Gradient Correction beyond Gradient Descent

Zefan Li  
leezf@sjtu.edu.cn

Bingbing Ni*  
nibingbing@sjtu.edu.cn

Teng Li  
tenglwy@gmail.com

Wenjun Zhang  
zhangwenjun@sjtu.edu.cn

Wen Gao  
wgao@pku.edu.cn

1Shanghai Jiao Tong University  2Anhui University  3Peking University

Abstract

The great success neural networks have achieved is inseparable from the application of gradient-descent (GD) algorithms. Based on GD, many variant algorithms have emerged to improve the GD optimization process. The gradient for back-propagation is apparently the most crucial aspect for the training of a neural network. The quality of the calculated gradient can be affected by multiple aspects, e.g., noisy data, calculation error, algorithm limitation, and so on. To reveal gradient information beyond gradient descent, we introduce a framework (GCGD) to perform gradient correction. GCGD consists of two plug-in modules: 1) inspired by the idea of gradient prediction, we propose a GC-W module for weight gradient correction; 2) based on Neural ODE, we propose a GC-ODE module for hidden states gradient correction. Experiment results show that our gradient correction framework can effectively improve the gradient quality to reduce training epochs by $\sim 20\%$ and also improve the network performance.

1. Introduction

To improve the performance of a deep learning system, there are dozens of approaches one can take: enlarging the model structure by making it deeper or larger [26, 28]; improving the initialization scheme [25]; collecting more data, or improving the optimization algorithm. This work aims to improve the model performance by advancing the quality of the gradient used by optimization algorithms.

Among the most popular optimization algorithms, Gradient Descent (GD) has become the most common algorithm to optimize various kinds of neural networks. At the same time, most state-of-the-art artificial intelligence libraries contain implementations of various algorithms for optimizing gradient descent (e.g., Tensorflow [1], Caffe [10] and PyTorch [20]). According to how much data to use when computing the gradient, there are three basic variants of GD: vanilla Batch gradient descent, stochastic gradient descent (SGD) and mini-batch gradient descent. Mini-batch gradient descent is a typical choice of the algorithm to train a neural network and the term SGD is also employed. However, vanilla SGD doesn’t guarantee good convergence. Therefore, more algorithms are proposed to deal with extant challenges [23] of SGD, e.g., Momentum [21], Nesterov [17], Adagrad [4], Adadelta [29], RMSprop [7], Adam [11], Nadam [3] and so on. These methods mostly focus on modifying the gradients or the learning rate within the GD/SGD optimization framework.

Different from these methods, we propose a gradient correction framework by introducing two plug-in modules: GC-W and GC-ODE to modify the calculated gradient. More specifically, the optimization path of GD is not always the optimal path to get to the objective. As shown in Figure 1, when trying to find a local minimal $B$ of a 2D surface, starting from point $A$, two different paths are given: 1) the blue path is obtained by vanilla gradient descent; 2) the red path is given by the gradient correction method. The blue path always points to the quickest direction where $z$ decreases, which is sometimes the drawback of the gradient descent optimization. This is because the GD algorithm ignores the directions where “$z$” temporarily increases but turns out to be the “optimal” direction in the end. By modifying the direction/scale of the gradient, one can get a better/faster path to get to the optimization objective.

A related work is DNI [9], which uses an extra module to generate synthetic gradients for back-propagation. DNI replaces the true gradients with the synthetic ones and updates the gradient synthetic module in an asynchronous manner. However, the quality of the synthetic gradient can hardly surpass the true gradients and only small datasets are tested in DNI. In this work, we first start from an inconspicuous observation in Section 3.1: Layers before down-sampling are the bottleneck during the training of a neural net-
Figure 1. Visualization of a 2D optimization example. View 1-3 show a 2D surface from three different perspectives. Starting from the point $A$, we try to find the local minimal point $B$ on this 2D surface. The blue curve represents the optimization path found by the vanilla gradient descent algorithm. The red curve represents another path from $A$ to $B$. Obviously, compared with the red one, the blue path is not the optimal path to get to the destination. With a proper modification on the gradient direction/scale, one can get a better/faster path to the objective.

