ASLR: an Adaptive Scheduler for Learning Rate

Alireza Khodamoradi
UC San Diego
alirezak@eng.ucsd.edu

Kristof Denolf
Xilinx
kristof@xilinx.com

Kees Vissers
Xilinx
keesv@xilinx.com

Ryan C. Kastner
UC San Diego
kastner@ucsd.edu

***Abstract*—Training a neural network is a complicated and time-consuming task that involves adjusting and testing different combinations of hyperparameters. One of the essential hyperparameters is the learning rate, which balances the magnitude of changes at each training step. We introduce an Adaptive Scheduler for Learning Rate (ASLR) that significantly lowers the tuning effort since it only has a single hyperparameter. ASLR produces competitive results compared to the state-of-the-art for both hand-optimized learning rate schedulers and line search methods while requiring significantly less tuning effort. Our algorithm's computational cost is trivial and can be used to train various network topologies included quantized networks.**

I. Introduction

Training a neural network is a time-consuming process that often requires a great deal of optimization of the hyperparameters to achieve a high-quality result. For example, in supervised learning, one has to select an initialization method to start the training, a cost function and an optimizer to perform the training, and a budget of epochs or a target accuracy to stop the training. The training process also involves other choices, such as input normalization, pruning methods, Etc.

Each method or function selected for training comes with a set of hyperparameters that must be tuned. For example, Stochastic Gradient Descent (SGD), which forms the core of many training algorithms, is defined as:

$$\theta_{i+1} = \theta_i - \lambda_i g(\theta_i) \tag{1}$$

SGD iteratively updates the network parameters θ (e.g., weights and biases) by multiplying the *learning rate* λ by the derivative of the cost function $g(\theta)$ and subtracting it from the parameters. The training script calculates the cost function using a subset of the training set. The size of this subset is called *batch size*. Throughout this work, we use $\nabla F(\theta)$ to refer to the gradient of the cost function in Batch Gradient Descent and $g(\theta)$ to refer to the gradient of the cost function in SGD.

Unfortunately, there are no concrete rules to select the exact values for hyperparameters. Moreover, their optimum values[1] heavily depend on the application, the network topology, and choices made for other training parameters. For example, applying quantization to the network parameters requires re-adjusting both hyperparameters (learning rate and batch size) in Equation 1 [1].

[1]It cannot be proved that hyperparameter values are optimum. Therefore it is loosely used to refer to their acceptable values.

Ideally, users can fallback on existing hyperparameters that were meticulously tweaked by experts. However, if this fails to achieve the required results, the user resorts to guessing the initial hyperparameters and proceeds to fine-tune the parameters [2], [3]. This tuning process is often a time-consuming task whose outcome depends on the initial guess, user experience, and a bit of luck.

The learning rate is one of the essential hyperparameters in a training process [4]–[6]. In this work, we aim to reduce the complexity of tuning the learning rate. We introduce an Adaptive Scheduler for Learning Rate (ASLR) that automatically adjusts the learning rate throughout the training process. Our scheduler is particularly useful for training a network with no provided learning rate since it has only one hyperparameter to tune. Our adaptive learning rate scheduler achieves competitive results compared to existing state-of-the-art and (manually) fine-tuned schedulers with multiple user-defined parameters.

The primary contributions are as follows.

- We introduce a novel adaptive learning rate scheduler with a single user-defined parameter and low tuning complexity. This algorithm can achieve competitive results compared to hand-tuned schedulers and line search methods, and its computational cost is trivial.
- We release our code as open-sourced to enhance accessibility and aid in future comparisons of our work.[2]

The remainder of this paper is organized as follows. Section II provides the necessary background material related to the complexity of learning rate tuning. Common trends and techniques for learning rate adjustment are reviewed in Section III. Our proposed algorithm is explained in Section IV. Experiment results are provided in Section V and conclusions are provided in Section VI.

II. Complexity of Learning Rate Tuning

Learning rate is perhaps the most important hyperparameter to tune [7], and in general, it is not possible to calculate the best learning rate a priori [8]. In the following, we provide a brief review of why learning rate tuning is complex and vital.

a) Complexity of loss surface: Gradient descent algorithm iteratively updates the network parameters by using the first derivative of a cost function. This process provides a direction and a value for changing each trainable parameter to minimize the cost function. However, the first derivative provides a rough estimate of the underlying curvature. To

[2]github.com/Xilinx/AdaptiveSchedulers

improve this process, one can use the learning rate to control the magnitude of the change. A large learning rate causes more significant changes to the parameters, while a small learning rate results in smaller changes at each step of the training process.

Figure 1 shows a simple example of a loss function and its underlying surface (i.e., loss surface). Function F has only one parameter θ, and its underlying curvature has one dimension. At step 1, the red arrow indicates the direction of change for θ to decrease the value of F. The learning rate controls the amount of change in that direction. In this example, if the learning rate is set too small, the search process will get stuck around the local minima (θ_L), and the optimal minima (θ_O) will not be obtained. A large learning rate will result in the search moving too far in the wrong direction (away from the optimal result θ_O), making the search process longer and possibly leaving it to diverge.

