bioRxiv preprint doi: https://doi.org/10.1101/348557; this version posted July 17, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

Posted online June 15, 2018 *bioRxiv, 2018,* ***1–5***

YamAdam: a hyperparameter-free gradient descent optimizer that incorporates unit correction and moment estimation

Kazunori D Yamada [1,2*]

[1]Graduate School of Information Sciences, Tohoku University, Sendai, Japan [2]Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

ABSTRACT

In the deep learning era, gradient descent is the most common method used for optimizing neural network parameters. Among the various mathematical optimization methods, the gradient descent method is the most naive. Adjustment of learning rate is necessary for quick convergence, which is normally done manually with gradient descent. Many optimizers have been developed to control the learning rate and increase convergence speed. Generally, these optimizers adjust the learning rate automatically in response to learning status. These optimizers were gradually improved by incorporating the effective aspects of earlier methods. In this study, we developed a new optimizer: YamAdam. Our optimizer is based on Adam, which utilizes the first and second moments of previous gradients. In addition to the moment estimation system, we incorporated an advantageous part of AdaDelta, namely a unit correction system, into YamAdam. According to benchmark tests on some common datasets, our optimizer showed similar or faster convergent performance compared to the existing methods. YamAdam is an option as an alternative optimizer for deep learning.

INTRODUCTION

There are many iterative mathematical optimization methods, including the gradient descent method, the Newton method, the quasi-Newton method, and the conjugate gradient method. The Newton method is a fast convergent method that utilizes the Newton direction to update parameters. The Newton direction is calculated using the gradient and Hessian of the objective function

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \boldsymbol{H}_t^{-1} \boldsymbol{g}_t, \tag{1}$$

where t stands for time, and $\boldsymbol{\theta}_t$, $\boldsymbol{H}_t$ and $\boldsymbol{g}_t$ stand for parameter vector, Hessian, and gradient vector at time t, respectively. The Newton method converges quadratically and offers faster convergent performance compared to first-order methods like the gradient descent method. The Newton method is able to reach an optimal value within a single iteration if the objective function is expressed by a quadratic equation. However, the Newton method has several disadvantages. For example, convergence may not occur if an initial parameter value is not close to an optimal value. The Hessian must also be positive definite, and the calculation cost for the inverse of the Hessian is large. The quasi-Newton method was developed to solve these problems. This method attempts to approximate a Hessian with an alternative matrix, rather than directly calculating it, and converges superlinearly.

In contrast to these complex methods, the gradient descent method is a more naive mathematical optimization method. Normally, convergence with gradient descent is not fast and is linear. Yet, because implementing the gradient descent method is easy and it exhibits global convergence, it is used frequently with neural networks, including deep learning. The update rule for gradient descent is

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t, \tag{2}$$

where α stands for the learning rate (step size), which decides the magnitude of the parameter update from the previous value to the current value. Adjustment of the learning rate is very important in order to efficiently search for an optimal value with the gradient descent method. If the learning rate is too large, an optimal value cannot be reached because oscillation around the optimal value will occur. If the learning rate is too small, a tremendous amount of time is required to reach the optimal value because the update magnitude is small and many updates are needed. Since the gradient descent method is used frequently, various strategies for improvement have been devised. For example, the stochastic gradient descent (SGD) method attempts to avoid falling into a local minimum by stochastically sampling data and repeatedly updating parameters with the sampled data. In recent deep learning, SGD is the de facto standard for parameter updates. Another strategy for improving gradient descent or SGD is using an optimizer. Many SGD optimizing methods have been developed. These optimizers essentially attempt to control the SGD learning rate. The most naive method is the bisection method, where the learning rate is updated by dividing the previous value by two when the number of updates reaches a predefined value. The Momentum method incorporates a momentum term into the parameter update. The momentum term literally works as a moment and helps update the direction toward the previously updated direction. This method was also further modified to incorporate an

*To whom correspondence should be addressed. Tel: +81 22 795 5856; Email: kyamada@ecei.tohoku.ac.jp

© 2018 The Author(s)
This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.0/uk/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

bioRxiv preprint doi: https://doi.org/10.1101/348557; this version posted July 17, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

acceleration term [1]. AdaGrad is an adaptive method that collects the square of the previous gradient and gradually decreases the learning rate [2].

