Speeding up Stochastic Gradient Descent: Going Beyond First Order Methods

Zeeshan Memon
zeeshan.memon@emory.edu
Emory University
Atlanta, Georgia, USA

Atharva Negi
atharva.negi@emory.edu
Emory University
Atlanta, Georgia, USA

Harsit Upadhya
harsit.upadhya@emory.edu
Emory University
Atlanta, Georgia, USA

Mitchell Scott*
mitchell.scott@emory.edu
Emory University
Atlanta, Georgia, USA

ABSTRACT

Stochastic gradient descent (SGD) is an extremely versatile optimization algorithm used for large dataset problems in machine learning. However, SGD is still slow to converge, leading to alternative algorithms, like RMSProp and ADAM, which speed up convergence by weighting the parameters' gradients differently. Nonetheless, these first order methods cannot account for parameter correlation, so a second order method- which accounts for curvature of the loss function- must be considered. We present a nonlinear iterative method- nlTGCR, which computes approximations to the true Hessian using the Fisher Information Matrix (FIM). On deep neural networks, we observe our proposed method outperforms standard first order methods on image classification using the CIFAR-10 dataset while taking a fraction of the time.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Mathematics of computing** → **Computations on matrices**; *Solvers*; Maximum likelihood estimation.

KEYWORDS

Deep Neural Network, Network Training, Fisher Information, Preconditioning

ACM Reference Format:
Zeeshan Memon, Atharva Negi, Harsit Upadhya, and Mitchell Scott. 2024. Speeding up Stochastic Gradient Descent: Going Beyond First Order Methods. In *Proceedings of International Conference on Machine Learning and Computing (ICMLC '25).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

*Authors are listed in alphabetic order, but this author wrote the entirety of the report.

ICMLC '25, February 14-17, 2025, Guangzhou, China

ACM ISBN 978-1-4503-XXXX-X/24/12…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Stochastic gradient descent (SGD) is a quintessential minimization algorithm, leveraging the well known full gradient descent algorithm over large data sets. While SGD has nice theoretical convergence results[1, 12], the practical convergence rate of vanilla SGD is slow, deteriorating as the data set gets larger. Since this SGD algorithm is run many times within every epoch for training any machine learning architecture, specifically neural networks, these seconds forfeited by a slow algorithm add up quickly.

This slow convergence of SGD is what motivated researchers in the early 2010's to develop methods like Root Mean Squared Propagation (RMSProp)[7], which uses an exponentially weighted moving average (EWMA) to focus on the more recent gradients in deciding the update and prevents premature drop-off of the search step. Further improvements were proposed with Adaptive Momentum Estimation (ADAM)[8], where they kept the EWMA of the gradients' squared- analogous to RMSProp- and augmented the search direction by computing the EWMA of the gradients themselves. Both ideas are examples of "first-order methods" as they only compute the first derivative of the loss function, the gradient. Additionally they might be referred to as "diagonal preconditioners" as they weight the gradients different, like a diagonal matrix acting on the gradient vector. N.B. "preconditioners" just refer to any matrix $\mathbf{A}$ where the condition number of $\mathbf{A}^{-1}\mathbf{B}$ is lower than the condition number of $\mathbf{B}$, by itself. Other nice properties are requiring $\mathbf{A}$ to have some structure that makes it easier to invert.

These first order methods, just like gradient descent in the deterministic optimization setting, fail to account for the curvature of the loss function, which is computed via the Hessian. Furthermore, under deterministic settings, computing the Hessian is computationally difficult, leading to storing and inverting a large dense matrix. This problem is exacerbated in the stochastic setting as small noise from batched data magnifies into large perturbations upon matrix inversion, leading to an ill- conditioned, indefinite Hessian. This is why most second-order methods will use an approximation to the true Hessian.

One approximation to the true Hessian is Full-Matrix AdaGrad[2], which uses a smoothed outer product of the gradients; however, this results in inverting a $D \times D$ matrix at each epoch, where D is the dimension of the gradient vector. Some methods assume specific structure of the Hessian and design approximations based on

this structure. One such example is Shampoo[5], which preconditions the gradient step using Kronecker structure on a flattened D^2 dimensional parameter space. Assuming Kronecker structure specifically appears in deep neural network training algorithms like K-FAC[11].