The complexity of a loss surface calculated for a network directly correlates with the number of trainable parameters in that network. In a real-world network, the curvature of a loss surface can depend on tens of millions of parameters [9]–[12]. Therefore calculating or estimating a "good" learning rate can be a challenging and expensive-to-compute task [13].

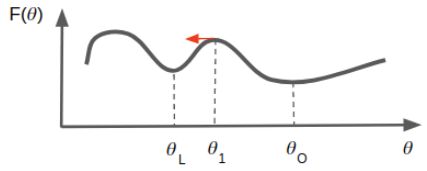

Fig. 1. A sketch of a loss surface with only one parameter. At $\theta = \theta_1$ the red arrow shows the direction of change in θ for descending F. The learning rate controls the size of the change. A small learning rate holds the optimization process around θ_L. A large learning rate results in a value farther from θ_O.

b) Dependency on other training parameters: Equation 1 is typically augmented in an attempt to improve the training results. A common technique used for enhancing SGD is *momentum*, which regularizes the changes at each step based on the variance of the cost function:

$$\begin{aligned} \theta_{i+1} &= \theta_i - \lambda_i v_i \\ v_i &= \beta v_{i-1} + (1-\beta) g(\theta_i) \end{aligned} \tag{2}$$

If used correctly, adding momentum improves the training process. At the same time, it adds additional hyperparameters that must be tuned, e.g., momentum has one additional tunable parameter, β. Other techniques to improve training also introduce new hyperparameters, e.g., weight decay [14]. In many cases, these hyperparameters can affect each other. For example, the learning rate loosely affects the momentum [15] and is strongly related to the batch size [16].

Training a neural network typically requires the tuning of tens of hyperparameters. A large number of hyperparameters makes the tuning process more challenging. Tuning these hyperparameters is a NP-complete problem [13], [17]. For example, hyperparameters available for training the most popular networks result from many trial and error attempts made by many contributors and are considered "finely-tuned". However, for any of those pairs (set of hyperparameters, network), it cannot be proved that the set is optimal for training its network. It is possible that another set of hyperparameters exists that can improve the training results for the network.

One way to relax this complexity is to reduce the number of hyperparameters. This reduction should not reduce the network's performance. In this work, we introduce a learning rate scheduler with a single user-defined parameter and demonstrate our proposed technique's performance by comparing its results with state-of-the-art techniques with finely-tuned parameters.

III. Common Practices for Learning Rate Tuning

In the following, we review four main trends in learning rate tuning, including their complexity, benefits, and disadvantages. Our work is inspired by these methods.

Before starting our review, we should clarify that in each method, only a number of user-defined parameters require careful tuning. These parameters are commonly referred to as *hyperparameters*. Other parameters that require trivial or no tuning are frequently referred to as *default parameters*.

A. Second Order Information

If the loss function $F(\theta)$ is infinitely differentiable at θ, the result of a small change in its input can be calculated using a Taylor decomposition:

$$F(\theta+\delta\theta) - F(\theta) = \frac{\delta\theta}{1!}F'(\theta) + \frac{(\delta\theta)^2}{2!}F''(\theta) + R_2(\theta) \tag{3}$$

Here, $R_2(\theta)$ is the Taylor remainder of order two. A good estimate for learning rate can be calculated by assuming $R_2(\theta) \approx 0$, taking a derivative with regards to $\delta\theta$ from both sides, and setting $\partial(F(\theta+\delta\theta) - F(\theta))/\partial\delta\theta$ to zero:

$$0 = F'(\theta) + \delta\theta F''(\theta) \tag{4}$$

Solving Equation 4 for $\delta\theta$, results in $-F'(\theta)/F''(\theta)$. Rewriting it in a more familiar form yields: $\delta\theta = -\nabla F(\theta)/H$. By comparing this result with the batch gradient descent equation $\theta_{i+1} = \theta_i - \lambda\nabla F(\theta)$, it can be concluded that an optimum learning rate is equal to the inverse of Hessian matrix of F.

Although that $\lambda = H^{-1}$ can provide a good approximation for learning rate[3], calculating the inverse of a large Hessian matrix is expensive. Moreover, using second order information in training increases the sensitivity to sharp minima. We discuss these drawbacks in more detail in the following.

[3]We assumed $R_2(\theta) \approx 0$.

a) Complexity: Calculating the inverse of the Hessian matrix has $O(n^3)$ complexity[4] where n is the number of trainable parameters in the network. Calculating this for modern networks with tens of millions of trainable parameters is computationally infeasible. An approximation of the Hessian matrix can be calculated by estimating its largest eigenvalue and the corresponding eigenvector using the power iteration method. However, the cost is still $10\times$ greater than a single calculation for gradient [5]. To improve this approximation, some methods use a layer-wise approximation. While a layer-wise approach can improve the training results, it can also increase the number of user-defined parameters. For example, the method proposed by [18] has 13 user-defined parameters.

