Zeroth Order GreedyLR: An Adaptive Learning Rate Scheduler for Deep Neural Network Training

Shreyas Subramanian
Amazon Web Services
Seattle, USA
subshrey@amazon.com

Vignesh Ganapathiraman
Amazon Web Services
Seattle, USA
vignesga@amazon.com

***Abstract*—Deep neural networks are a powerful tool for a wide range of applications, including natural language processing (NLP) and computer vision (CV). However, training these networks can be a challenging task, as it requires careful selection of hyperparameters such as learning rates and scheduling strategies. Despite significant advances in designing dynamic (and adaptive) learning rate schedulers, choosing the right learning rate / schedule for a machine learning task is still more art than science. In this paper, we introduce *Zeroth order GreedyLR*, a novel scheduler that adaptively adjusts the learning rate during training based on the current loss and gradient information. To validate the effectiveness of our proposed method, we conduct experiments on several NLP and CV tasks. The results show that our approach outperforms several state-of-the-art schedulers in terms of accuracy, speed, and convergence. Furthermore, our method is easy to implement, computationally efficient, and requires minimal hyperparameter tuning. Overall, our study provides a useful tool for researchers and practitioners in the field of deep learning.**

***Index Terms*—machine learning, training, learning rate scheduling**

I. Introduction

Machine learning training optimization is a crucial aspect of developing accurate and efficient machine learning models. It involves finding the best configuration of the components such as learning rate, batch size, number of layers, number of hidden units, regularization parameters, and optimizer choice for training a model on a given dataset to achieve optimal performance. One of the most crucial and influential component to configure is the learning rate. Learning rate determines the step size taken during the optimization process to update the model's parameters based on the calculated gradients in gradient-based optimizers such as SGD and ADAM. Setting the right learning rate is instrumental in determining how quickly or slowly the model learns and converges to an optimal solution. Roughly speaking bigger learning rate leads to quicker learning and smaller learning results in slower learning.

Obtained the best "fixed" learning rate that works for a given model is hard, and often found by hand-tuning over a pre-determined range of learning rates. Recently, it has become common practice to device a learning rate schedule that requires setting multiple learning rates in the due course of a training. For instance, it is often desirable to reduce the learning rate while approaching a critical point in the loss landscape to avoid divergence of the parameters. Methods such as [3, 7] hypothesize that learning in neural network happens in phases and argue that setting different learning rates to each of the phases aids convergence. Methods such as [8, 9] vary the learning rate in cycles based on preset heuristics.

Adaptive optimization methods, such as Adam (Adaptive Moment Estimation), [5] or RMSProp (Root Mean Square Propagation) [1], adjust the learning rate dynamically based on the gradients and the history of previous updates. However, it is now known that adaptive optimizers don't work very well in practice in their default settings [7, 12]. Methods such as [1, 11] aim to find the optimal learning rate at every step of the training, by posing it as a line search problem.

The main disadvantage of fixed learning rate methods are that they are very general and do not adapt to the specific characteristics of the optimization problem or the model architecture. We often see in practice that different problems and architectures require different learning rates and schedules to achieve optimal performance. On the other hand, methods that estimate learning rate based on the training information can be effective as they naturally adapt to the specifics of the optimization landscape. However, they are often expensive and slower. Therefore, there is a pressing need for a learning rate scheduler that is embarrassingly simple, yet customized and adaptable to the specific optimization problem at hand.

Our contributions are as follows:

1) We propose a novel, yet simple scheduler called Zeroth order GreedyLR which adaptively chooses the learning rate based on recent training progress
2) We conduct extensive experiments to validate the effectiveness of our method on a variety of computer vision and NLP tasks
3) We show that on most of the experiments, our method outperforms or performs competitively to popular adaptive and dynamic scheduler baselines suggesting that our method can be used as a "sane default" for common machine learning tasks.

A. Related works

Several works have studied the problem of identifying the optimal learning rate for stochastic optimizers for deep learning training. They can be largely categorized into two groups: *passive*: methods that propose a fixed learning rate

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

schedule and *active*: method that dynamically change the learning rate throughout the training.

Passive methods typically work by defining deterministic modifications to the learning rate at predetermined training iterations. Popular examples include cosine scheduler by Loshchilov and Hutter [6], step decay scheduler, polynomial scheduler to name a few. On the other hand active methods use local information to either increase or decrease the learning rate. A representative example would be the popular *ReduceLROnPlateau* [2] scheduler that reduces the current learning rate when a chosen metric (say training mini-batch loss) has plateaued.

Hypergradient descent (HD) by Baydin et al. [2] is an active method that treats learning rate α_t of a parameter θ_t as a learnable parameter instead of a hyperparameter. α_t is updated at every iteration with the standard gradient update rule using the so-called hypergradient.

Assuming a standard gradient-based optimizer such as SGD, the typical update rule for parameter θ_t at iteration t is written as:

$$\theta_t \leftarrow \theta_{t-1} - \alpha_t \nabla f(\theta_{t-1}),$$

where f is the underlying objective function and θ_t is the parameter that SGD is optimizing.