K-FAC uses this assumption, not on the Hessian, but the Fisher Information Matrix (FIM), which in expectation over all of the different configurations of data given the specific batch size, is the Hessian of the neural network. Since that is infeasible to compute, the FIM can be thought of as a symmetric, positive definite (SPD) approximation to the true Hessian. This technique of doing neural network training with the FIM is not novel, as proposed by the Fish-Leg method[4]. However, this method uses ML to learn a proper inverse, which might be inaccurate.

Instead, we rely on the SPD structure of the FIM to use the Generalized Conjugate Residual (GCR)[3] framework, which solves $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is SPD. However, in neural network training, there is no expectation of a linear activation function, so we need a method which allows for nonlinear updates $r_j = -f(x;\theta)$, as proposed by He[6]. The objective of this paper is clear; we propose a method to accelerate neural network training. We use the nonlinear Truncated Generalized Conjugate Residual (nlTGCR) framework with the FIM to design an efficient and scalable optimization algorithm for neural network training.

2 METHODOLOGY

The standard SGD algorithm consists of minimizing the loss function $\mathcal{L}(\theta)$, where θ are the weights for the neural network. We do this by computing the gradient (of the t^{th} epoch), $\mathbf{g}_t = \nabla_\theta \mathcal{L}_t(\theta)$. This means, given a learning rate schedule,$\{\eta_t\}_{t=0,1,\cdots}$, the SGD algorithm can be written as

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{M}^{-1}\mathbf{g}_t. \tag{1}$$

The matrix $\mathbf{M}$ is the preconditioning matrix, and for vanilla SGD, $\mathbf{M} = \mathbf{I}$, the identity matrix. This framework allows us to write other optimization algorithms. For example, if we let $\mathbf{s}_t = \frac{1}{t}\sum_{\tau=1}^{t} g_\tau^2$, or the Root Mean Squared for the gradient $\mathbf{g}_t$, then the preconditioning matrix $\mathbf{M}$ can be written as

$$\mathbf{M}^{-1}_{\text{RMSProp}} = \text{diag}\left(\sqrt{\mathbf{s}_t} + \epsilon\right)$$
$$\mathbf{M}^{-1}_{\text{ADAM}} = \text{diag}\left(\sqrt{\mathbf{s}_t} + \epsilon\right).$$

This highlights the similarity of RMSProp and ADAM, which both use the smoothed square root of the EWMA of gradients squared, differing only in their search direction. It also corroborates the terminology as to why they are called diagonal preconditioners.

Before nlTGCR is explained in detail, we derive some components of the method. First we turn our attention to our inspiration from deterministic optimization, the generalized Gauss-Newton matrix. Let us consider a loss function $\ell(y, z)$ which is convex in z. Here y is the correct labels and $z = f(x;\theta)$ is the model with input data x and parameters θ. Examples of convex losses include negative log-likelihood and cross-entropy as well as least squared loss, used later in this derivation. This means that the Gauss-Newton approximation to the Hessian is

$$\mathbf{G} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{J}_i^\top \mathbf{H}_i \mathbf{J}_i, \tag{2}$$

where $\mathbf{J}_i$ is the Jacobian of $f(x_i,\theta)$ with respect to θ, and $\mathbf{H}_i$ is the Hessian of $\ell(y_i, z_i)$ with respect to $z_i = f(x_i,\theta)$. It is important to note that $\mathbf{G}$ is equal to the Hessian of $h(\theta)$ if we replace each $f(x_i,\theta)$ with its local 1st order approximation centered at the current θ:

$$f(x_i,\theta') \approx f(\theta) + \mathbf{J}_i(\theta' - \theta). \tag{3}$$

When $\ell(y, z) = \frac{1}{2}\|y - z\|^2$, we have $\mathbf{H}_i = \mathbf{I}$ and so

$$\mathbf{G} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{J}_i^\top \mathbf{J}_i, \tag{4}$$

which is the matrix used in the well-known Gauss-Newton approach for optimizing nonlinear least squared problems.

