Differentiable Self-Adaptive Learning Rate

Bozhou Chen, Hongzhi Wang* Member, IEEE, Chenmin Ba

Abstract—Learning rate adaptation is a popular topic in machine learning. Gradient Descent trains neural network with a fixed learning rate. Learning rate adaptation is proposed to accelerate the training process through adjusting the step size in the training session. Famous works include Momentum, Adam and Hypergradient. Hypergradient is the most special one. Hypergradient achieved adaptation by calculating the derivative of learning rate with respect to cost function and utilizing gradient descent for learning rate. However, Hypergradient is still not perfect. In practice, Hypergradient fail to decrease training loss after learning rate adaptation with a large probability. Apart from that, evidence has been found that Hypergradient are not suitable for dealing with large datasets in the form of minibatch training. Most unfortunately, Hypergradient always fails to get a good accuracy on the validation dataset although it could reduce training loss to a very tiny value. To solve Hypergradient’s problems, we propose a novel adaptation algorithm, where learning rate is parameter specific and internal structured. We conduct extensive experiments on multiple network models and datasets compared with various benchmark optimizers. It is shown that our algorithm can achieve faster and higher qualified convergence than those state-of-art optimizers.

Index Terms—Learning rate, adaptation, parameter-specific, internal structured.

1 INTRODUCTION

Learning rate is the core of an optimizer and a key hyper-parameter in artificial neural networks. Related researches can be classified as three categories, including basic gradient descent, adaptation algorithms and hypergradient based algorithms.

The early GD(gradient descent) algorithm [1] fixes the learning rate in the whole training session and utilizes the first-order derivative of parameters. Three implementations are SGD(stochastic gradient descent) [2], minibatch gradient descent and batch gradient descent. GD is able to fit a neural network on a dataset as long as with sufficient time. However, trade-off between convergence speed and model performance always exists and they cannot be achieved in the same time. Cause a large learning rate could accelerate training but could not reach the best performance [3]. While a small learning rate is just the opposite. Apart from that, minibatch training and batch training are essentially non-convex optimization, where GD usually stucks into a local minima.

Adaptation methods are taken to avoid GD’s fakes. Momentum [4] takes the exponential moving average of historical gradient as step size. History gradient can help run out of the local minima. However, the trade-off between convergence speed and performance still exists. AdaGrad [5] subtly scales the step size by multiplying learning rate with the reciprocal of two-norm of historical gradients to have step size reduce along with training session. While AdaGrad is interfered a lot by historical gradient. Because step size keeps decreasing along with training session resulting that it converges very sluggishly in the later stage. Actually, step size in AdaGrad decreases purposelessly. RMSProp [6] replaces two-norm in AdaGrad with exponential moving average of historical gradients to reduce the interference, which can be regarded as an extension of GD and AdaGrad. Adam [7] combines the best properties of Momentum and RMSProp, so that it can solve both the two fakes of GD surprisingly. However, Adam is not sensitive enough, especially when step size are needed to increase. Additionally, evidence shows that Adam always performs unsatisfactory in the late stage of a training session. When fitting a large datasets with minibatch training, a popular proposal is Momentum and change learning rate with a scheduler manually but not Adam to get stronger model.

HD(hypergradient descent) implements adaptation by calculating the derivative of learning rate with respect to cost function and utilizing gradient descent for learning rate. So HD knows exactly the need of cost function for learning rate, i.e., when to increase and when to decrease. However, case where HD misunderstands usually exists, i.e., training loss increases unexpectedly after HD’s adaptation. Additionally, evidence can be found to show that HD are not suitable for minibatch training tasks. In other word, it’s a hard problem for HD to process large datasets. Although HD gained a good training loss on published datasets, the performance on the validation dataset are nearly of no practical value.

In this paper, we propose a novel optimizer DSA(differentiable self-adaptive learning rate), which is an extension and improvement of HD and SGD. The core of DSA is the implementation of learning rate. Actually, the derivative of learning rate in HD is computed by an inefficient approximation. In DSA, the approximation to gradient is replaced with an effective detection to decrease HD’s misunderstanding. Besides, many other tricks are adopted to enhance the proposed algorithm. For example, learning rate has been internal structured and is specific for each parameter.

Contributions of DSA are summarized as follows.

• Proposed DSA could miss up HD’s inherent shortcoming and we give out a complete program to deal with large
2 Method

In this section, we will illustrate DSA (differentiable self-adaptive) algorithm in detail. Firstly, the basic algorithm HD (Hypergradient Descent) will be introduced as a fore-shadowing for DSA in Section 2.1 and analysed about the practical pitfalls in Section 2.2. Finally, we propose our algorithm DSA in Section 2.3.