Baydin et al. [2] find the optimal α_t by finding the partial derivative of f with respect to α_t:

$$\frac{\partial f(\theta_{t-1})}{\partial \alpha} = \nabla f(\theta_{t-1}) \; . \; (-\nabla f(\theta_{t-2})). \tag{1}$$

Thus, α_t is updated using the current gradient $\nabla f(\theta_{t-1})$ and a copy of the previous gradient $\nabla f(\theta_{t-2})$. The final update hypergradient update rule is written as:

$$\alpha_t = \alpha_{t-1} + \beta \; . \; \nabla f(\theta_{t-1}) \; . \; (-\nabla f(\theta_{t-2})),$$

where β is the hypergradient learning rate.

Note that HD defines a learning rate parameter α_t for each underlying parameter θ_t. To put this in perspective, for a model such as LLaMA [10], HD would introduce an additional 1.4 trillion parameters, which can be prohibitively expensive. Additionally, HD has a hyperparameter β, which needs to be tuned manually.

GreedyLR uses a single learning rate α for all the parameters, akin to vanilla SGD, and doesn't involve any explicit learning. GreedyLR introduces two tuneable hyperparameters, but we see that the default values suggested in this paper works for most cases.

In this work, we also explore a relaxed version of hypergradient descent called hypergradient descent param-group (HD param-group), which reduces the number of parameters significantly. We call this *first order LR* We observe that HD param-group outperforms HD in several cases.

Stochastic line search (SLS) by Vaswani et al. [11] is a search-based active line search method based on the classic Armijo line search technique [1]. At a training iteration t, SLS searches for the optimal learning rate α_t that satisfies the following varation of the Armijo condition:

$$f(\theta_t - \alpha_t \nabla f(\theta_t)) \leq f(\theta_t) - c \; . \; \alpha_t ||\nabla f(\theta_t)||^2,$$

where c is the Armijo constant, a hyperparameter.

SLS is a principled method for dynamically choosing the learning rate, and has been shown to work well on standard benchmark experiments. However, SLS requires a line-search procedure at every iteration, which in turn requires an additional forward pass to compute the term on the LHS of the above equation. Other search based methods have also been considered in the past. Khodamoradi et al. [4] perform a simple heuristic search to update the learning rate when the validation loss plateaus.

II. Method

The GreedyLR scheduler amplifies the effect of step size or learning rate based on whether there is an improvement or deterioration of loss. Algorithm 1 describes the full scheduler implemented in detail. In the simplest case, we initialize a fixed $factor$ parameter in the range of $(0, 1)$, multiply the learning rate by this $factor$ when the new loss value is worse than one time-step back; or divide by the same $factor$ in case there is an improvement (loss has decreased). Our implementation in the PyTorch framework is an extension of the existing *ReduceLROnPlateau* implementation which only reduces, but does not increase LR, limiting the range of possible schedules.[3]

Loss landscapes can be very complex and noisy, and so the following practical additions to the scheduler algorithm are made to ensure good performance across problem types:

1) To help with noisy loss functions, we introduce a $threshold$ so that any increase or decrease of loss values within the $threshold$ is ignored. We also implement a smoothing window that calculates the streaming average of the loss value within the specified window, and uses this to output the learning rate for the next step. This is a parameter that can be set to $True$ pr $False$ depending on whether smoothing is required. For example, for a window length of 10, we continuously average loss values up to the $10th$ value, then continue to move the $10-$sized window with calculation of the average loss within the window. The average loss is then used as a point estimate for comparing current and previous loss.
2) We implement three additional parameters that help mitigate impulsive reactions to change in loss values:
    a) *Patience* - Number of epochs after which learning rate will be reduced or increased. For example, with *patience* 5, we measure if there is an improvement continuously for 5 epochs before increasing learning rate. Similarly, we measure if there is a continuous deterioration for 5 epochs before decreasing learning rate.

[2]https://keras.io/api/callbacks/reduce_lr_on_plateau/

[3]https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html

b) *Cooldown* - Number of epochs to continue decreasing learning rate when loss has stopped increasing, before testing for new conditions
c) *Warmup* - Number of epochs to continue increasing learning rate when loss has stopped decreasing, before testing for new conditions

3) We allow users to define a upper and lower bound learning rate that is output from the scheduler. Any learning rate produced by the scheduler is limited within these bounds.
4) We also implement a reset functionality for problems where resetting all scheduler parameters midway in training after a particular epoch may be beneficial.