Using $\ell(y, f(x,\theta)) = -\log p(y|f(x,\theta))$, the now motivated Gauss-Newton $\mathbf{G} = \mathbf{F}$, the FIM.

$$\mathbf{F} = \mathbb{E}\left[\frac{d\log p(y|x,\theta)}{d\theta}\frac{d\log p(y|x,\theta)}{d\theta}^\top\right] \tag{5}$$

Clearly, $\mathbf{F}$ is symmetric and positive definite, so we revisit linear system solves with (GCR)[3]. Recall, this method solves an SPD $\mathbf{Ax} = \mathbf{b}$ by iteratively finding search directions $\{\mathbf{p}_i\}_{i=1:j}$ so that $\{\mathbf{Ap}_i\}_{i=1:j}$ are orthogonal. To overcome the nonlinearity in the activation functions and the FIM approximation, we do a nonlinear update $r_j = -f(x_j)$, as opposed to the linear GCR update. Substituting $\mathbf{F} = \mathbf{A}$ in GCR, we iterative have the $\mathbf{P}$ and $\mathbf{V}$ basis to minimize the loss function

$$\mathbf{P}_j = [\mathbf{p}_{j_m}, \mathbf{p}_{j_m+1}, \cdots, \mathbf{p}_j] \tag{6}$$
$$\mathbf{V}_j = [\mathbf{F}(x_{j_m})\mathbf{p}_{j_m}, \cdots, \mathbf{F}(x_j)\mathbf{p}_j] \tag{7}$$

The subscript $\mathbf{p}_{j_m}$ is a result of keeping only the m most recent search directions, so as to avoid getting bogged down by old data. Additionally, we notice that the FIM is a function of the data in the current batch. Recall only in expectation does it converge to the true Hessian.

To explicitly compute the FIM, we have two methods - a new FIM from the batch of data and the average over all of the past data points. The first FIM is easy to explain as we get the Fisher vector product with the gradients computed from the data in that batch. We have $\mathbf{g} = \mathbf{Fa}$, where $\mathbf{g}$ is the gradient vector, and $\mathbf{a}$ is the accelerated gradient vector. The history updated Fisher matrix is computed using the Nyström approximation. Since we know that $\mathbf{F} \in \mathbb{R}^{n\times n}$ is symmetric, we can randomly select indices which correspond to the columns of the matrix $\mathbf{F}$, resulting in an $n \times k$ matrix. We then use those same indices to subsample the rows to arrive at a $k \times k$ matrix, where $k \ll n$. This means inverting such a matrix will not only be easier since it is a smaller matrix, but is guaranteed to be SPD as well.

Mathematically, let $\mathcal{I}$ be the set of k indices that were randomly sampled, then the Nyström approximation is

$$\mathbf{F} \approx \mathbf{F}_{:,\mathcal{I}}\mathbf{F}^{-1}_{\mathcal{I},\mathcal{I}}\mathbf{F}^\top_{:,\mathcal{I}} \tag{8}$$
$$= \hat{\mathbf{F}}, \tag{9}$$

where $\mathbf{F}_{:,\mathcal{I}}$ are all of the rows of $\mathbf{F}$, $i = 1, \cdots, n$, and only the columns of $\mathbf{F}$ that are in the randomly selected index set. This is a rank k approximation of the true FIM, but since we are storing many iterations of the Fisher matrix in this averaged history, it is okay to make this approximation.

Another way the FIM is used in our optimizer is as a preconditioner, which is similar to what Martens does[11]. First assume that there is a deep neural network with many linear layers. In each layer, if we assume the gradient of the m inputs and n outputs,$\mathbf{A}$ and $\mathbf{G}$, respectively, are statistically independent, then we can approximate the FIM by

$$\begin{aligned}\mathbf{F} &= \mathbb{E}\left[\text{vec}\left(\mathbf{g}\mathbf{a}^\top\right)\text{vec}\left(\mathbf{g}\mathbf{a}^\top\right)^\top\right] \\ &\approx \mathbf{A}\otimes\mathbf{G}.\end{aligned}$$

