.parameters_preclassifier(),
                     lr_preclassifier,
                     net.classifiers_parameters_list(),
                     classifiers_lr)
```

Finally, we can train the model. The only differences here with the usual training procedure is that each classifier needs to be updated as if it was alone, and that we need to update the model averaging weights, here the switch weights.

```
def train(epoch):
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        # We update the model averaging weights in the optimizer
        optimizer.update_posterior(net.posterior())
        optimizer.zero_grad()

        # Forward pass of the Alrao model
        outputs = net(inputs)
        loss = nn.NLLLoss(outputs, targets)

        # We compute the gradient of all the model's weights
        loss.backward()

        # We reset all the classifiers gradients, and re-compute them with
        # as if their were the only output of the network.
        optimizer.classifiers_zero_grad()
        newx = net.last_x.detach()
        for classifier in net.classifiers():
            loss_classifier = criterion(classifier(newx), targets)
            loss_classifier.backward()

        # Then, we can run an update step of the gradient descent.
        optimizer.step()
        # Finally, we update the model averaging weights
        net.update_switch(targets, catch_up=False)
```