In summary, Algorithm 1 for the GreedyLR scheduler takes in several parameters such as the $optimizer$, $factor$, $patience$, $cooldown$, $warmup$, $minLR$, $maxLR$, $smooth$, $window$, and $reset$. The algorithm initializes the learning rate (lr) and other variables such as $bestLoss$, $warmupCounter$, and $cooldownCounter$. We then calculate the validation loss for a given epoch using the optimizer's current learning rate. This can be any metric other than validation loss as well; the main assumption here is that minimization is a "good" direction. Without loss of generality the algorithm can be rewritten for maximization as well. Also note that the $threshold$ parameter is implicitly defined in the function that decides if there is an improvement. For example, $valLoss < bestLoss$ implies that the current validation loss is better than the best loss so far, beyond the $threshold$ value. To further help with noisy loss trajectories we use the $smooth$ boolean parameter. If $smooth$ is true, the validation loss is calculated as a streaming average over the $window$ length specified. The algorithm then updates the learning rate based on the validation loss and the number of consecutive good and bad epochs. If the number of consecutive good epochs exceeds the $patience$ parameter, the learning rate is increased by a predefined $factor$. If the number of consecutive bad epochs exceeds the $patience$ parameter, the learning rate is decreased by the same $factor$. If the $reset$ parameter is greater than 0 and the number of epochs exceeds the $reset$ parameter, the learning rate is reset to its initial value $initLR$, and the counters and streaming average values are reset.The scheduler can also be used with a $warmup$ and $cooldown$ stage, which continues the increase or decrease of the learning rate, despite having opposing signals from other checks. For example, with $cooldown$, the algorithm continues to decrease the learning rate for the $cooldown$ number of steps even if loss decreases. The algorithm finally returns the learning rate within the range of minLR and maxLR, which the optimizer uses as the next learning rate for the following epoch.

Algorithm 1 GreedyLR Scheduler

procedure GREEDYLR($optimizer$, $factor$, $patience$, $cooldown$, $warmup$, $minLR$, $maxLR$, $smooth$, $window$, $reset$)
  Initialize: $lr \leftarrow initLR$, $bestLoss = \infty$, $warmupCounter = 0$, $cooldownCounter = 0$, $numGoodEpochs = 0$, $numBadEpochs = 0$
  $trainLoss, valLoss \leftarrow$ function to train model for one epoch
  if $smooth$ is $True$ **then**
    $valLoss \leftarrow$ streaming average of $valLoss$ in $window$
  end if
  if $valLoss < bestLoss$ **then**
    $bestLoss \leftarrow valLoss$
    $numGoodEpochs \leftarrow numGoodEpochs + 1$
    $numBadEpochs \leftarrow 0$
  else
    $numGoodEpochs \leftarrow 0$
    $numBadEpochs \leftarrow numBadEpochs + 1$
  end if
  if $cooldownCounter < cooldown$ **then**
    $cooldownCounter \leftarrow cooldownCounter + 1$
    $numGoodEpochs \leftarrow 0$
  else
    $cooldownCounter \leftarrow 0$
  end if
  if $warmpupCounter < warmup$ **then**
    $warmupCounter \leftarrow warmupCounter + 1$
    $numBadEpochs \leftarrow 0$
  else
    $warmupCounter \leftarrow 0$
  end if
  if $numGoodEpochs > patience$ **then**
    $lr = \frac{lr}{factor}$
  end if
  if $numBadEpochs > patience$ **then**
    $lr = lr \times factor$
  end if
  if $reset > 0$ and $epochs > reset$ **then**
    $lr \leftarrow initLR$
    $numGoodEpochs \leftarrow 0$
    $numBadEpochs \leftarrow 0$
    if $smooth$ is $True$ **then**
      delete streaming average
    end if
  end if
  return $lr \in [minLR, maxLR]$
end procedure

III. EXPERIMENTS

The GreedyLR scheduler is implemented in PyTorch and tested with datasets, models, optimizers and schedulers available in Huggingface. The design of experiments conducted are summarized below in Table I. All experiments were conducted on ml.g4dn.8xlarge and ml.g4dn.16xlarge instances using Amazon SageMaker. Conducting an exhaustive set of experiments can be prohibitively expensive, even with a small sample of available models, optimizers, schedulers and compatible datasets. Considering the table below, we would

have to run 4800 experiments, with other parameters like initial learning rate, batch size and dataset configurations being fixed. To help assess if GreedyLR can be used as a good, default scheduler for training across use cases, we select 132 experiments from the full DOE above, and collect over 200 datapoints comparing GreedyLR performance against combinations of popular optimizers and schedulers, at various stages of training runs. For each experimental run, we measure if the GreedyLR scheduler beats the baseline optimizer and scheduler combination at 10%, 50% and 100% of the maximum steps. We do this to record a snapshot of the entire training run, as opposed to focusing on only some stages of training (say initial convergence, or only final loss). We also record the difference in final loss at the end of each run, and test the scheduler on 6 different tasks - translation, question and answer (QnA), summarization, named entity recognition (NER), image classification and image segmentation. We use the same seed and initial learning rate for each run comparing a baseline scheduler and GreedyLR, and use the same base optimizer when comparing schedulers. For the base optimizers, we use default values provided in the huggingface implimentation as is common to do, and use a common initial Learning Rate (initial LR). For all runs using GreedyLR, we use a patience of 10, lower bound LR as 10% of the starting LR (e.g. $1e-5$ for a initial LR of $1e-5$), and smoothing with a window size of 50. With all experiments, we caveat that an exhaustive grid search, along with a simple fixed LR scheduler and a basic optimizer like SGD can theoretically beat all other options considered. However, for many model-dataset combinations with complex loss landscapes, we aim to find a good default scheduler with an overall average benefit across all stages of convergence.