The complexity of using second order information for estimating the learning rate can be relaxed to $O(n^{1.5})$ [6]. Instead of calculating the Hessian matrix, they introduce a notion of distance, $G(\theta)$.

By solving $G(\theta)$ using Kullback–Leibler divergence, they achieved a Fisher matrix for $G(\theta)$ that can be factorized into a Kronecker product of two smaller matrices. Then they create a model based on the approximated distance. By comparing this model with training results, they determine a trust region (a norm ball) to control and adjust the learning rate at each step.

b) Sensitivity: Techniques that utilize second order information are generally sensitive to sharp minima [19], [20]. Using second order information may also lead to a reduction in the generalization of the network accuracy. It is unclear why deep neural networks generalize well [21], but one common belief is that SGD finds wide minima, which in turn tends to generalize better [20], [22]. Thus, it may be beneficial to avoid such sharp minima, and utilizing second order information makes that less likely.

B. Adaptive Optimization Methods

A popular approach for regularizing the learning rate throughout the training is extracting useful information from previous steps [4], [23]–[25]. This can be done by using averaging methods and estimating the first or second moments (or both) of the gradient. Exponential Moving Average (EMA) is a commonly used technique employed in these methods:

$$\mathit{EMA}_i(g) = \beta \mathit{EMA}_{i-1}(g) + (1-\beta) g(\theta_i) \tag{5}$$

This equation calculates a biased average. Dividing the result by $(1-\beta^i)$ can correct the bias.

Using a moving average results in smaller values when the input (g in Equation 5) has a large variance. For example, SGD with momentum (SGDM) uses a single *EMA* to dampen the learning rate when the variance is too high (Equation 2).

ADAM [4] uses two *EMA*s:

$$\theta_{i+1} = \theta_i - \lambda \frac{\widehat{\mathit{EMA}}_i(g)}{\sqrt{\widehat{\mathit{EMA}}_i(g^2)} + \epsilon} \tag{6}$$

[4]By using optimized CW-like algorithms, this complexity can be reduced to $O(n^{2.373})$

where $\widehat{\mathit{EMA}}$ is bias-corrected *EMA*. Moving averages can calculate an expectation for their input: $\widehat{\mathit{EMA}} \approx E[g]$. And the first and second moments are related as: $E[g^2] = E^2[g] + \text{var}(g)$. Therefore, the fractional portion in Equation 6 has an inverse correlation with var(g) and the effect of the learning rate (λ) is regularized based on the variance of g.

With an increase in user-defined parameters, adaptive optimization methods can provide a fast decay in cost function at the beginning of the training. However, in some cases, they produce a poor generalization [23]. In the following, we review this drawback for these methods.

a) Reducing variance: There are two sources for variance when calculating SGD ($g(\theta)$). One is due to the underlying pathological curvature of the loss function. The other is related to the sampling of mini-batches that do not fully represent the entire data set. The variance plays an essential role in the optimization process, as we describe in the following.

Referring back to Figure 1, the derivative of $F(\theta)$ provides the direction of search. If the learning rate is not small enough, it is unlikely to reach θ_L; the search process will cause θ to move back and forth near θ_L. This increases the variance in $\nabla F(\theta)$. Using an adaptive optimization method, an increase in variance can reduce the effect of the learning rate. E.g., in Equation 2, an increase in variance results in a smaller v_i and, therefore, λv_i becomes a smaller value, which forces the algorithm to take smaller steps.

However, for large datasets, $\nabla F(\theta)$ is not used. Instead, its stochastic estimate, $g(\theta)$, is calculated using mini-batches in the SGD algorithm. This estimation itself comes with a variance [16]:

$$\text{SGD fluctuation} \propto \frac{\text{learning rate}}{\text{batch size}} \tag{7}$$

In many applications, the SGD variance - more commonly known as the *SGD noise* - can improve the training results. Using a smaller batch size (which typically results in higher SGD noise) is encouraged for achieving a better generalization [26]. For these applications, moving averages can depress the generalization by reducing the SGD noise.

C. Schedulers

By using a set of user-defined parameters, schedulers adjust a global learning rate or a set of per-layer learning rates (in exchange for an increase in the complexity of hyperparameter tuning) throughout the training process. For example, in multi-step decay, the user sets a starting value, a set of milestones, and a set of decays for each milestone, to adjust a global learning rate during the training process.

A scheduler provides a way to adjust the learning rate at virtually every step of the training. The main disadvantage of using schedulers is their tuning process. Typically, a user starts with a guess or a suggestion from the literature and fine-tunes these parameters using their experience and trial and error. This process can be very time-consuming.