2.1 Hypergradient Descent

This section is dedicated to reviewing Hypergradient Descent [8]. Basically, the gradient of learning rate with respect to cost function is gained through the parameter updated from the last iteration. The updating method of Gradient Descent is as

\[ W_t = W_{t-1} - \alpha \nabla f(W_{t-1}), \quad (1) \]

where \( \alpha \) is the learning rate and \( \nabla f(W_{t-1}) \) is the gradient of previous parameter \( W_{t-1} \). Obviously, \( W_t \) can be regarded as a function of \( \alpha \). HD makes the assumption that the optimal value of \( \alpha \) does not change much between two consecutive iterations so that it can use the update rule for the previous step to optimize \( \alpha \) in the current one. In other words, HD uses \( \partial f(W_{t-1})/\partial \alpha \) to approximate \( \partial f(W_{t-1})/\partial \alpha \). For this, we will compute \( \partial f(W_{t-1})/\partial \alpha \). Noting that \( W_{t-1} = W_{t-2} - \alpha \nabla f(W_{t-2}) \), and applying the chain rule, we can get

\[
\frac{\partial f(W_{t-1})}{\partial \alpha} = \nabla f(W_{t-1}) \cdot \frac{\partial (W_{t-2} - \alpha \nabla f(W_{t-2}))}{\partial \alpha} = \nabla f(W_{t-1}) \cdot (-\nabla f(W_{t-2})),
\]

which allows us to compute the needed hypergradient with a simple dot product and the memory cost of only one extra copy of the original gradient. With this hypergradient, we construct a update rule for the learning rate as

\[
\alpha_t = \alpha_{t-1} - \beta \frac{\partial f(W_{t-1})}{\partial \alpha} = \alpha_{t-1} + \beta \nabla f(W_{t-1}) \cdot \nabla f(W_{t-2})
\]

introducing \( \beta \) as \( \alpha \)'s step size. Considering the sequence \( \alpha_t \), Formula [4] can be modified to

\[ W_t = W_{t-1} - \alpha_t \nabla f(W_{t-1}). \quad (4) \]

2.2 Analyses about Hypergradient Descent

In this section, we will take more in-depth analysis for Hypergradient Descent, including the sensitivity of HD, why HD may misunderstand the cost function and other problems mentioned in Section 2.

Sensitivity of HD

HD is more sensitive than other traditional algorithms because learning rate in HD can increase according to occasions but not just decrease. Take a simple convex optimization \( \arg \min_{x} x^2 \) as example which is visualized in Fig. 1. Optimization starts from \( x_0 \), and \( x \) steps to \( x_1 \) with the initial learning rate \( \alpha_1 \). Suppose \( x_1 \) is still far from the extreme point, i.e., \( |x_1 - x_{opt}| \gg \alpha_1 \). In the second iteration, the step size of learning rate is \( \beta \nabla x_1^2 \cdot \nabla x_0^2 > 0 \). Therefore, learning rate will increase and the step size of \( x \) will be larger than that of occasions without HD. Actually, we could find a more intuitive explanation for this behavior of HD. In each current iteration, if the step direction indicated by parameter’s gradient is the same to that of last iteration, it can be understood as the current position is on a smooth descending surface, where speeding up is a better policy to approaching the extreme point.

Shortcoming of HD

In the last paragraph, we learned why HD can control learning rate sensitively. However, the explanation is also the reason why HD usually misunderstands the cost function. In the example of last paragraph, we have a key assumption that \( x_1 \) is still far from the extreme point, which ensures that increasing the learning rate will not cause the extreme point to be crossed after one step. While unfortunately, occasions like Fig. 2 always exist. As shown in the figure, \( x_1 \) is already very close to the extreme opint after the first iteration. While according to HD’s updating rule for learning rate, step size increases to a larger value resulting the miss of HD, i.e., \( x_2 > x_2^2 \). We have taken a small experiment to calculate the frequency of HD’s misunderstanding. From the second iteration, we compare \( L = f(W_{t-1} - \alpha_{t-1} \nabla f(W_{t-1})) \) and \( L_{adapt} = f(W_{t-1} - \alpha_t \nabla f(W_{t-1})) \), where \( f(W) \) is the cost function or the loss function, the smaller the better. \( L \) is the loss if we don’t apply HD to take adaptation for learning rate, while \( L_{adapt} \) is the opposite. Obviously, \( L_{adapt} \leq L \) is
as our expectation. If $L_{adapt} > L$, we think one misunderstanding happens and count as a miss. When the training session is over, we can calculate the misunderstanding frequency as $\frac{M}{(T-1)}$, where $M$ is the count of miss and $T$ is the total iterations. We apply HD to train a simple MLP on four different feature datasets and record the average miss rate in a complete training session as shown in TABLE 1. We can see that nearly one-fourth adaptation is a failure when dealing with a somewhat complex dataset.

**TABLE 1: Miss rate of HD on four different feature datasets**

| DataSets | IRIS | WINE | CAR | AGARICUS |
|----------|------|------|-----|----------|
| MissRate | 0.057 | 0.228 | 0.061 | 0.029 |

When dealing with minibatch training tasks, the miss rate of HD will be larger. Because the assumption that using $\frac{\partial f(W_t)}{\partial \alpha}$ to approximate $\frac{\partial f(W_t)}{\partial \alpha}$ requires to remain unchanged between two consecutive iterations. But cost function is different between two consecutive iterations, cause training samples are changing with iterations in minibatch training. In each current iteration, if the step direction of parameter on current distribution is the same to that on the last iteration’s distribution, learning rate will increase. This is absolutely ridiculous. We can use an effective detection to solve this problem, which will be discussed in Section 2.3. Here, in order to make readers better understand why hypergradient based learning rate adaptation method is not suitable for minibatch training, we first make an assumption. Assuming that every time HD reaches a new surface, it can decide whether learning rate should increase or decrease at the current position immediately, regardless of the change of cost function. That is, if the current position is far from the extreme point, the learning rate increases, and vice versa. Then let’s take a simple regression task as evidence to illustrate the idea. The target is to fit two points $(1, 2)$, $(1, 3)$ with line $y = w \cdot x$, i.e., $\min \sum (y_i - w \cdot x_i)^2$.