Tables II and III summarize detailed results that are presented in tables V to X for each task. Referring to Table II, we first collate points in the training where GreedyLr with the same optimizer as the baseline scheduler clearly beats ("yes"), beats with no significant difference ("yes*"), clearly does not beat ("no"), and does not beat, but with no significant difference ("no*"). In these experiments, we define insignificant difference in any stage of the training (i.e., 10%, 50%, and 100% of the max steps) as the absolute difference in loss being less than $+/-0.1$, and record these as "yes*" and "no*" cases. Table III calculates summary statistics from results that are presented in tables V to X. Across use cases, GreedyLR is overall as good or better ("yes", "yes*" and "no*") than schedulers tested over 86% of the time, and overall better ("yes" and "yes*") over 57% of the time. GreedyLR is clearly better about 25% of the time ("yes"). Looking at just the final loss, GreedyLR with the same base optimizer performs better over 91% of the time. We also record the average benefit (when loss values from GreedyLR are lower than the competing scheduler), maximum benefit across experiments, and the maximum deficit (when GreedyLR is worse). A histogram of the final loss benefit is shown in Fig. 1; as expected we see that there is a positive skew signifying an overall benefit. We see in Table IV that GreedyLR has consistent performance across stages, with a small but important benifit in the initial convergence stage (10%).

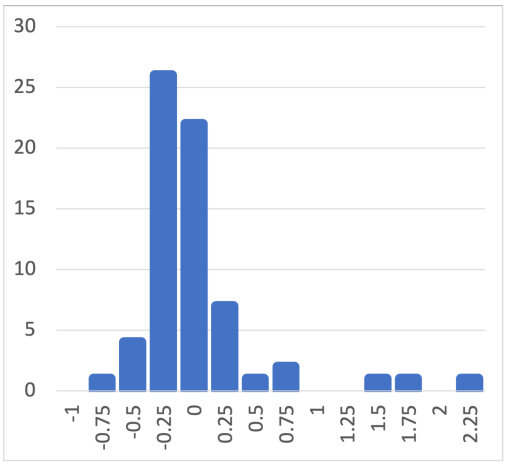

Fig. 1: Distribution of final loss delta (positive means GreedyLR was better).

TABLE I: Design of Experiments (DOE) for performance comparisons between GreedyLR and other schedulers

Models	**Datasets**
google/pegasus-x-base	wmt16
facebook/wmt19-de-en	Opus 100
facebook/blenderbot_small-90M	News Commentary
google/long-t5-tglobal-base	SQUAD
xlm-roberta-base	Adversarial QA
bert-based-uncased	Quoref
bart-base	CIFAR-10
Resnet-50	CIFAR-100
Resnet-152	Tiny Imagenet
google/vit-base-patch16-224	segments/sidewalk-semantic
nvidia/mit-0	amazon_reviews_multi
facebook/bart-base	XSUM
t5-base	Conllpp
bert-base-uncased	Wikiann
xlm-roberta-base	Xglue
camembert-large	
Optimizers	**Schedulers**
AdamW	Linear
Adafactor	Cosine
Adagrad	Polynomial
SGD	Constant with warmup
	GreedyLR (ours)

TABLE II: Results summary - counts of how often GreedyLR scheduler beats other schedulers across all experiments, and at three stages (10, 50 and 100% of max steps). Yes means that GreedyLR with the same base optimizer beats the scheduler in comparison, and no means that it does not. * indicates that the loss delta at the measured point is less than $+/-0.1$

yes	yes*	no	no*	sum	Final loss delta in +/- 0.1
48	64	26	58	196	39

A. *Task specific results*

Table V describes our results on translation tasks. Models tested include the Pegasus X base, WMT19, Blender Bot and T5 Long. The table includes the full huggingface model path for and dataset configuration. We fine tune models with datasets accessible through huggingface and explore different

TABLE III: Summary of performance calculated from Table II

Metric	**Percentage (%)**
Overall as good or better	86.73
Overall Better	57.14
Overall Worse	13.271
Final as good or better	91.67
Final worse	8.33
Overall as good	62.24
Clearly better	24.49
Metric	**Baseline - GreedyLR loss**
Average benefit	0.16
Max benefit	2.3
Max deficit	-0.62

TABLE IV: Summary of performance calculated from Table by stage showing what percentage of times GreedyLR is overall as good, or better II

Stage 1 (10%)	Stage 2 (50%)	Stage 3 (100%)
92.42	81.81	85.94

max steps, initial learning rates, and configurations (here, this means we try two pairs of translations - German to English, and English to French). The main motivation to explore multiple max steps is to reduce computational burden. Once again, several more experiments can be performed just for the translation task, but our aim with this work was to cover a breadth of use cases. As we can see from the results, GreedyLR performance is predominantly similar to other schedulers in comparison. Performance during the different stages also varies; in this case, we see initial convergence has a slight benefit over other stages where performance is more or less the same. Two experimental runs with BlenderBot and the Opus100 dataset did not finish (DNF); we do not penalize the baseline schedulers for this, and omit these datapoints for the overall summary calculations as this may be seed specific, or implementation specific. Table VI describes our results on the QnA task. Models tested include the XLM Roberta base, BERT base (uncased), and BART. Once again, the table includes the full huggingface model path for and dataset configuration used. We use huggingface provided scripts to fine tune models with datasets mentioned and explore different max steps with the same initial LR. We see consistent benefits across stages when comparing with base optimizers and schedulers.