work. We conclude that the layers before down-sampling are vital when performing gradient correction. We handle this problem from two aspects: 1) Weight gradient correction; 2) Hidden states gradient correction. We propose two plug-in modules respectively. Inspired from the idea of gradient prediction, we design the GC-W module for weight gradient correction, which uses an attention-like structure to predict the gradient offset. The GC-W is optimized by the optimal gradient directions and scales according to the historical training information. For hidden states gradient correction, we use the neural ODE. We propose the GC-ODE module to improve the gradient coherence of hidden states. As a plug-in branch, the GC-ODE module makes use of neural ODE layers [2] to generate extra gradient data flow with dynamic system properties.

In general, the contributions of this paper can be summarized as follows:

- We create a new gradient correction framework to improve the gradient quality during network optimization.
- We introduce two gradient correction modules: GC-W module for weights gradient correction and GC-ODE module for hidden-state gradient correction.
- Based on extensive experiment results, we show that our gradient correction framework can reduce training epochs and also improve the final performance of a network.

2. Related Work

Our method is mainly analogous to two kinds of methods: gradient descent optimization algorithms and methods improving back-propagation.

Gradient Descent Optimization Algorithms Vanilla GD/SGD [22] algorithm, computes the gradient of the objective function for parameter updating. However, many challenges are encountered, e.g., the setting of the learning rate schedule, the optimization for highly non-convex objectives or saddle points, and so on. Many improvements are proposed to deal with these problems. Momentum [21] is a method to accelerate SGD training by introducing the momentum concept and dampening the oscillations during optimization. Nesterov [17] improves the momentum term to achieve better convergence. Adagrad [4] uses adaptive learning rates for different parameters by performing large updates for infrequent parameters and setting small updates for frequent parameters. Adadelta [29] is an extension of Adagrad, encouraging the parameter update in the direction with less accumulated squared gradients. RMSprop [7] is proposed by Geoff Hinton, which is identical to the first update vector of Adadelta. Adam [11] is a method that combines Momentum and RMSprop, which works well in practice. Another algorithm Nadam [3], incorporating Nesterov into Adam, also turns out to be useful in practice. These methods modify the gradients or learning rates by non-learning strategies. Other methods includes [16, 30]. Different from these methods, we use learning-based modules to improve the gradient quality. As a plug-in method, our gradient correction can be applied together with the aforementioned algorithms to achieve better performance.

Methods Improving Back-Propagation Other methods explore various kinds of improvements upon back-propagation. DTP [13] tries to replace back-propagation with the proposed difference target propagation by using auto-encoders to assign targets to each layer. DTP shows comparable results to back-propagation for deep networks in small datasets. ADMM [27] explores an unconventional training method that uses alternating direction methods and Bregman iteration to train neural networks without gradient descent steps. NoBackTrack [19] introduces an algorithm that computes a stochastic unbiased estimation of the parameter gradients. This algorithm works in an online, memoryless setting and is claimed to require no back-propagation through time. DNI [9] introduces a module of future computation of the network graph and predicts
layer gradient by only local information. GIM [15] removes end-to-end back-propagation by dividing a deep architecture into gradient-isolated modules that are trained by a greedy, self-supervised loss per module. GIM enables asynchronous, decoupled training of neural networks, allowing for the training of arbitrarily deep networks on larger-than-memory input data. Other methods include [6,14,18]. Compare with these methods, our work can advance the quality of gradient and therefore improve the performance of the resulting neural networks.

3. Method

The gradient obtained by gradient descent is a key factor, determining whether a network can be "well-trained". As we perform a gradient descents optimization, many factors can lead to a degradation in the quality of the obtained gradients:

- Low-quality data, such as unclean data or out-of-distribution data, may inject noise in the corresponding gradient.