Here $\mathbf{F}$ is of size $mn \times mn$, where the weight matrix Θ is $m \times n$. This can be further broken down by using Kronecker products[9], where the inverse of $\mathbf{F}$ is

$$\mathbf{F}^{-1} \approx \mathbf{A}^{-1}\otimes\mathbf{G}^{-1}.$$

Then this inverse can be performed efficiently by being written as

$$(\mathbf{A}^{-1}\otimes\mathbf{G}^{-1})\text{vec}(\mathbf{X}) = \text{vec}(\mathbf{G}^{-1}\mathbf{X}\mathbf{A}^{-\top}).$$

This provides a reasonable approximation to the FIM which we then can apply cheaply in the Fisher gradient matrix-vector multiplication to cluster the eigenvalues of our approximation to the Hessian, speeding up convergence. We use the preconditioner $\tilde{\mathbf{F}}$ to accelerate the linear system solve $\tilde{\mathbf{F}}^{-1}\mathbf{g} = \tilde{\mathbf{F}}^{-1}\mathbf{F}\mathbf{x} \approx \mathbf{x}$. This allows a cheap matrix vector product as opposed to an expensive linear system solve.

Lastly, since we have an SPD matrix, we have found as a heuristic that a few (1-2) iterations of Conjugate Gradient speeds up convergence. However, as CG uses the full dataset, it is not feasible to run more than 2 iterations. This simple method primes the problem and the algorithm, letting it get off on a great first step.

3 EXPERIMENTAL SETUP

The objective of this paper is to compare optimizers on an equal playing field to ensure that nlTGCR has an advantage based on the algorithm and not an unfair metric. Many papers that are showing off their new optimizer will perform some sort of visual task on either multilayer perceptrons (MLPs) or convolutional neural networks (CNNs)[8]. In fact, benchmarking on these vision tasks has become very standard. Schmidt *et. al.* spell out the benchmarking scheme by using MLPs and CNNs on MNIST, Fashion-MNSIT, CIFAR-10, and CIFAR-100[13]. Since we are working on optimizers, a midsized dataset, such as CIFAR-10, was used so that the data is complex enough to not get outstanding classification, but not complex enough that it would take all day training. On CIFAR-10, we have implemented 2 methods for MLP to see how the optimizers scale as the network gets deeper, and a CNN. The task of this data set is given the training batches, correctly categorize the test images into the correctly labeled category.

To tackle this task, the neural network architectures are described below. For the smaller MLP, there are two hidden layers with 90 nodes per layer. The connections are dense with an tanh (nonlinear) activation function. The output layer has 10 nodes as there are 10 categories. This results in 180 internal nodes, so to keep the number of nodes relatively constant, the larger MLP has 5 hidden layers with 30 nodes per layer. Just like before these are dense layers with a tanh activation function. Lastly, the CNN architecture is relatively standard. First, there is a convolutional 2D layer with a 3×3 kernel, followed by a ReLU and maxpooling, which returns a $32 \times 16 \times 16$ output shape. This process - Convolutional 2D, ReLU, maxpooling- is repeated to arrive at a $32 \times 8 \times 8$ output shape. This is then flattened and the output layer is 10 nodes corresponding to the 10 classes in CIFAR-10. These architectures were chosen to be simple models to test the optimizers; high test accuracy wasn't the aim of this study, so the models could be altered to allow for a higehr accuracy.

CIFAR-10 has 60,000 32×32 color images, which means there is an input size of $32 \times 32 \times 3$ to account for the red, green, and blue channels. These 60,000 images are randomly divided into five training batches and one test batch, each with 10,000 images. Each image is labeled into one of 10 categories, such as airplane, bird, dog, horse, ship, *etc.*. The test batch distribution is identical to the training batch distribution as the test batch contains exactly 1,000 randomly-selected images from each class.

The data was loaded and preprocessed using the standard normalization techniques. As opposed to standard normalizers and scalers from well known ML packages, we ran through all of the images in the training dataset, separating the red, green, and blue channels, where we then found the average and standard deviation for each channel, as seen in Tab. 1.

Method	Red	Green	Blue
Mean	0.49139968	0.48215827	0.44653124
SD	0.24703233	0.24348505	0.26158768

Table 1: Computed mean and standard deviation to preprocess the training set of CIFAR-10 with normalization.