Table VII describes our results on the Image classification task. Models tested include the Resnet-50, Resnet-152, and the Google Vision Transformer Base with a patch size of 224. We use the CIFAR-10, CIFAR-100 and the TinyImageNet datasets available through Huggingface. Once again, the table includes the full huggingface model path for and dataset configuration used. We use huggingface provided scripts to fine tune models with datasets mentioned with the same initial LR ($1e-4$) and max steps (1000). We see consistent benefits across stages when comparing with base optimizers and schedulers. One of the runs training TinyImageNet with Vision Transformer recorded the highest benefit across all runs (loss history and LR scheduler history included in Fig. 2 and Fig. 3, with GreedyLR curves in red). This is particularly noteworthy as the model is complex, and the baseline optimize and schedulers are commonly used - Adagrad with polynomial schedule. Note that loss histories for all runs can be made available to interested readers.

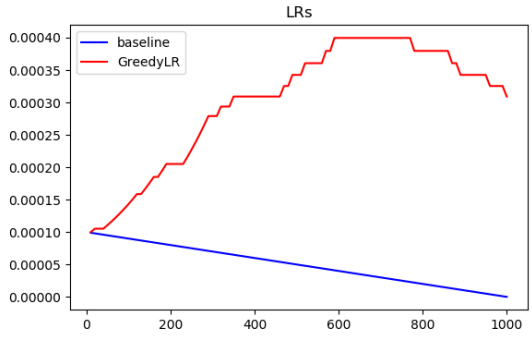

Fig. 2: Scheduler histories for Polynomial and GreedyLR

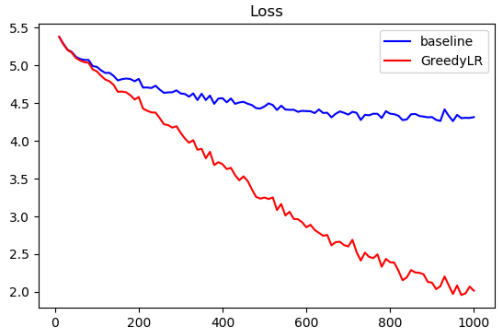

Fig. 3: Loss histories for Adagrad when used with Polynomial and GreedyLR schedulers

Table VIII describes our results on the Image segmentation task. We test the NVIDIA MIT-0 base model and fine tune the model with the Segments Sidewalk dataset available through Huggingface. The table compares results from running fine tuning using the base optimizers and schedulers, against the same base optimizers with GreedyLR. We also include the full huggingface model and dataset path used. We use huggingface provided scripts to fine tune models with datasets mentioned with the same initial LR ($1e-3$) and max steps (1000). With a LR of $1e-4$, no optimizer-scheduler combination showed any convergence or learning, so we omit these results from the paper. Across stages, and especially at max steps, we see GreedyLR predominantly beats other schedulers of choice.

Table IX describes our results on the Summarization task. Models tested include the Facebook/Bart Base and T5 Base models. We use the Amazon Reviews (multi) and XSUM datasets available through Huggingface. Once again, the table includes the full huggingface model path for and dataset configuration used. We use huggingface provided scripts to fine tune models with datasets mentioned with the same initial LR ($1e-4$) and max steps of 5000 for AdamW and Adafactor cases, but 1000 for all other cases. Overall, inital stages see a higher benefit, with the final stage being slightly worse in the case of GreedyLR. Overall, we consider that there is no clear benefit, and this is apparent from the loss history seen

TABLE V: Results for translation experiments. * - No significant difference (less that +/- 0.1 delta). Two experiments diverged with schedulers, marked as DNF for Did Not Finish.