Another common practice for learning rate adjustment is pairing a scheduler with an adaptive optimizer. While this

can combine both approaches' benefits, it also requires fine-tuning user-defined parameters for both the scheduler and the optimizer in addition to selecting the right combination for the (scheduler, optimizer) pair.

D. Methods with Line Search

These methods monitor one of the training metrics, such as validation or training loss, to adjust the learning rate during the training [7]. A variety of line search methods have been proposed in previous work. L4 [27] requires five user-defined parameters. It maintains a minimum attainable loss throughout the training, and by locally linearizing the loss at each step, it solves a linear equation to calculate the next best learning rate. L4 can be unstable [28].

In [29], authors use a probabilistic belief over the Wolfe conditions [30] to monitor the descent and use a line search to calculate the next best learning rate. This line search requires second order information [5]. As mentioned in Section III-A, using second order information can be costly and sensitive to sharp minima.

A less computationally expensive estimation of an upper bound for a good learning rate can be obtained from the *Armijo condition* [31]. Based on this condition a "good" learning rate should give a *sufficient* decrease in loss function (Figure 2)[6]:

$$F(\theta_i + \lambda p_i) \leq F(\theta_i) + c\lambda \nabla F_i^T p_i \tag{8}$$

Here, p_i is the direction of change, $0 < c < 1$, and $\nabla F_i^T p_i$ is the directional derivative. Line search algorithms benefit from Armijo condition to search for a good learning rate [13].

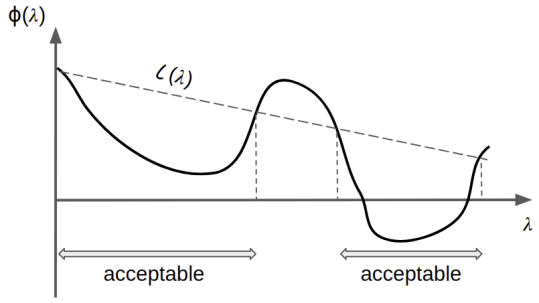

Fig. 2. Armijo condition: a "good" learning rate should give sufficient decrease in loss function. Here, $\Phi(\lambda) = F(\theta_i + \lambda p_i)$ and $l(\lambda) = F(\theta_i) + c\lambda \nabla F_i^T p_i$ are left and right sides of Equation.8, respectively. Acceptable values for "good" λ are when $\Phi(\lambda) < l(\lambda)$.

More straightforward line search methods have shown better results. Authors in [28] use the Armijo condition in a line search and showed improvements on SGD and faster convergence compared to previous work [4], [24], [32]–[35]. ASLR relates to this category because of its search algorithm. And it differs from this category because it does not perform a search after each step. Instead, it schedules a learning rate for the next epoch. Therefore, ASLR has a trivial computational cost.

[5]http://tinyurl.com/probLineSearch

[6]Figure is created based on a drawing from [13]

IV. ASLR

The intuition behind using schedulers is that there exists a global learning rate (or a set of learning rates, one per layer) that can produce a target result for a training process. By fine-tuning the scheduler parameters, a user tries to find these "good" learning rates based on the training process's outcome and using a validation set.

Adjusting learning rate based on changes in training error is likely to result in poor generalization similar to adaptive optimization methods (Section III-B) and second order information (Section III-A). As mentioned in [7], the learning rate should decay each time the validation error plateaus.

Theoretically, a decaying learning rate is necessary to guarantee convergence of SGD [36]. It is empirically shown that keeping the learning rate constant or decaying it cautiously often works better [29]. Also, a decay-only policy may get stuck around a local minima (Figure 1). In ASLR, similar to methods based on the second order information, we allow both increase and decrease in learning rate throughout the training.

In our proposed scheduler, a user fine-tunes a starting value for the learning rate, and then after each epoch, the validation error is monitored. If the validation error plateaus, a simple search algorithm starts to adjust the learning rate. This adjustment continues after every next epoch and stops as soon as an improvement is observed in validation error (see Algorithm 1). We explain each part in more detail in the following.

a) Estimating the Starting Value: From section III-A, H^{-1} can provide an accurate estimate for per-parameter learning rates λ. Let's assume λ_g (a scalar) is a good global learning rate. And $\lambda_j, (0 \leq j < n)$ are optimal per-parameter learning rates with n being the total number of trainable parameters in the network.

Let's $\lambda_{max} = \max_j\{\lambda_j\}$ be the upper bound and $\lambda_{min} = \min_j\{\lambda_j\}$ be the lower bound for learning rate. A reasonable per-layer learning rate, λ_g, must satisfy the following condition:

$$\lambda_{min} \leq \lambda_g \leq \lambda_{max} \tag{9}$$

The right inequality is from Armijo condition and the left inequality is from *curvature condition* [37]. Together, they are referred to as *The Wolfe conditions* [30].