We fit the line in the form of minibatch, i.e., the first batch contains one point $(1, 2)$ and the second contains another point. Iterations are visualized in Fig. 3, noting that we substitute parameter $w$ with $x$ in figures. Obviously, the optimal resolution is $x_{opt}$, However, HD will never achieve that point. Because in each iteration, HD try to increase step size to make the current cost function as smaller as possible. As a result, $|x_0 - x_{opt}| < |x_1 - x_{opt}| < |x_2 - x_{opt}| < |x_3 - x_{opt}|$ and HD gets into a trap. Learning rate increases blindly, $x$ is more and more close to the local minima after each iteration but not the global optimal.

Although HD could decrease the training loss to a quite low scale, the performance of the trained model is actually ordinary. TABLE 2 tells us the miss rate when processing large datasets in minibatch training, where nearly half of learning rate’s adaptation is a miss. Fig. 4 indicates that although HD may get a good train loss such as SVHN, the trained neural network still fail to get a good accuracy on validation dataset.

**TABLE 2: Miss rate of HD with minibatch**

| DataSets | MNIST | SVHN | CIFAR10 | CIFAR100 |
|----------|-------|------|---------|----------|
| MissRate | 0.4909 | 0.4946 | 0.4970 | 0.4977 |
2.3 Differentiable Self-Adaptive Learning Rate

In this section, we will propose our algorithm DSA while trying to solve HD’s pitfalls. Totally, Section 2.3.1 aims to solve HD’s misunderstanding and Section 2.3.2 is against the problem in minibatch training. In Section 2.3.3 techniques to further improve DSA will be illustrated in detail. Finally, pseudo code of DSA will be drawn in Section 2.3.4.

2.3.1 Detection rather than Experience

As talked in Section 2.1, HD approximates $\partial f(W_t)/\partial \alpha$ with $\partial f(W_{t-1})/\partial \alpha$. This approximation takes use of the experience in the last iteration to decide the change for learning rate. While DSA takes detection for the future distribution rather than empirical adaptation to avoid HD’s misunderstanding. In DSA, we introduce an internal variable $\tilde{W}$

$$\tilde{W} = W_{t-1} - \alpha \cdot \nabla f(W_{t-1}), \quad (5)$$

which is a function of $\alpha$. Nextly, we try to minimize $f(\tilde{W}(\alpha))$ by optimizing $\alpha$ through

$$\frac{\partial f(\tilde{W})}{\partial \alpha} = \nabla f(\tilde{W}) \cdot \frac{\partial (W_{t-1} - \alpha \nabla f(W_{t-1}))}{\partial \alpha} \quad \text{(6)}$$

Therefore, update rule for learning rate will be modified to

$$\alpha_t = \alpha_{t-1} - \frac{\partial f(W)}{\partial f(\tilde{W})} \cdot \nabla f(W_{t-1}).$$

While the parameter’s step stays the same

$$W_t = W_{t-1} - \alpha_t \cdot \nabla f(W_{t-1}).$$

The cost is $f(\tilde{W}(\alpha_t))$ after $t$-th iteration. If we don’t apply DSA adaptation for learning rate, the cost will be $f(W(\alpha_{t-1}))$. And it’s surely that $f(\tilde{W}(\alpha_t))$ is smaller than $f(W(\alpha_{t-1}))$ because DSA adaptation for learning rate is essentially a gradient descent for $f(\tilde{W}(\alpha))$ with respect to $\alpha$.

![Fig. 5: Update rule for learning rate in DSA](image)

The most important difference from HD is the detection with current learning rate, i.e., $\tilde{W}$. If $\tilde{W}$ crosses the extreme point, the learning rate should decrease. Otherwise, the learning rate will increase. In other words, HD makes decision according to the history while DSA according to the future. We visualize the update rule of learning rate in Fig. 5. Parameter $W$ steps from the start point S, reaching and taking detection at R. P is the detected point with $\alpha_{t-1}$, i.e., the position of $\tilde{W}$. If gradient at P has the same direction to R, i.e., $\nabla f(W) \cdot \nabla f(W) > 0$, learning rate will increase. Otherwise it decreases. Obviously, DSA can avoid HD’s misunderstanding perfectly.

2.3.2 Dealing with Minibatch Training

Another pitfall of HD is the ability of processing large datasets using minibatch training. While DSA will get into the trap more easily than HD as shown in Fig. 3. Because DSA can fit a convex function far more quickly than HD and other optimizers. In other word, the ideal occasion for DSA is that only one cost function exists in the training, i.e., batch training. Fortunately, we can use DSA to train a neural network in the form of batch training, where the network is pretrained with Momentum optimizer on minibatch data. Evidence told us this kind of further training using DSA could always enhance the network to a stronger one and we will show this in the experiment.

2.3.3 Further Improvements for DSA

In this part, we will give three techniques to make DSA more sensitive, efficient and safe. Skills include parameter specific learning rate, internal structured learning rate and step size equal learning rate.