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
google/ pegasus-x base	wmt16	de-en	5000	1.00E-04	AdamW	Linear	Yes*	No*	No	-0.2548
		de-en	5000	1.00E-04	Adafactor	Cosine	No*	No	No	-0.2475
		de-en	5000	1.00E-04	Adagrad	Polynomial	Yes*	No	No	-0.2117
		de-en	5000	1.00E-04	SGD	Constant with warmup	Yes*	No*	No	-0.2548
facebook/ wmt19 de-en	wmt16	de-en	5000	1.00E-04	AdamW	Linear	Yes*	Yes*	Yes*	0.0373
		de-en	5000	1.00E-04	Adafactor	Cosine	Yes*	Yes*	Yes*	0.058
		de-en	5000	1.00E-04	Adagrad	Polynomial	No*	No*	No*	-0.0252
		de-en	5000	1.00E-04	SGD	Constant with warmup	No*	No*	No*	-0.0215
		de-en	1000	1.00E-03	Adagrad	Polynomial	No*	No*	No*	-0.0091
		de-en	1000	1.00E-03	SGD	Constant with warmup	No*	No*	No*	-0.0049
facebook/ blender bot small 90M	Opus 100	de-en	1000	1.00E-04	AdamW	Linear	Yes*	Yes*	No*	-0.0021
		de-en	1000	1.00E-04	Adafactor	Cosine	Yes*	Yes*	Yes*	0.0121
		de-en	1000	1.00E-04	Adagrad	Polynomial	Yes	No*	DNF	-
		de-en	1000	1.00E-04	SGD	Constant with warmup	No*	No*	DNF	-
google/ long-t5 tglobal base	News Commentary	en-fr	1000	1.00E-03	AdamW	Linear	Yes*	Yes*	Yes*	0.019
		en-fr	1000	1.00E-03	Adafactor	Cosine	No*	Yes*	Yes*	0.087
		en-fr	1000	1.00E-03	Adagrad	Polynomial	Yes*	Yes*	Yes*	0.00221
		en-fr	1000	1.00E-03	SGD	Constant with warmup	No	No	No	-0.4363

below in Fig. 4 and Fig. 5. We suspect that no optimizer-scheduler combination can have a clear benefit when the loss landscape is very noisy (see Fig. 5. Finally table X describes

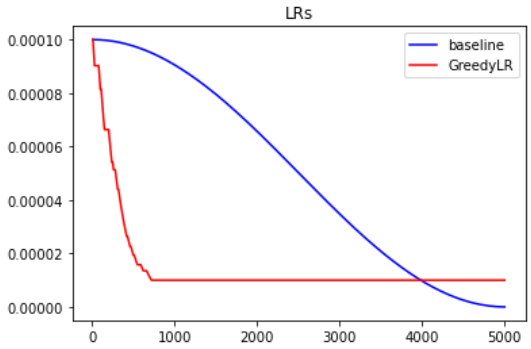

Fig. 4: Scheduler histories for Cosine and GreedyLR

our results on the Named Entity Recognition (NER) task. Models tested include the BERT base (uncased), XLM Roberta base, and Camembert Large. We use the XGLUE, WIKIANN and CONLLP datasets available through Huggingface. Once again, the table includes the full huggingface model path for and dataset configuration used. We use huggingface provided scripts to fine tune models with datasets mentioned with the same initial LR ($1e^{-4}$) and max steps of 1000. We see significant benefit in the case of Camembert models (see

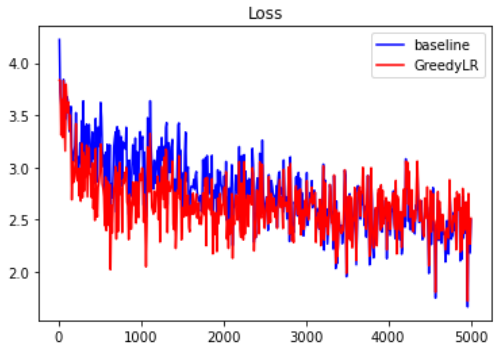

Fig. 5: Loss histories for Adafactor when used with Cosine and GreedyLR schedulers

example loss history in Fig. 6 and scheduler LR history 7 for SGD with constant scheduler) and mixed results with a slight benefit when using GreedyLR for the other two models considered.

B. Comparing with first-order GreedyLR

A natural extension of the above GreedyLR concept that reacts based on sign of the change in loss values, is to react based on the gradient magnitude. One such implementation is the Hypergradient descent concept, where a per-parameter or per-parameter-group learning rate is dynamically tuned

TABLE VI: Results for QnA experiments. * - No significant difference (less that +/- 0.1 delta)

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
xlm-roberta-base	SQUAD	-	5000	1.00E-04	AdamW	Linear	No	Yes	Yes	1.772
	-		5000	1.00E-04	Adafactor	Cosine	Yes	Yes	Yes	0.3095
	-		5000	1.00E-04	Adagrad	Polynomial	Yes	Yes	No*	-0.0975
	-		5000	1.00E-04	SGD	Constant with warmup	No*	No	No	-0.3229
bert-based-uncased	Adversarial QA	-	1000	1.00E-04	AdamW	Linear	No	No*	Yes*	0.0319
	-		1000	1.00E-04	Adafactor	Cosine	Yes*	No	Yes	0.4981
	-		1000	1.00E-04	Adagrad	Polynomial	Yes*	No	Yes	0.1219
	-		1000	1.00E-04	SGD	Constant with warmup	No*	Yes*	No*	-0.0179
bart-base	quoref	-	1000	1.00E-04	AdamW	Linear	Yes	No	Yes	1.598
	-		1000	1.00E-04	Adafactor	Cosine	No*	No*	Yes*	0.087
	-		1000	1.00E-04	Adagrad	Polynomial	No*	No	Yes	0.9319
	-		1000	1.00E-04	SGD	Constant with warmup	Yes	No	No*	-0.0178

TABLE VII: Results for Image Classification experiments. * - No significant difference (less that +/- 0.1 delta). † - Ignoring mismatched sizes and a train val split of 0.2, although we are not evaluating here.