Once the preprocessing was performed, one of the architectures was carried out, where important metrics-training loss history per epoch, test accuracy, and time per epoch- were observed. The first two are standard in any classification task, especially when you are performing supervised learning like we are here. The last one is particularly important in our case, as we are comparing optimizers. Since nlTGCR is a second order method, there will be extra computations in approximating the FIM, factorizing the FIM, and applying it to the gradient matrix-vector product. With this extra computation, we use time per epoch to see if the extra computation is worth it compared to the standard first order methods.

4 RESULTS AND DISCUSSION

All three experiments were run using the same architecture and hyperparameters- with constant learning rate $\eta = 0.001$, batch size $|\mathcal{B}| = 128$- over 100 epochs. The results for loss history for each architecture are seen in Fig. 1. We observe significantly lower loss compared to the first-order methods, RMSProp and ADAM, over all neural network architectures. There is a large jump visualized in both of the MLP experiments due to an initial iteration of Conjugate Gradient (CG) to get nlTGCR to a much lower training loss. This is just a heuristic for dense neural networks assuming positive

definiteness. However, this CG step is not always advantageous; it could cause divergence in the method, such as the experiment on CNN's, which is why this large jump isn't seen.

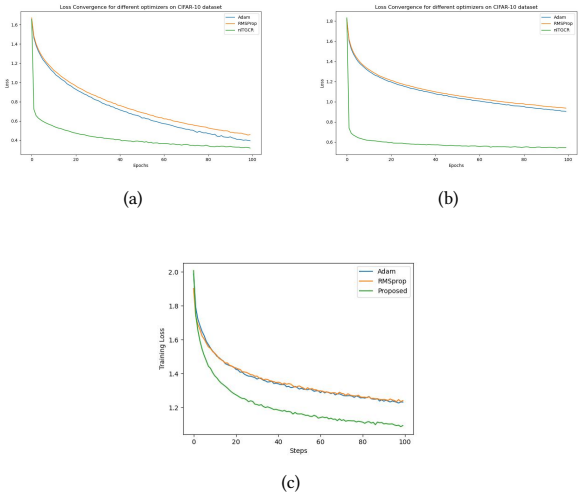

Figure 1: Training Loss History over the 100 epochs for (a) 2-layer MLP (b) 5-layer MLP (c) CNN.

Having a method that converges to a lower training loss isn't advantageous if the model cannot classify the images correctly. Turning our attention to Tab. 2, we can observe that for both of the MLP tests, nlTGCR has a higher test accuracy on the images. This isn't the case for the CNN test, which isn't that suprising, given that nlTGCR was designed with deep neural networks with dense Hessians in mind. The convolutional layers, while reducing the number of parameters for the model, instills a hierarchical matrix structure that nlTGCR doesn't account for, which could explain its performance. However, even though nlTGCR wasn't designed with CNN's in mind, it still performs comparable to the first order methods.

Model	Optimizer	Accuracy (%)
MLP (2-layer)	Adam	51.3
	RMSProp	51.2
	nlTGCR	**54.52**
MLP (5-layer)	Adam	51.7
	RMSProp	52.1
	nlTGCR	**53.85**
CNN (2-layer)	Adam	66.6
	RMSProp	**66.8**
	nlTGCR	65.8

Table 2: Comparing nlTGCR with standard optimizers over different architectures.

Lastly, since there is additional computation with nlTGCR, we need to assure that this lower loss and higher accuracy is worth it. This is assessed by time per epoch. Since all optimizers are run over the entire dataset, with the same batch size, all models have the same number of calls to the optimizer. In our case, all models are coded by hand, not relying on highly optimized packages created by other people. For the 5 layer MLP, we observe the time per epoch data displayed in Tab. 3. RMSProp and ADAM both roughly take 7 seconds; whereas, nlTGCR takes less than half a second. There are many factors to get this result, but the largest contributor is because of the just-in-time (JIT) compiling that was used in nlTGCR. Python is notorously known to be slow since it is interpretable and doesn't have type-casts, but with JIT, Python gets down to C level speeds for large matrix problems.