**Parameter Specific Learning Rate**

In normal machine learning algorithm, learning rate is always a scalar, while parameters’ requirements for learning rate is various on the scale. Therefore, we try to make it a vector and specific for every parameter. The proposed paper of HD has denoted that It is straightforward to generalize the introduced method to the case where $\alpha$ is a vector of per-parameter learning rates [8]. However, truth obeys wishes. Actually, the gradient for $\alpha$ is a result of two vectors’ inner product

$$\nabla f(\tilde{W}) \cdot \nabla f(W_{t-1}) = \sum_i \nabla f(\tilde{W}_i) \nabla f(W_{t-1}_i).$$

If we directly generalize the update rule to the form of parameter specific, it would be

$$\nabla f(\alpha)_i = -\nabla f(\tilde{W})_i \nabla f(W_{t-1}_i).$$

However, the famous problem grad loss will be more significant in formula 10. The scale of $\nabla f(\tilde{W})_i \nabla f(W_{t-1}_i)$ is always near $10^{-6}$ and even smaller, which makes learning rate adaptation stay in name only. Given that, we use learning rate’s gradient as an indicator. That is, $\alpha$ will increase a step of $\beta$ when the gradient is positive. The update rule of learning rate turns into

$$\alpha_i^{(t)} = \alpha_i^{(t-1)} + \beta \frac{\nabla f(\tilde{W})_i \nabla f(W_{t-1})_i}{\nabla f(W)_i \nabla f(W_{t-1})_i + \epsilon}$$

where $\nabla f(\tilde{W})_i \nabla f(W_{t-1}_i)/(\nabla f(W)_i \nabla f(W_{t-1})_i + \epsilon) \in \{-1, 1, 0\}$ and $\epsilon$ is a infinitesimal.

**Internal Structured Learning Rate**

The formula 11 is not safe, because $\alpha$ is possible to be negative or oversized. Here we endow learning rate an internal structure $\sigma(\alpha) \gamma$, where $\gamma = 0.1$ and $\sigma$ is sigmoid activation

$$\sigma(\alpha) = \frac{1}{1 + \exp^{-\alpha}}.$$
As a result, learning rate \( \sigma(\alpha) \cdot \gamma \) is in the range of \((0, 0.1)\). Straightforward to derive that the update rule for \( \alpha \) is still formula 11. And the step of \( W \) will be

\[
W^{(t)} = W^{(t-1)} - \frac{\gamma}{1 + \exp(-\alpha^{(t)})} \cdot \nabla f(W^{(t-1)}). \tag{12}
\]

### 3.1 Experiment Setting

#### Neural Networks and Datasets

We applied DSA to ResNet [9], FMP [10], DNN [11], [12] and MLP [13]. Famous ResNet takes deep residual learning and here we apply ResNet-18. DNN is the common deep convolution neural network. The core of FMP is the fractional maxpool layer and it’s visualized as Fig. 6 supposing the input is a image of \(28 \times 28\). FMP is composed of 6 convolution block and 1 linear block. Each convolution block ends up with a fractional maxpool. Each output of a convolution layer is processed by prelu activation.

#### TABLE 3: Large dataset information

| Dataset   | #Train / #Test | #Attributes | #Class |
|-----------|----------------|-------------|--------|
| MNIST     | 60,000/10,000  | 1’28’28     | 10     |
| SVHN      | 73,257/26,032  | 3’32’32     | 10     |
| CIFAR10   | 50,000/10,000  | 3’32’32     | 10     |
| CIFAR100  | 50,000/10,000  | 3’32’32     | 100    |

#### TABLE 4: Small dataset information

| Dataset   | #Train / #Test | #Attributes | #Class |
|-----------|----------------|-------------|--------|
| IRIS      | 120/30         | 4           | 3      |
| WINE      | 142/36         | 13          | 3      |
| CAR       | 1,382/346      | 6           | 4      |
| AGARICUS  | 6,499/1,625    | 116         | 2      |

#### Baseline

Selected optimizers include SGD [1], Momentum [4], AdaGrad [5], AdamDelta [17], RMSProp [6], Adam [7], AdamW [18] and Hypergradient Descent [8]. Note that, Hypergradient Descent has mutiple versions and we choose the one based on SGD here.

#### Metrics

We recorded accuracy, F1-score, recall and precision on validation dataset to measure the effect of different optimizers from different perspectives [19]. Additionally, the train loss is recorded to demonstrate the convergence effect and miss rate is used to compare DSA and HD.

#### Implementations

The structure of FMP is visualized in Fig. 6 and FMP is designed with reference to [20], [21], which does very well on MNIST and SVHN. FMP has a huge mount of parameters, which is a challenge to DSA. DNN is designed with reference to [11], [12], [22] and we set channel size as a smaller value to simplify the model so that it can converge faster than the original. MLP sequences 5 fully connected layers. The dimensions of each layer are: 32, 64, 256 and 128. The output of the first and third layers are processed using sigmoid activation. The output of the second and the fourth layers are processed by prelu activation [23]. The features output by the neural network are processed by \( \log \_softmax \) activation and cross-entropy loss function.