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
Resnet-50	CIFAR-10	†	1000	1.00E-04	AdamW	Linear	Yes	Yes	Yes	0.2977
		†	1000	1.00E-04	Adafactor	Cosine	Yes	Yes	Yes	0.313
		†	1000	1.00E-04	Adagrad	Polynomial	Yes	Yes	Yes	0.1308
		†	1000	1.00E-04	SGD	Constant with warmup	Yes*	Yes*	No*	-0.0019
Resnet-152	CIFAR-100	†	1000	1.00E-04	AdamW	Linear	Yes	Yes	Yes	0.4857
		†	1000	1.00E-04	Adafactor	Cosine	Yes	Yes	Yes	0.5629
		†	1000	1.00E-04	Adagrad	Polynomial	No*	No*	No*	-0.0048
		†	1000	1.00E-04	SGD	Constant with warmup	Yes*	Yes*	No*	-0.0006
google/vit-base-patch16-224	Tiny Imagenet	†	1000	1.00E-04	AdamW	Linear	Yes	No	Yes*	0.025
		†	1000	1.00E-04	Adafactor	Cosine	Yes	No	Yes*	0.0466
		†	1000	1.00E-04	Adagrad	Polynomial	Yes	Yes	Yes	2.3
		†	1000	1.00E-04	SGD	Constant with warmup	No*	No*	No*	-0.015

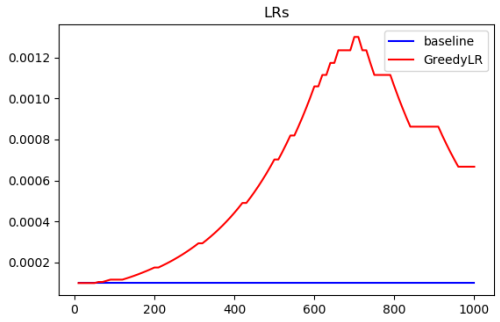

Fig. 6: Scheduler histories for Constant and GreedyLR

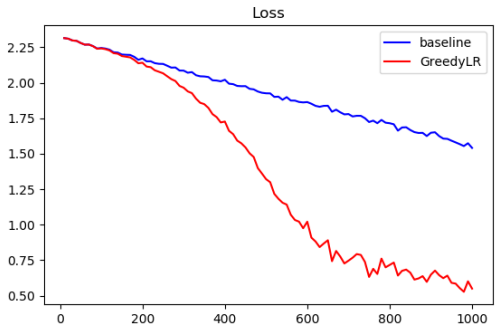

Fig. 7: Loss histories for SGD when used with constant and GreedyLR schedulers

with a separate optimizer. In our early experiments, we find that allowing GreedyLR to tune a global LR rate can be more beneficial from a per-parameter tuning; see Figures 8, 9 and 10 for loss histories, GreedyLR scheduler, and the

TABLE VIII: Results for Semantic segmentation experiments. * - No significant difference (less that +/- 0.1 delta).

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
nvidia/mit-0	segments sidewalk semantic	-	1000	1.00E-03	AdamW	Linear	Yes*	Yes*	Yes*	0.052
		-	1000	1.00E-03	Adafactor	Cosine	Yes	Yes	Yes	0.3671
		-	1000	1.00E-03	Adagrad	Polynomial	Yes*	Yes	Yes	0.1589
		-	1000	1.00E-03	SGD	Constant with warmup	Yes*	Yes*	No	-0.1583

TABLE IX: Results for Summarization experiments. * - No significant difference (less that +/- 0.1 delta). § - Here we use a source prefix - "summarize:"

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
facebook/bart-base	amazon reviews multi	§	5000	1.00E-04	AdamW	Linear	Yes	Yes*	No*	-0.0485
		§	5000	1.00E-04	Adafactor	Cosine	Yes	Yes	No*	-0.0115
		§	1000	1.00E-04	Adagrad	Polynomial	Yes*	No*	No*	-0.0333
		§	1000	1.00E-04	SGD	Constant with warmup	No	No	No	-0.621
t5-base	xsum	§	1000	1.00E-04	AdamW	Linear	Yes*	No*	Yes*	0.033
		§	1000	1.00E-04	Adafactor	Cosine	Yes*	Yes*	No*	-0.0165
		§	1000	1.00E-04	Adagrad	Polynomial	Yes*	No*	No*	-0.0106
		§	1000	1.00E-04	SGD	Constant with warmup	No*	No*	No	-0.1129

TABLE X: Results for Named Entity Recognition (NER) experiments. * - No significant difference (less that +/- 0.1 delta).