Optimizer	Time (s)
Adam	7.02
RMSProp	7.10
nlTGCR	**0.42**

Table 3: Comparing time per epoch for for different optimizers on five layer MLP.

This means that nlTGCR outperformed standard first order methods in almost every test at about 17 times faster speeds.

5 CONCLUSIONS

In this paper, we presented nonlinear truncated generalized conjugate residual (nlTGCR). We derived it from the Fisher Information Matrix, and explained approximations to use the Fisher Information as a preconditioner for accelerating the stochastic optimization problem. We tested this against first order methods, such as ADAM and RMSProp. All optimizers were test on neural networks based visual tasks such as image classification on CIFAR-10. The nlTGCR method outperformed both first order methods on training loss, test accuracy, and time per iteration for the MLPs. On CNN, the test accuracy was comparable, but still dominated in training loss and speed.

Future experiments using larger data-sets, such as CIFAR-100, would help illuminate how our optimizer works on data sets of all time. Additionally, for a more thorough project, it would be interesting to compare to other second order methods as well. These methods would include Fish-Leg[4], K-FAC[11], and Shampoo[5], as well as natural gradient methods[10]. This would allow us to position nlTCGR into the specific category on which it optimizes best.

Additionally, to practitioners of ML optimizers, it might be interesting to add a line of experiments comparing the necessity of second-order methods. First order methods potentially break down when parameters are correlated; however, this is normally fixed by drop-out of the correlated features. Testing drop-out + ADAM compared to second order method on a highly correlated dataset such as MovieLens would be interesting to potentitally convert the experimentalist to start using higher order optimization.

ACKNOWLEDGMENTS

MS would like to acknowledge that this work was partially supported by the National Science Foundation under Award Number DMS-2208412.

REFERENCES

[1] Julius R. Blum. 1954. Approximation Methods which Converge with Probability one. 25, 2 (1954), 382–386. https://doi.org/10.1214/aoms/1177728794 Publisher: Institute of Mathematical Statistics.

[2] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. 12 (2011), 2121–2159.

[3] Stanley C. Eisenstat, Howard C. Elman, and Martin H. Schultz. 1983. Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. 20, 2 (1983), 345–357. https://doi.org/10.1137/0720023 Publisher: Society for Industrial and Applied Mathematics.

[4] Jezabel R. Garcia, Federica Freddi, Stathi Fotiadis, Maolin Li, Sattar Vakili, Alberto Bernacchia, and Guillaume Hennequin. 2022. Fisher-Legendre (FishLeg) optimization of deep neural networks. https://openreview.net/forum?id=c9lAOPvQHS

[5] Vineet Gupta, Tomer Koren, and Yoram Singer. 2018. Shampoo: Preconditioned Stochastic Tensor Optimization. https://doi.org/10.48550/arXiv.1802.09568 arXiv:1802.09568

[6] Huan He, Ziyuan Tang, Shifan Zhao, Yousef Saad, and Yuanzhe Xi. 2024. NLTGCR: A class of Nonlinear Acceleration Procedures based on Conjugate Residuals. https://doi.org/10.48550/arXiv.2306.00325 arXiv:2306.00325

[7] Geoffrey Hinton. 2014. Neural Networks for Machine Learning Lecture 6e: rmsprop: Divide the gradient by a running average of its recent magnitude. (2014). https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[8] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/arXiv.1412.6980 arXiv:1412.6980 [cs]

[9] Charles F. Van Loan. 2000. The ubiquitous Kronecker product. 123, 1 (2000), 85–100. https://doi.org/10.1016/S0377-0427(00)00393-9

[10] James Martens. 2020. New insights and perspectives on the natural gradient method. https://doi.org/10.48550/arXiv.1412.1193 arXiv:1412.1193

[11] James Martens and Roger Grosse. 2020. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. https://doi.org/10.48550/arXiv.1503.05671 arXiv:1503.05671

[12] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. 22, 3 (1951), 400–407. https://doi.org/10.1214/aoms/1177729586 Publisher: Institute of Mathematical Statistics.

[13] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. 2021. Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers. https://doi.org/10.48550/arXiv.2007.01547 arXiv:2007.01547 [cs]

Received 20 December 2024