AdaDelta [3] is a more sophisticated method and is currently one of the best optimizers available, as it it improves on the performance of AdaGrad. The basic idea behind AdaDelta is that it incorporates a unit mismatch correction function between the update values and original parameters. Matching units is naturally ensured with second-order methods like the Newton method, though vanilla SGD, AdaGrad, and other methods fundamentally lack this feature. The update rule of AdaDelta is described by Algorithm 1. AdaDelta no longer has a learning rate.

Algorithm 1 AdaDelta update rule

Require: Objective function $f(\theta)$, Initial parameter θ_0
$\beta=0.95$, $\epsilon=10^{-6}$

1: $v_0=0$, $s_0=0$, $h_0=0$
2: **for** $t=0:t<T$ **do**
3: $g_t \leftarrow \nabla f(\theta_t)$
4: $v_{t+1} \leftarrow \beta v_t+(1-\beta)g_t^2$
5: $s_{t+1} \leftarrow \beta s_t+(1-\beta)h_t^2$
6: $h_{t+1} \leftarrow \frac{\sqrt{s_{t+1}+\epsilon}}{\sqrt{v_{t+1}+\epsilon}}g_t$
7: $\theta_{t+1} \leftarrow \theta_t-h_{t+1}$
8: **end for**
9: **return** θ_T

Here, ϵ is a small value to avoid division by zero and β is a hyperparameter to control parameter updates. In the context of unit correction, the most important variable is s. As shown in line 5, the unit s corresponds to square root of h^2, i.e., the parameter update h. Also, h and θ are of the same unit, thus there is no unit mismatch between the update and parameters.

Adam [4] is another good optimizer that improves on AdaGrad. The update rule of Adam is described by Algorithm 2. Here, ϵ is a small value to avoid division by zero, α is the learning rate, and β_1 and β_2 are the hyperparameters that define the smoothing coefficient of the exponential moving average. The fundamental idea behind Adam is attempting to adjust the learning rate dependent on learning progress. In the update rule, m_t in line 4 and v_t in line 5 approximate the first moment and second moment, namely the mean and variance of the gradient from the first update to time t. By dividing the mean by the variance of the previous gradient information, when the gradient fluctuates near a local minimum, the update size will grow smaller to be able to fall into the minimum. In the opposite situation, the magnitude of updates will be larger, resulting in acceleration of convergence. This idea is similar to that of the Newton method, in which if an alteration of the gradient, namely Hessian, is larger, the magnitude of the update will be smaller. The convergence performance of SGD with Adam is quite fast and Adam is currently one of the most popular optimizers.

As described earlier, optimizer performance has gradually improved through incorporating the features and merits of previously developed optimizers. In this study, we developed a novel optimizer by incorporating a beneficial feature of AdaDelta (the unit correction system) to Adam. In addition, we implemented an adjustment system for the smoothing coefficient for the exponential moving average of update information based on previous update information, resulting in removal of all hyperparameters.

Algorithm 2 Adam update rule

Require: Objective function $f(\theta)$, Initial parameter θ_0
$\alpha=0.001$, $\beta_1=0.9$, $\beta_2=0.999$, $\epsilon=10^{-6}$

1: $m_0=0$, $v_0=0$
2: **for** $t=0:t<T$ **do**
3: $g_t \leftarrow \nabla f(\theta_t)$
4: $m_{t+1} \leftarrow \beta_1 m_t+(1-\beta_1)g_t$
5: $v_{t+1} \leftarrow \beta_2 v_t+(1-\beta_2)g_t^2$
6: $\hat{m} \leftarrow \frac{m_{t+1}}{1-\beta_1^{t+1}}$
7: $\hat{v} \leftarrow \frac{v_{t+1}}{1-\beta_2^{t+1}}$
8: $h=\alpha\frac{\hat{m}}{\sqrt{\hat{v}-\epsilon}}$
9: $\theta_{t+1} \leftarrow \theta_t-h$
10: **end for**
11: **return** θ_T

ALGORITHM

Algorithm 3 shows the update rule for the optimizer we developed: YamAdam (Yamada modified Adam). YamAdam does not have any hyperparameters.