Model	Dataset	Dataset config or additional arguments (if any)	Max steps	Initial LR	Baseline Optimizer	Basline Scheduler	Beats at 10% max steps	Beats at 50% max steps	Beats at 100% max steps	Final loss delta (Ours - Baseline)
bert-base-uncased	conllpp	-	1000	1.00E-04	AdamW	Linear	No*	Yes*	No*	-0.0493
		-	1000	1.00E-04	Adafactor	Cosine	No	No*	Yes*	0.052
		-	1000	1.00E-04	Adagrad	Polynomial	Yes*	Yes*	Yes*	0.0211
		-	1000	1.00E-04	SGD	Constant with warmup	Yes*	Yes*	No*	-0.0267
xlm-roberta-base	wikiann	en	1000	1.00E-04	AdamW	Linear	No*	Yes	No*	-0.0343
		en	1000	1.00E-04	Adafactor	Cosine	No*	No*	No*	-0.01
		en	1000	1.00E-04	Adagrad	Polynomial	Yes*	Yes*	Yes*	0.0045
		en	1000	1.00E-04	SGD	Constant with warmup	Yes*	No*	No*	-0.0267
camembert-large	xglue	ner	1000	1.00E-04	AdamW	Linear	Yes*	Yes*	Yes*	0.0117
		ner	1000	1.00E-04	Adafactor	Cosine	Yes*	Yes*	Yes*	0.0132
		ner	1000	1.00E-04	Adagrad	Polynomial	Yes*	Yes	Yes	0.2696
		ner	1000	1.00E-04	SGD	Constant with warmup	Yes	Yes	Yes	0.9912

calculated per-parameter and per-parameter-group scheduled from hypergradient descent for a particular parameter group. In Fig. 10, the blue band shows the extent of per-parameter trajectories, whereas the blue solid line is the mean trajectory when using hypergradient descent on all parameters. The Red line shows the per-parameter-group trajectory when compared to the constant baseline when used with SGD.

IV. Conclusion

In this paper, we test a novel scheduler zeroth-order GreedyLR with multiple tasks, models, datasets and configurations against popularly used optimizers and schedulers. We show that GreedyLR can be a good, default options for a scheduler across task types, and performs as good, or better in a significant majority of cases, across different stages of training. We also provide initial results against a natural

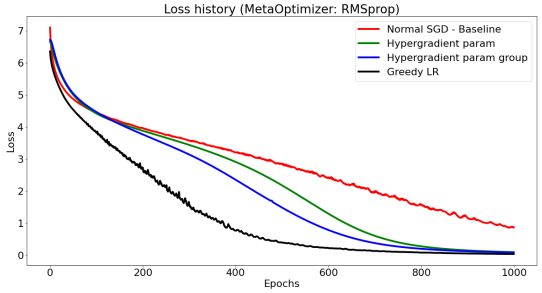

Fig. 8: Loss histories comparing GreedyLR (black), with SGD baseline (red), per-parameter (green) and per-group (blue) hypergradient descent

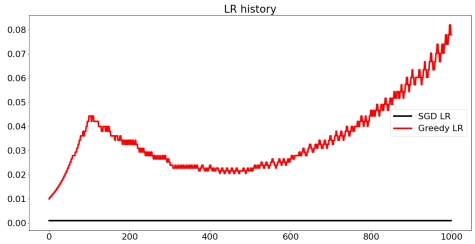

Fig. 9: Generated GreedyLR schedule for LR compared to constant SGD schedule

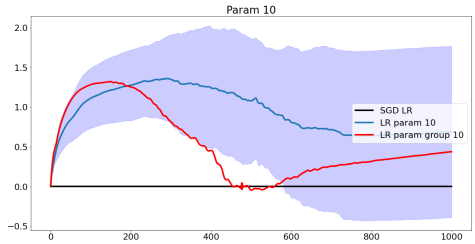

Fig. 10: Per-parameter and per-group LR histories calculated using hypergradient descent

evolution of our scheduler, hypergradient descent or first-order GreedyLR, and show that global LR tuning is still beneficial even for training over complex loss landscapes.

References

[1] Larry Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16:1–3, 1966.

[2] Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark W. Schmidt, and Frank D. Wood. Online learning rate adaptation with hypergradient descent. *ArXiv*, abs/1703.04782, 2017.

[3] Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *ArXiv*, abs/1406.2572, 2014.

[4] Alireza Khodamoradi, Kristof Denolf, Kees A. Vissers, and Ryan Kastner. Aslr: An adaptive scheduler for learning rate. *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[6] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv: Learning*, 2016.

[7] David Macêdo, Pedro Dreyer, Teresa B Ludermir, and C. Zanchettin. Training aware sigmoidal optimizer. *ArXiv*, abs/2102.08716, 2021.

[8] Leslie N. Smith. Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, 2015.

[9] Leslie N. Smith and Nicholay Topin. Super-convergence: very fast training of neural networks using large learning rates. In *Defense + Commercial Sensing*, 2017.

[10] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aur'elien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023.

[11] Sharan Vaswani, Aaron Mishkin, Issam Hadj Laradji, Mark W. Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. Painless stochastic gradient: Interpolation, line-search, and convergence rates. In *Neural Information Processing Systems*, 2019.

[12] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *NIPS*, 2017.