A user can find an initial learning rate that satisfies Equation 9 with a simple *learning rate range test* [15]: running the training for a few epochs while increasing the learning rate linearly. By checking the accuracy against the learning rate, one can observe the boundaries for a reasonable starting value. Then the user can select a value between those boundaries, for example, the middle point.

b) Adjusting Process: The adaptive algorithm starts with the user-specified initial value for the learning rate. When there is no improvement in training results, it searches for the next good learning rate using a simple search algorithm shown in Algorithm 1. Figure 3 illustrates an example of learning rate adjustment with ASLR.

Algorithm 1: ASLR Search Algorithm

```
Require: initial learning rate c
  min_cost ← 1
  search_direction ← 1
  search_range ← 1
  search_steps ← 0
  while training do
    process one epoch and for each mini-batch generate
    new per-layer learning rates (c_u in Equation 10)
    cost ← validation cost
    if min_cost < cost then
      s ← Equation 10
      if c + search_direction × s = 0 then
        c ← 0.9 × c
      else
        c ← c + search_direction × s
      end if
      search_steps ← search_steps+1
      if search_steps = search_range then
        search_range ← search_range+1
        search_steps ← 0
        search_direction ← (−1)×search_direction
      end if
    else
      min_cost ← cost
      search_direction ← 1
      search_range ← 1
      search_steps ← 0
    end if
  end while
```

In Algorithm 1, an update to the learning rate is only possible after processing one epoch. A search for a better learning rate can potentially be possible after any step of the training. However, since most training scripts use SGD and not Batch Gradient Descent, the results of each step include SGD fluctuation (Equation 7). Our proposed algorithm updates the learning rate between epochs to dampen this noise and avoid the evaluation's cost after every step.

c) Step Size in Learning Rate Adjustment: Extremely small changes cannot be applied to the learning rate because each time ASLR adjusts the learning rate, the search process may take several epochs of the training to reach a good learning rate. Therefore we have no choice other than adding discontinuity to the learning rate and apply a feasible change to the learning rate while adjusting it in our algorithm.

Heuristically, we observed that feasible changes in the current learning rate, c, should be equal to $10^{\lfloor \log_{10} c \rfloor}$.

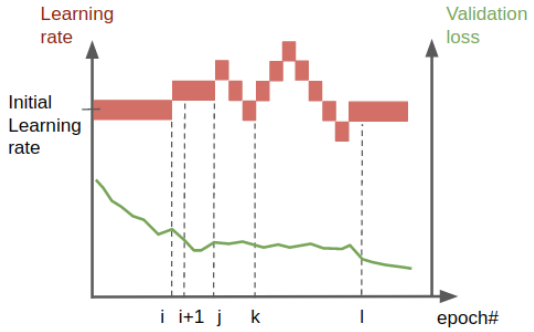

Fig. 3. Adjusting learning rate. With no improvement in validation loss at epoch i, learning rate increases by s (Equation 10). Search stops after the improvement in validation loss at epoch $i+1$. At epoch j, improvement stops again and similar to before, current learning rate increases by s. With no improvement in validation loss at the next epoch, *search_region* increases by one and *search_direction* changes. This happens again at epoch k. And continues until epoch l when there is an improvement in validation loss.

d) Search Direction and Search Range: As shown in Algorithm 1, each time that the *search_range* is increased, *search_direction* is changed. This mechanism helps to scan a range around the current learning rate for finding a good learning rate. As reported in [38], adaptive methods can suffer from generating extreme values for the learning rate. By gradually increasing the range and changing the direction, we minimize our chance of generating extremely large or extremely small learning rates.

e) Drawing the Learning Rate from a Uniform Distribution: The starting learning rate (provided by the user) and any other value calculated by the algorithm is, at best, an estimation for a good learning rate. The discontinuity created by the step size applies a limit on these estimated learning rates. For example at $c = 0.05$, the step size is $10^{\lfloor \log_{10} 0.05 \rfloor} = 0.01$. If the *search_direction*=1, the next possible value for the learning rate is 0.06. In ASLR, we will not ignore all possible values between 0.05 and 0.06.

As mentioned in Section III-A, authors in [6] explained how to (more) efficiently use second order information to calculate a trust region (norm ball) and use it to control and adjust the learning rate at each step. Motivated by their work, we fix a range around our estimated learning rates and draw per-batch and per-layer learning rates from that region. The region used in ASLR is a uniform distribution centered at the current learning rate with a width equal to the step size. With c being the current learning rate, for each step of the training, a per-layer learning rate, c_u, is calculated and provided to the optimizer as following:

$$\begin{cases} c_u \sim \mathcal{U}(c - \frac{s}{2}, c + \frac{s}{2}) \\ s = 10^{\lfloor \log_{10} c \rfloor} \end{cases} \tag{10}$$

This way, when $c = 0.05$, we draw our learning rates from $\mathcal{U}(0.045, 0.055)$ and when $c = 0.06$, we draw our learning rates from $\mathcal{U}(0.055, 0.065)$ (see Figure 4).