Algorithm 3 YamAdam update rule

Require: Objective function $f(\theta)$, Initial parameter θ_0
Sigmoid function $\phi(x)$, $\epsilon=10^{-6}$

1: $m_0=0$, $v_0=0$, $s_0=0$, $h_0=0$, $\beta_0=0$
2: **for** $t=0:t<T$ **do**
3: $g_t \leftarrow \nabla f(\theta_t)$
4: $m_{t+1} \leftarrow \beta_t m_t+(1-\beta_t)g_t$
5: $v_{t+1} \leftarrow \beta_t v_t+(1-\beta_t)(g_t-m_t)^2$
6: $s_{t+1} \leftarrow \beta_t s_t+(1-\beta_t)h_t^2$
7: $h_{t+1} \leftarrow \frac{\sqrt{s_{t+1}+\epsilon}}{\sqrt{v_{t+1}+\epsilon}}m_{t+1}$
8: $\beta_{t+1} \leftarrow \phi\left(\frac{\|h_t\|_1+\epsilon}{\|h_{t+1}\|_1+\epsilon}\right)-\epsilon$
9: $\theta_{t+1} \leftarrow \theta_t-h_{t+1}$
10: **end for**
11: **return** θ_T

In the algorithm, the first and second moment information is stored as an exponential moving averages with the smoothing coefficient, β_t in line 4 and 5. These values are estimations of the mean and variance of the gradient, respectively. In line 6, the square of the update value is stored in a similar manner as above. A novel update value is calculated using these values in the equation in line 7. The smoothing coefficient of the exponential moving average is then calculated in line 8 based on the current and previous update values. If the next update size is smaller, β_t will be larger, causing the following gradient information to be considered less important. Conversely, if the next update size is larger, the following gradient information

bioRxiv preprint doi: https://doi.org/10.1101/348557; this version posted July 17, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

will be regarded as more important. In this system, the smoothing coefficient acts as momentum, where the influence of current gradient information on the accumulated gradient information will change depending on update intensity. Lastly, the parameter vector is updated by the equation in line 9.

SIMULATION EXPERIMENTS

To benchmark the performance of our optimizer, we conducted two simulation experiments in which the convergent performance with logistic regression and four layer multilayer perceptron (MLP) were tested on various datasets.

Methods

Benchmark datasets As benchmarks, we utilized three datasets: the Modified National Institute of Standards and Technology (MNIST) dataset [5], the Letter Recognition (LR) dataset, and the Sensorless Drive Diagnosis (SDD) dataset [6]. The number of attributes of each instance on the datasets is 784, 16 and 48, the dimension of output vector is 10, 26 and 11 and the number of instances is 70,000, 20,000 and 58,509 respectively. From the datasets, first 60,000, 15,000 and 40,000 instances were used as training datasets, and the remaining 10,000, 5,000 and 18,509 instances were used as validation datasets, respectively.

Network architectures For the simulation, we designed a logistic regression model and a four-layer MLP, consisting of an input layer, two middle layers, and an output layer. The unit number for the middle layers was set to 500. For both architectures, output vectors were the outputs of the softmax function and the models were for multi-class classification. We used ReLU as the activation function for middle layers and suitable optimized initial parameters [7] were used. We used Theano version 0.8.2 (University of Montreal) with Python version 3.5.2 as the framework for implementing the architectures.

Parameters of compared optimizers For vanilla SGD and AdaGrad, the learning rate α was set to 0.01. For AdaDelta, the smoothing coefficient for the exponential moving average β was set to 0.95. For Adam, the learning rate α, the smoothing coefficients for the exponential moving average β_1 and β_2 were set to 0.001, 0.9, and 0.999 respectively.

Benchmark procedure For each method, the training losses for epochs 1 to 50 were recorded 10 times while randomly changing the initial parameters. The mean training loss values of the 10 trials were plotted as lines on a scatter plot.

Computational environment The benchmarks were conducted on an Intel(R) Xeon(R) CPU E5-2680 v2 @2.80 GHz with 64GB RAM with a Tesla K20m (NVIDIA) as part of the NIG supercomputer at ROIS National Institute of Genetics in Japan.