1. https://archive.ics.uci.edu/ml/datasets
The number of training epochs on large datasets is 200 and that on small datasets is commonly 1000. Actually, DSA will take one more loop than other adaptation optimizers in each iteration, so we always tend to use fewer epochs when applying DSA. DSA’s experiments on large datasets is based on pretrained model, the pretraining is 180 iterations and the formal training of DSA is 10 epochs. The batchsize of MNIST, SVHN, CIFAR10, CIFAR100 is 128, 64, 128, 32 respectively. Large datasets are preprocessed with some efficient skills including random crop for images, random horizontal flip for images and normalization for each image. When training convolution neural network with SGD, Momentum and Adam, we apply a learning rate scheduler where learning rate will be scaled $\gamma$ times than the current in the half and three quarters of the training session. Commonly, $\gamma = 0.1$. Pretrained ResNet is obtained from the model trained by Momentum cooperated with learning rate scheduler. Pretraining stops at some iteration after the second learning rate scaling. While if neural network is DNN or FMP, we will apply Adamax as optimizer to pretrain the model. Because there are significant grad loss in DNN and huge amount of parameters in FMP, which are both hard problems for Momentum. If we use SGD or Momentum to pretrain DNN and FMP, ten thousands of iterations would be needed. Adamax is practically the most superior baseline here. SGD’s initial learning rate is 0.1. Momentum’s initial learning rate is 0.1 and the momentum rate is 0.9. AdaGrad’s initial learning rate is 0.01. RMSProp’s initial learning rate is 0.01. Adam and AdamW’s initial learning rate is 0.001. Adamax’s initial learning rate is 0.002. HD’s initial learning rate is 0.1 and step size of learning rate is 0.01. DSA’s step size of $\alpha$ is 0.1 and initial learning rate of DSA is 0.001, i.e., initial $\alpha$ is -4.6. When dealing with large image datasets on pretrained model, the initial learning rate of DSA is $10^{-5}$ and the step size of $\alpha$ is 0.3. Above settings for hyper-parameters refer to the proposal [24]. Our experiments are conducted on GTX 3060Ti GPU for all the groups.

### 3.2 Experimental Results

In this section, we show the performances of DSA in different occasions. Experiments in this section consist of two groups totally. Firstly, we apply DSA to fit convolution neural networks. ResNet is trained with all selected image datasets, while FMP and DNN are just evaluated on MNIST and SVHN because the performances are limited on the other two datasets. And the baselines used on FMP and DNN are ADAM family optimizers, because other optimizers would take ten thousands of iterations to train these two networks. The second group is a series of experiments training MLP on small feature datasets.

#### Experiments with Image Datasets

Results on ResNet have been collected in TABLE 6 and TABLE 8. TABLE 8 and TABLE 6. Results on DNN and FMP are shown in TABLE 9 and TABLE 11 and TABLE 12. In tables, the baseline with $\ast$ is the pretrained. DSA is trained for only 10 epochs after the pretraining, but it can get the peak of performance rapidly. That is, DSA can always...
enhance the pretrained model by $0.1\% \sim 1\%$ on accuracy. Interestingly, we find that the maximum of DSA’s metric is usually smaller than that of pretrained baseline. For example, the precision’s maximum of DSA on ResNet with SVHN is 97.70 and that of Momentum(pretrained baseline) is 97.80, i.e., DSA is smaller than Momentum. While the precision’s minimum of DSA is 94.66 and that of Momentum is 94.07, i.e., DSA is larger than Momentum. And the accuracy of DSA is $0.3\%$ larger than that of Momentum. In fact, the trained model will favor the latest trained minibatch data more than others. So the neural network always do better on some part of a dataset. While in the formal training of DSA, which is in the form of batch training, DSA treated all training samples fairly. So the results of DSA is surely to be better totally. While such a training mode is only suitable for DSA, because only DSA can achieve fast adaptation for step size of model’s parameters.

**Experiments with Feature Datasets**

In this section, we display the performance of DSA when processing MLP on feature datasets. As described in experiment settings, MLP is trained for 1000 epochs in a single session. While we do not show the result after all the epochs

---

**TABLE 13: Train MLP with WINE (1000 Epochs)**

|                | ACCU | F1-SCORE | RECALL | PRECISION |
|----------------|------|----------|--------|-----------|
| SGD            | 44.44| 0.0~61.54| 0.0~44.44| 0.0~100.0 |
| Momentum       | 44.44| 0.0~61.54| 0.0~44.44| 0.0~100.0 |
| RMSProp        | 98.28| 97.30~100.0| 96.88~100.0| 94.74~100.0 |
| AdaDelta       | 86.21| 55.56~97.30| 50.00~100.0| 62.50~94.74 |
| AdaGrad        | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |
| Adam           | 97.22| 92.31~100.0| 85.71~100.0| 93.75~100.0 |
| AdamW          | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |
| Adamax         | 98.28| 94.12~100.0| 88.89~100.0| 96.77~100.0 |
| DSA            | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |

---

**TABLE 15: Train MLP with IRIS (30 Epochs)**

|                | ACCU | F1-SCORE | RECALL | PRECISION |
|----------------|------|----------|--------|-----------|
| SGD            | 43.33| 0.0~60.47| 0.0~43.33| 0.0~100.0 |
| Momentum       | 36.67| 0.0~53.66| 0.0~36.67| 0.0~100.0 |
| RMSProp        | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |
| AdaDelta       | 56.67| 0.0~100.0| 0.0~100.0| 0.0~100.0 |
| AdaGrad        | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |
| Adam           | 60.00| 14.29~100.0| 33.33~100.0| 7.69~100.0 |
| AdamW          | 56.67| 0.0~100.0| 0.0~100.0| 0.0~100.0 |
| Adamax         | 56.67| 0.0~100.0| 0.0~100.0| 0.0~100.0 |
| HD             | 20.00| 0.0~33.33| 0.0~20.00| 0.0~100.0 |
| DSA            | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |

---

**TABLE 14: Train MLP with CAR (500 Epochs)**

|                | ACCU | F1-SCORE | RECALL | PRECISION |
|----------------|------|----------|--------|-----------|
| SGD            | 69.36| 0.0~81.91| 0.0~69.36| 0.0~100.0 |
| Momentum       | 93.06| 62.50~97.49| 61.54~97.90| 58.82~97.08 |
| RMSProp        | 90.75| 44.44~96.60| 42.11~98.70| 35.29~94.58 |
| AdaDelta       | 88.44| 0.0~97.26| 0.0~98.30| 0.0~96.25 |
| AdaGrad        | 91.62| 60.00~98.46| 53.85~95.88| 52.94~97.08 |
| Adam           | 99.42| 98.75~100.0| 97.53~100.0| 99.17~100.0 |
| AdamW          | 99.42| 94.74~99.79| 94.44~100.0| 90.00~100.0 |
| Adamax         | 97.11| 75.00~95.88| 64.29~100.0| 70.59~99.17 |
| HD             | 69.36| 0.0~81.91| 0.0~69.36| 0.0~100.0 |
| DSA            | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |

---

**TABLE 16: Train MLP with AGAICUS (100 Epochs)**

|                | ACCU | F1-SCORE | RECALL | PRECISION |
|----------------|------|----------|--------|-----------|
| SGD            | 52.43| 0.0~68.79| 0.0~52.43| 0.0~100.0 |
| Momentum       | 52.43| 0.0~68.79| 0.0~52.43| 0.0~100.0 |
| RMSProp        | 99.94| 99.94~99.94| 99.87~100.0| 99.88~100.0 |
| AdaDelta       | 79.88| 73.17~83.90| 72.26~100.0| 57.70~100.0 |
| AdaGrad        | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |
| Adam           | 99.94| 99.94~99.94| 99.87~100.0| 99.88~100.0 |
| AdamW          | 99.94| 99.94~99.94| 99.87~100.0| 99.88~100.0 |
| Adamax         | 99.88| 99.87~99.88| 99.87~99.88| 99.87~99.88 |
| HD             | 52.43| 0.0~68.79| 0.0~52.43| 0.0~100.0 |
| DSA            | 100.0| 100.0~100.0| 100.0~100.0| 100.0~100.0 |

---

**Fig. 7: Train loss on MLP**

**Fig. 8: Validation accuracy on MLP**
Fig. 9: Regression loss and track of parameters

because nearly all the optimizer can get an accuracy of 100 percent on IRIS and AGAICUS. Instead, we will choose a distinguishable epoch for each dataset and capture the performance of each optimizer at that point. Concretely, WINE, CAR, IRIS and AGAICUS are captured at 1000th epoch, 500th epoch, 30th epoch and 100th epoch, respectively. We conclude the performance in TABLE 13, 14, 15 and 16. IRIS is the simplest of the four datasets and each optimizer can gain a perfect performance on it as long as training with enough iterations. So we abstract the information of first 30 from the total 1000 epochs to compare the convergence speed of optimizers. Obviously, RMSPprop, AdaGrad and DSA are the most remarkable by achieving an accuracy of 100 percent in only 30 epochs. Similar with AGAICUS, although there are thousands of training samples, it’s just a yes or no problem. So it’s also a not complex task, where AdaGrad and DSA obtain full marks again with few epochs. Actually, DSA can accomplish this within just 30 epochs and this is far more faster than any other optimizers. WINE is a little troublesome task for there are some classes with very little training samples. This is why there are still some classes with a RECALL of 0 even after all the 1000 epochs. Fortunately, AdaGrad, AdamW and DSA still finish the task effectively. As for CAR dataset, only DSA can do it all right within 500 epochs. We also visualize the train loss in Fig. 7 and the validation accuracy in Fig. 8. With help of DSA, train loss always decreases to 0 rapidly surrounding all the other optimizers in the figures, especially on CAR and AGARICUS. There is no doubt that DSA is the most sensitive and efficient algorithm compared with these stat-of-arts.

3.3 Case Study

In this section, we conduct two case studies with DSA and baselines. The first is a simple regression problem to calculate the sum of four real numbers. The second is a convex optimization, i.e., minimization problem.

Regression

As the first case, we solve a simple regression problem calculating the sum of four real numbers with target of

$$\min_w ||w \cdot x^T - y||$$, where $w = [w_1, w_2, w_3, w_4] \in \mathbb{R}^4$.