Because $\mu(c_u) = c$, our effective learning rate [16] for processing each epoch is still equal to c.

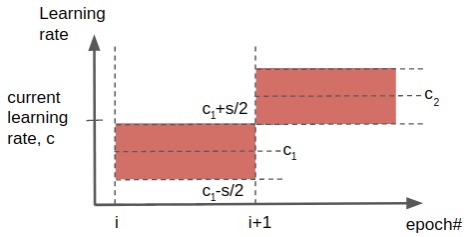

Fig. 4. Drawing learning rates from a uniform distribution. At each step, per-layer learning rates, c_u are drawn from $\mathcal{U}(c_1 - \frac{s}{2}, c_1 + \frac{s}{2})$. When c_1 is the learning rate for epoch i, calculated with Algorithm 1.

f) Limitations: We believe that it is essential for every work to state its limitations. We carefully performed extensive experiments and repeated all the reported tests multiple times. Our proposed algorithm is tested on a variety of models ranging from quantized and custom networks to popular networks with Non-quantized parameters, including very deep networks such as the VGG family and residual networks such as the ResNet and DenseNet families. We have also open-sourced our code to allow reproducibility.

Although we did not observe an example of our algorithm's failure, we can not prove or guarantee that this algorithm is superior to all other manually fine-tuned schedulers for all network topologies and training scripts.

We also can not provide a precise comparison between the time spent on tuning our algorithm parameter (starting value) against the time spent on tuning other schedulers' parameters due to the lack of reporting such processes in the literature.

V. Results

To evaluate ASLR we used a selection of moderate and hard to classify datasets consisting of ImageNet [39] and both CIFAR10 and CIFAR100 [40] datasets. We selected a variety of different network architectures, including very deep architectures, networks with skip connections, and dense architectures.

We also tested ASLR on networks with quantized parameters. This is a significant test result because hyperparameters of a network must be re-tuned after the quantization is applied [7], [41]. Our results show that ASLR can be employed to train quantized networks with no additional tuning.

Unfortunately, there are no widely recognized benchmarks to use for comparison. Therefore, in our setup, we use publicly available implementations to evaluate ASLR against other work.

In all of our experiments, the reported accuracy results are average over three runs with different seeds. We also set ASLR's initial learning rate similar to the initial rate of the network that we compared against and therefore did not have to perform the initial learning rate search described in Section IV.

In the following, we first describe our results for comparing ASLR against line search methods, which includes test results on CIFAR10 and CIFAR100 datasets using ResNet34. We then compare ASLR and different schedulers on CIFAR10 and ImageNet on various network topologies, including quantized networks.

1) Comparing with Line Search Methods: To compare our work with methods mentioned in Section III-D, we used the implementation [7] that is described at [28] and integrated ASLR into this implementation.

TABLE I
Comparing validation accuracy of ASLR with Line Search Methods on ResNet34

Dataset	Batch Size	L4	SGD_Armijo	ASLR
CIFAR10	64	87.5%	93.4%	93.6%
	128	86.2%	93.6%	94.2%
CIFAR100	64	63.7%	73.8%	74.5%
	128	60.8%	74.8%	75.7%

Table I shows a comparison between our results and two other line search methods. To generate these results, we set ASLR's initial learning rate to 0.1 (with no additional tuning) and total epochs to 150. Accuracy results are average over three runs with different seeds.

An interesting observation is the processing time between the three methods. At each step, ASLR draws the learning rates from a uniform distribution. Whereas L4 [27] and SGD_Amijo [28] have to do a line search. Compared to ASLR, these line search methods required extra time for processing each epoch. We calculated the average of per-epoch additional time needed for these methods on a desktop machine with one GPU for all the training epochs. Table II shows our results.

TABLE II
Comparing average training time per epoch between ASLR, L4, and SGD_Amijo (CIFAR10 and ResNet34).

	Training time per epoch (Seconds)		
batch size	ASLR	L4	SGD_Amijo
64	85	119	129
128	84	113	127

The validation accuracy evolution curve for experiments in Table I is shown in Figure 5. The oscillation in ASLR's curve is due to its search algorithm. Each time the validation loss plateaus, ASLR starts its search, and the search range expands after each epoch until an improvement is observed in validation loss. Changes in the learning rate during this search cause the oscillation in its validation accuracy curve.