Results and discussion

Logistic regression As shown in Figure 1, we conducted a performance benchmark with five methods: vanilla SGD (no optimizer), AdaGrad, AdaDelta, Adam, and YamAdam on the MNIST (a), LR (b) and SDD (c) datasets with logistic regression. In the results for all datasets, Adam and AdaDelta exhibited rapid convergence compared to the other existing methods. These results were expected based on previous reports [3, 4]. YamAdam also exhibited similar or better convergent performance than Adam, and similar or slightly better performance than AdaDelta. The difference in performance was most obvious with the SDD dataset, in which YamAdam was best and AdaDelta was second best. YamAdam incorporates the unit correction system utilized in AdaDelta, which may positively affect performance on the SDD dataset. Problem difficulty may be a factor

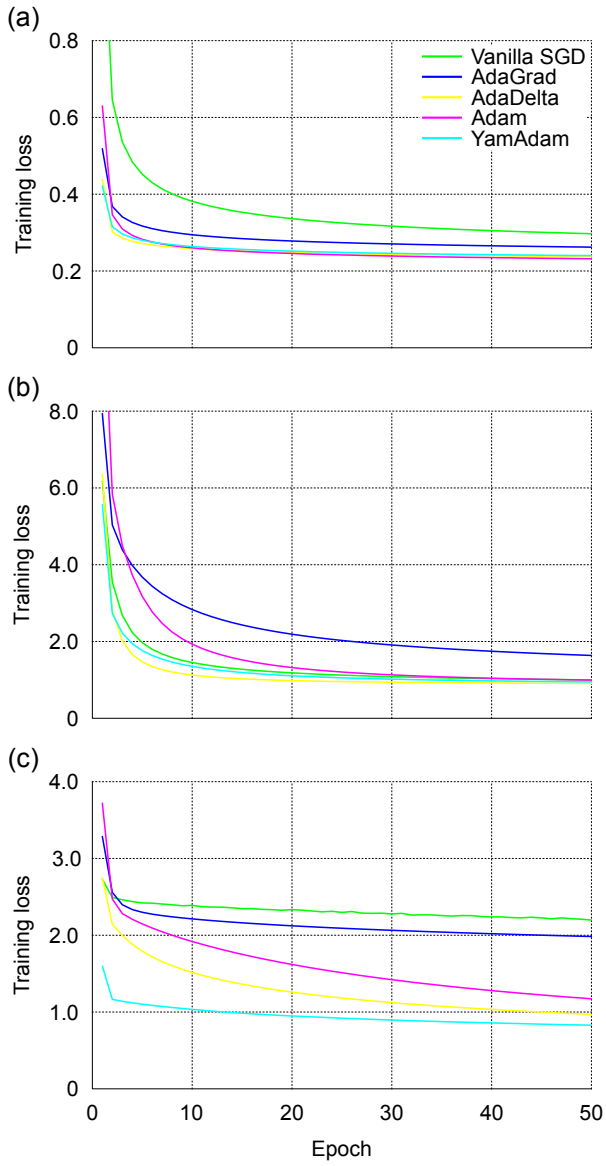

Figure 1. Comparison of optimizers on logistic regression. The performance of each optimizer was benchmarked on (a) MNIST, (b) LR, and (c) SDD datasets.

bioRxiv preprint doi: https://doi.org/10.1101/348557; this version posted July 17, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

contributing to the superior performance of methods that have the unit correction system. MNIST is a relatively easy dataset compared to the other two datasets. The accuracies for each method with MNIST were greater than 0.9, even for the normal logistic regression architecture. On the other hand, the accuracies with the SDD dataset were 0.30 – 0.75, suggesting that the SDD dataset was more difficult than MNIST. Although there are many factors that contribute to dataset difficulty, unit mismatching of each attribute in an instance of the dataset is a possible factor. For example, in the first instance in SDD, the value of the first element is -6.4707×10^{-6} and the 38th is 40.409. Difference in value magnitude, namely unit mismatch, is mostly unique to the SDD dataset relative to the other datasets. This may be why YamAdam performed better on SDD. When we benchmarked the methods on a standardized SDD dataset, there was reduced difference in performance.