Obviously, the optimal of each $w_i$ is 1. We set this case to observe the optimization detail of each optimizer from a more intuitive perspective of tracking the value of parameters. In the regression, each $w_i$ is initialized in a kaiming uniform distribution with $a = \sqrt{\frac{5}{2}}$. We randomly generate 10,000 pieces of $(x_1, x_2, x_3, x_4)$ from a uniform distribution $[0, 1)$ and use the sum of $x_i$ as labels. In this case, we fine-tune the learning rate for each optimizer. Finally, learning rate of Adam family and AdaGrad is set to 0.1. DSA’s initial learning rate is 0.05 and the step size of $\alpha$ is 0.5. The others stay the same.

Regression loss and track of parameters are visualized in Figure 9. From the view of regression loss, DSA reaches $10^{-13}$ within about 100 epochs and this is at least twice as rapid as the other optimizers. From the track of parameters, we can see that every optimizer would have a fluctuation around 1. While DSA stabilized first through learning rate adaptation. HD also achieves stability with more iterations and adaptation, i.e., DSA is more sensitive.

Convex Optimization

The second case is a convex optimization with target of

$$\min_{w_1, w_2} a * w_1^2 + b * w_2^2$$.

We set this case to illustrate learning rate adaptation of DSA more intuitively. Obviously, the
optimal is $w_1 = w_2 = 0$. In the experiment, we set different initial position of $(w_1, w_2)$ and different $(a, b)$ combination to observe each optimizer’s performance. Practice consists of two groups. We set a trap for SGD in the first group, where SGD will fluctuate around extreme points by setting $(a, b) = (1, 95)$. And the initial position for the first group is $(w_1(0), w_2(0)) = (-1, 1)$. Similarly, we set a trap for ADAM in the second group, where $(w_1(0), w_2(0)) = (-1, 1)$ and $(a, b) = (1, 1000)$. When conducting the first group, we set all the optimizer’s learning rate or initial learning rate as 0.01 for fair. And to ensure the convergence under HD, we set its step size of learning rate as $10^{-7}$. As for the second group, initial learning rate is 0.001 uniformly and step size of HD is $10^{-4}$. Single test contains 1000 iterations.

Tracks of $(w_1, w_2)$ are visualized in Fig. 10 and Fig. 11. For the first group, we can see that $w_2$ of SGD get into the trap, i.e., fluctuating around 0 wildly. HD and Momentun also stuck into the trap. But HD could get rid of it through its learning rate adaptation. The other optimizers all avoid the trap, especially the DSA. DSA speeds up rapidly in the beginning and slows down timely in the end. As a result, DSA could reach the optimal with extremely few iterations. As for the second group, we set a so troublesome initial position on an extremely steep slope that nearly all the optimizers begin with fluctuations, more or less. It seems that only AdaGrad keeps away from fluctuations. Actually the learning rate of AdaGrad decreases rapidly because of the large gradient of $w_2$ in the first iteration. This makes AdaGrad survived from the fluctuations, while this also makes the learning rate decreases too much so that AdaGrad could not reach the optimal within limited iterations. Surprisingly, although DSA also get into the trap in the beginning, it quickly takes adaptation and rush towards the optimal within only 10 iterations, which is far beyond the capabilities of others.

3.4 Ablation Experiment and Sensitivity Analysis

In this section, we take ablation experiment to evaluate the effect of techniques mentioned in Section 2.3.3, including parameter specific and step size equal learning rate. And we also support evidence that detection technique proposed in Section 2.3.1 is a better policy than adaptation of HD. Technique of internal structure is not analysed here because it’s necessary to ensure learning rate is positive. Additionally, sensitivity analysis for DSA’s hyper-parameters is conducted then.
The dataset is 1000, 500, 200 and 200, respectively. The initial step size of each parameter is smaller than that of DSA.

Ablation Experiment
Experiments are arranged as two groups. The first for detection technique and the second for parameter specific and step size equal learning rate.

Firstly, we show that detection technique of DSA can decrease miss rate greatly compared to HD. Miss frequency is computed in real time along with iterations as Fig. 12. DSA makes no mistakes in the beginning on each dataset. In the later of a training, miss rate of DSA increases because it steps around the extreme point, and so is HD.

Secondly, we show that parameter specific and step size equal learning rate are both important techniques for DSA. DSA will be tested without these techniques. We denote DSA without parameter specific as DSA†, and DSA+ for DSA without step size equal learning rate.

After retest the optimization task in case study with DSA† and DSA+, we draw the tracks of each in Fig. 13. In the track of DSA†, we find that learning rate is very small so that the convergence is very slow. $w_1$ needs a large learning rate and $w_2$ needs a small learning rate, while learning rate is globally unique and small because of $w_2$. So the steps of $w_1$ slow down passively. As for DSA+, the parameter is updated by basic gradient descent and we can get that it’s not faster than DSA obviously because the step size of each step is smaller than that of DSA.

We evaluate DSA† and DSA+ on WINE, CAR, MNIST and SVHN, compared with DSA and some other baselines as shown in TABLE 1. The number of iterations on each dataset is 1000, 500, 200 and 200, respectively. The initial learning rates of DSA† and DSA+ are both 0.001 and the step size of $\alpha$ is 0.1 when dealing with feature datasets. And the initial learning rate will be $10^{-6}$ when applied on the pretrained model of large image datasets.