2) Comparing with Schedulers: To our knowledge, there are no widely recognized benchmarks to use for comparing our method with methods described in Section III-C. Therefore

[7] https://github.com/IssamLaradji/sls

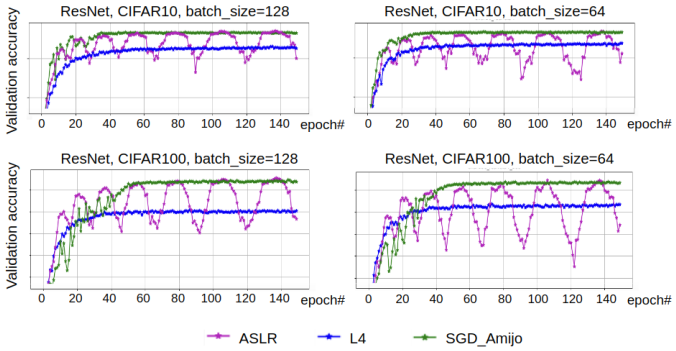

Fig. 5. Comparison between the validation accuracy evolution curve for ASLR and line search methods: L4 and SGD_Amijo.

for this part of our experiments, we have selected a diverse range of networks with publicly available implementations and already-tuned hyperparameters for CIFAR10 and ImageNet datasets.

CIFAR10: We compared ASLR with state-of-the-art results for a number of networks selected from ResNet [9], DensNet [12], WRN [11], and VGG [10] families. We also selected two reduced precision networks: WRN_1bit and CNV_1bit [42] and a custom VGG11 network [8] where parameters are quantized and fully connected layers are removed to avoid over-fitting for the CIFAR10 dataset.

TABLE III
COMPARING VALIDATION ACCURACY OF ASLR WITH SCHEDULERS ON CIFAR10.

Network	Scheduler accuracy	ASLR accuracy
Resnet20	92.2%	92.2%
Resnet56	93.3%	93.9%
DenseNet40	92.8%	92.9%
WRN20_1bit	95.2%	94.9%
VGG11_8bits	91.5%	91.4%
VGG11_6bits	91.2%	91.2%
CNV_1bit	78.5%	78.5%

The results of our comparisons are shown in Table III. In the following, we describe the schedulers used to generate the results in *Scheduler accuracy* column.

ResNet20, ResNet56, and DenseNet40 used multi-step-decay scheduler with nine, nine, and five user-defined parameters respectively [9]. WRN20_1Bit used a cosine annealing scheduler with two user-defined parameters [10]. VGG11 used a multi-step decay scheduler with seven user-defined parameters. And CNV_1Bit used a multi-step-decay scheduler with nine user-defined parameters.

To generate the results in *ASLR accuracy* column, we set the initial learning rate of ASLR equal to the initial learning rate of the scheduler that we compared against (the scheduler in the same row of the table). Results are an average of three runs with different seeds. Table III shows that ASLR can achieve similar or better results compared to highly tuned manual schedulers while having only one user-defined parameter.

Figure 6 illustrates a comparison between ASLR and the multi-step-decay scheduler used with ResNet20. Both schedulers achieved similar validation accuracy results. As shown in Figure 6, throughout the training, ASLR starts its search earlier than the first decay in the other scheduler, and by the end of the training, it almost follows the finely tuned multi-step-decay. Similar behavior was observed when training other networks in Table III.

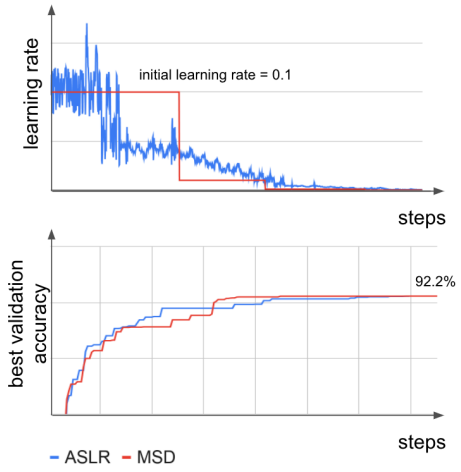

Fig. 6. Comparison between ASLR and multi-step-decay on ResNet20

ImageNet: We selected three networks from ResNet and VGG families to test ASLR on the ImageNet dataset. ResNet10, ResNet50, and VGG11. Table IV shows the results of our experiments on this dataset. The schedulers used to generate the results in *Scheduler accuracy* column are cosine annealing schedulers with two user-defined parameters [10]. ASLR's initial learning rate was set to the scheduler's initial learning rate in the same row in Table IV. All results are an average of three runs with different seeds.

TABLE IV
COMPARING VALIDATION ACCURACY OF ASLR WITH SCHEDULERS ON IMAGENET.