Four-layer MLP As shown in Figure 2, we conducted similar experiments with a four-layer MLP. Results showed no qualitative differences compared to the logistic regression results for performance with each method, even though more layers were stacked and the parameters were increased in the architecture.

CONCLUSION

We developed YamAdam, a novel optimizer. YamAdam is a hyperparameter-free optimizer that includes a unit correction system (derived from AdaDelta) and moment estimation (derived from Adam). Our method exhibited similar or faster convergent performance compared to the existing optimizers, especially in computations on a unit-mismatched dataset. Many different kinds of datasets exist, and our optimizer is another option for use in deep learning.

ADDITIONAL INFORMATION

Acknowledgements

Computations were partially performed on the NIG supercomputer at ROIS National Institute of Genetics.

Funding

This work was supported in part by the Top Global University Project from the Ministry of Education, Culture, Sports, Science and Technology of Japan (MEXT), KAKENHI from the Japan Society for the Promotion of Science (JSPS) under Grant Number 18K18143.

Availability of data and material

The YamAdam source code is available on GitHub (git@github.com:yamada-kd/YamAdam.git).

Abbreviations

LR: Letter Recognition; MLP: multilayer perceptron; MNIST: Modified National Institute of Standards and Technology; ReLU: rectified linear unit; SDD: Sensorless Drive Diagnosis; SGD: stochastic gradient descent

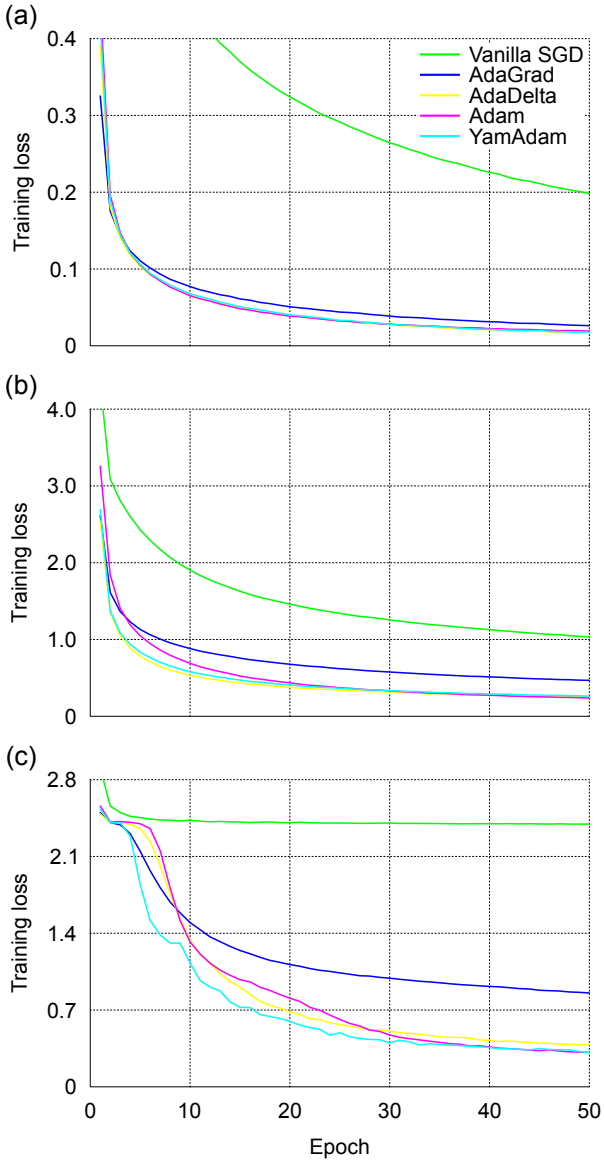

Figure 2. Comparison of optimizers on MLP. The performance of each optimizer was benchmarked on (a) MNIST, (b) LR, and (c) SDD datasets.

Competing interests

The authors declare that they have no competing interests.

REFERENCES

1. Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.
2. John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
3. Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
4. Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

bioRxiv preprint doi: https://doi.org/10.1101/348557; this version posted July 17, 2018. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-ND 4.0 International license.

5. Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.
6. Dheeru Dua and Efi Karra Taniskidou. Uci machine learning repository. 2017.
7. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.