**Sensitivity Analysis**
Now we take sensitivity analysis for DSA using the convex optimization case, where $(a,b) = (1,1000)$ and $(w_1,w_2)$ starts from $(-0.06,0.001)$. Two key hyper-parameters of DSA are initial learning rate which is essentially $\alpha$, and step size of $\alpha$, i.e., $\beta$. Firstly, $\beta$ is fixed as 0.3 and initial learning rate is observed on various scales including $(0.01,0.001,0.0001)$. The second group fixed initial learning rate as 0.001 and tuned $\beta$ from $(0.1,0.001)$. Performance with different $\alpha$ and $\beta$ are demonstrated in Fig. 14 and Fig. 15. When the initial learning rate increases from 0.0001 to 0.01, DSA tends to approach the optimal with fewer iterations. However, it will need more steps to decrease learning rate around the optimal. DSA also approaches the optimal faster when $\beta$ increases from 0.01 to 1, because learning rate will adapt more sensitively. But it also takes more iterations to decrease learning rate in the end, because learning rate is adapted very large in the beginning. Totally, a sensitive and stable enough configuration on this task is $\gamma\sigma(\alpha) = 0.001, \beta = 0.3$.

**4 Conclusion & Future work**
In this paper, we propose the optimizer with sensitive self-adaptive learning rate for fast and high-qualified convergence. Compared with existing optimizers, DSA has stronger adaptive capabilities and is competent to a variety of machine learning tasks. While this requires a reasonable initial learning rate and step size. In addition, in the later stage of training, the learning rate is still in an active state, i.e., pointless adaptation. Therefore, how to determine a reasonable initial value or eliminate the negative influence of the initial value and make the learning rate of DSA converge stably will be the main issue to be studied next.

**5 Related work**
Learning rate is sure to be the most significant hyperparameter for a machine learning model, which is the core of an optimizer. Current optimizers can be classified to three categories, that is traditional SGD [1], self-adaptive optimizer such as Adam [7], differentiable learning rate based optimizer known as hypergardient descent [8].

Gradient descent [1] is nearly the earliest optimization algorithm, the three primary types including stochastic gradient descent (SGD), mini-batch type and batch type. Model parameters are updated according to their first-order gradient of the cost function, where backpropagation is applied to the calculation of gradients. However, this
kind of naive optimization algorithm performs without any adaptation mechanisms for learning rate, which causes a great dependence to gradient’s magnitude and a trade-off between speed and accuracy.

According to gradient descent’s fakes, one method is to adopt adaptive updating methods for learning rate. Those proposed methods include Momentum [4], AdaGrad [5], RMSProp [6], Adam [7] and so on. Gradient descent with momentum [4] accumulates an exponentially decaying moving average of past gradients and continues to move in their direction [26]. Momentum has the effect of dampening down the change in the gradient [27], which makes it more stable than traditional gradient descent. AdaGrad [5] adaptively scaled the learning rate for each dimension with the sum of the squares of the gradients as denominator. That is, learning rate will keep decreasing in a training session. RMSProp extends AdaGrad to avoid the effect of a monotonically decreasing learning rate [28]. AdaDelta [17] is an adaptive optimizer without learning rate, which is analogous to Newton’s method to find a more accurate step size for each update. Adam [7] combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. Variants of Adam gain new features, for example, AdamW [18] adds a regular term, Adamax [7] extends the two norm to the infinite norm to obtain more stable and concise results, SparseAdam [7] is designed to deal with sparse tensors.

Learning rate scheduler is a trick in the training which has similar effects with adaptive optimizer. For example, we usually set a milestone for learning rate in the half and the three quarters of the training session and in each milestone, the learning rate will be scaled down. Evidence shows SGD always does better than those adaptive optimizers with the help of scheduler.

However, scheduler is still a manual work. Hypergradient descent (HD) [8] is the algorithm that best matches our understanding of adaptation, cause HD is the first to utilize the gradient of learning rate to the cost function to update learning rate. That is, HD makes learning rate differentiable. Similar related works like multi-level HD [29] and so on are still on the way.

ACKNOWLEDGMENTS

This paper was supported by NSFC grant (U1866602 71773025). The National Key Research and Development Program of China (2020YFB1006104).

REFERENCES

[1] C. Lemaréchal, “Cauchy and the gradient method,” Doc Math Extra, vol. 251, no. 254, p. 10, 2012.
[2] L. Bottou, “Stochastic gradient descent tricks,” in Neural networks: Tricks of the trade. Springer, 2012, pp. 421–436.
[3] S. Ruder, “An overview of gradient descent optimization algorithms,” CoRR, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747
Bozhou Chen received the BS degree from the Harbin Institute of Technology, China. He is currently studying for a master’s degree of computer science at Harbin Institute of Technology, Harbin Institute of Technology, China. His research interests include knowledge completion, NAS and some other machine learning problems.

Hongzhi Wang is a Professor and doctoral supervisor at Harbin Institute of Technology, ACM member. His research area is data management, including data quality and graph management. He is a recipient of the outstanding dissertation award of CCF and Microsoft Fellow.

Chenmin Ba received the bachelor’s degree from Harbin Institute of Technology, China, in 2021. He is currently studying for a master’s degree of computer science at Harbin Institute of Technology. His research interests include software vulnerability identification and software vulnerability location.