Network	Scheduler accuracy	ASLR accuracy
Resnet10	65.5%	66.0%
Resnet50	75.2%	75.2%
VGG11	67.7%	70.9%

VI. CONCLUSIONS

This work provided a brief review of commonly used learning rate adjustment methods and explained their gains and

[8]https://github.com/Xilinx/brevitas
[9]https://keras.io/examples
[10]https://github.com/osmr/imgclsmob

disadvantages. We described the complexity of finding reasonable learning rates and introduced an Adaptive Scheduler for Learning Rate (ASLR) with a single user-defined parameter. We explained how our algorithm adjusts the learning rate during the training process and showed that even though it has a simple algorithm, it can achieve competitive results compared to training scripts with finely-tuned hyperparameters. Our result section provided performance results for ASLR on various network topologies, including custom networks with quantized parameters. The ability to train uncommon and quantized networks is an essential feature of ASLR and shows that this scheduler can train a wide range of network designs. This feature can reduce the time for testing and designing custom networks by reducing the tuning time spent on hyperparameters for the learning rate. We also showed that ASLR has a smaller computation complexity compared to line search methods.

REFERENCES

[1] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: a deeper understanding," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 5813–5823.

[2] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convlutional neural networks," in *arXiv:1812.01187*, 2018.

[3] F. Hutter, J. Lücke, and L. Schmidt-Thieme, "Beyond manual tuning of hyperparameters," in *Künstl Intell 29*. Springer, 2015, pp. 329–337.

[4] D. P. Kingma and J. L. Ba, "Adam: a method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[5] J. Martens, "Deep learning via hessian-free optimization," in *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, 2010, pp. 735–742.

[6] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximation curvature," in *arXiv:1503.05671*, 2016.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[8] R. D. Reed and R. J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA: MIT Press, 1998.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *arXiv:1512.03385*, 2015.

[10] K. Simonyan and A. Zisserman, "Very deep convlutional networks for large-scale image classification," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[11] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *arXiv:1605.07146*, 2017.

[12] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *arXiv:1608.06993*, 2018.

[13] J. Nocedal and S. J. Wright, *Numerical optimization*, 2nd ed. Springer, 2000.

[14] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Proceedings of the 4th International Conference on Neural Information Processing Systems*, 1991.

[15] L. N. Smith, "Cyclical learning rates for training neural networks," in *arXiv:1506.01186*, 2017.

[16] S. Smith, P. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," in *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[17] B. DasGupta and H. T. Siegelmann, "On the complexity of training neural networks with continuous activation functions," in *7th ACM Conference on Learning Theory, 1994*, 1994.

[18] Z. Yao, A. Gholami, D. Arfeen, R. Liaw, J. Gonzalez, K. Keutzer, and M. W. Mahoney, "Large batch size training of neural networks with adversarial training and second-order information," in *arXiv:1810.01021*, 2020.

[19] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, "Sharp minima can generalize for deep nets," in *arXiv:1703.04933*, 2017.

[20] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang, "On large-batch training for deep learning generalization gap and sharp minima," in *arXiv:1609.04836*, 2017.

[21] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *arXiv:1611.03530*, 2016.

[22] S. Hochreiter and J. Schmidhuber, "Flat minima," in *Neural Computation*, vol. 9, 1997, pp. 1–42.

[23] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," in *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, 2019.

[24] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," in *Journal of Machine Learning Research (JMLR)*, 2011.

[25] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2015.

[26] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," in *arXiv:1804.07612*, 2018.

[27] M. Rolinek and G. Martius, "L4: practical loss-based stepsize adaption for deep learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2018.

[28] S. Vaswani, A. Mishkin, I. Laradji, M. Schmidt, G. Gidel, and S. Lacoste-Julien, "Painless stochastic gradient: Interpolation, line-search, and convergence rates," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2019.

[29] M. Mahsereci and P. Hennig, "Probabilistic line searches for stochastic optimization," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*. Curran Associates, Inc., 2015, pp. 181–189. [Online]. Available: http://papers.nips.cc/paper/5753-probabilistic-line-searches-for-stochastic-optimization.pdf

[30] P. Wolfe, "Convergence conditions for ascent methods. ii: some corrections," in *SIAM Review*, vol. 13, 1971, pp. 185–188.

[31] L. Armijo, "Minimization of functions having lipschitz continuous first partial derivatives," in *Pacific Journal of Mathematics*, vol. 16, 1966.

[32] M. Zeiler, "Adadelta:an adaptive learning rate method," in *arXiv:1212.5701*, 2012.

[33] M. Schmidt, N. L. Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," in *arXiv:1309.2388*, 2016.

[34] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2013, pp. 315–323.

[35] A. Defazio, F. Bach, and S. Lacoste-Julien, "Saga: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Proceedings of the 27th International Conference on Neural Information Processing Systems*, 2014.

[36] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, vol. 22, no. 3, pp. 400–407, 09 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729586

[37] P. Wolfe, "Convergence conditions for ascent methods," in *SIAM Review*, vol. 11, 1969, pp. 226–235.

[38] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," in *International Conference on Learning Representations (ICLR)*, 2019.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[40] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, Toronto University, 2009.

[41] S. Shin, Y. Boo, and W. Sung, "Knowledge distillation for optimization of quantized deep neural networks," in *arXiv:1909.01688*, 2019.

[42] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *arXiv:1612.07119*, 2016.