- Calculation error and system accumulated error can also introduce noise in the computation of the gradient.

- The current gradient is mostly determined by the setting of the current optimization algorithm, which cannot jump out of the framework of SGD.

Therefore, one can hardly get the perfect gradient for gradient propagation. To jump out of the box, we propose our gradient correction method to reduce the noise of the gradient to improve the quality of the obtained gradients.

Figure 2. Observation on Middle Layer. Up: Distance between the current and the optimal weights during training (the smaller the better). Bottom: Gradient direction perturbation between the current and the optimal gradient direction (the smaller the better).

3.1. Observation of Middle Layers

In this section, we start from an observation of the gradients for middle layer features.

**Observation.** Layers before down-sampling are the bottleneck during the training of a neural network.

As shown in Figure 2, we present some visualization of ResNet-18 middle layers during training. We choose 6 layers (2 for each resolution). The same line style denotes the same resolution, i.e., solid, dashed, dotted line style. For each resolution, we choose 2 layers, one is the first layer for this resolution (i.e., conv21, conv31, conv41); the other one is the layer before downsampling (i.e., conv22, conv32, conv42). As we plot the gradient error versus training iteration, the gradient error gradually increases as the gradient propagates from back to the front. It’s worth noticing that, when the gradient propagates to the layers before downsampling (i.e., conv22, conv32, conv42), the error significantly increases. Also, we plot gradient direction perturbation (concerning the optimal direction) against the training epochs, in Figure 2 (bottom). All layers before down-sampling introduce huge fluctuations, compared with deeper layers. In another word, the gradients of layers before down-sampling have lower quality than other layers. Therefore, we conclude that layers before down-sampling are the bottleneck during the training of a neural network.

To deal with the above problem, in the next section, we present our gradient correction scheme for the middle layer features, especially for those layers before down-sampling.

3.2. Gradient Correction for Weights

In this section, we present our gradient correction scheme for weights (GC-W) of a neural network. The idea of this scheme is to introduce an extra gradient correction module, performing gradient correction operation during...
backward propagation. We will first describe how we design this new module. Then we show how to train this module through two different loss functions.

### 3.2.1 Gradient Correction Module

For a neural network layer \( y = f(x, w) \), we denote the input as \( x \), the trainable weight as \( w \) and the output as \( y \). After forward/backward propagation, we get the gradients for input \( G_x \), weights \( G_w \) and output \( G_y \). As we show in Figure 3, we introduce a new module to perform gradient correction on weight gradient \( G_w \). This new module takes the current weight gradient \( G_w \) and the gradient bank \( \tilde{G}_w \) as the input outputs the offset for gradient correction \( \Delta G \). Then we perform an add operation between \( G_w \) and \( \Delta G \) to get the new gradient \( \tilde{G}_w \). Here, the \( \tilde{G}_w \) is the moving average gradient of \( W \), calculated from the past few iterations. \( G_w \) is initialized as a zero tensor at the very beginning. Much like the self-attention mechanism, \( \tilde{G}_w \) is then sent to two \( 1 \times 1 \) convolution functions \( g(x) \) and \( f(x) \) to generate tensor \( K \) and tensor \( V \). Then a multiplication operation is performed between \( K \) and \( V \) to get the tensor \( A \). The multiplication operation is performed between \( Q \) and \( A \). When doing gradient correction, we simply add \( \Delta G \) to \( G_w \) as the new gradient \( \tilde{G}_w \) for weight. It should be noticed that we add an squeeze-and-excitation (SE) structure [8] from \( \Delta G \) to \( \tilde{G}_w \), which turns out to be useful according to our experiments. We name this module as GC-W (with the SE structure) and GC-W* (without the SE structure).

### 3.2.2 How does the GC-W Module work?

The purpose of this GC-W (Gradient Correction for weights) module is to generate beneficial modification for gradients. Suppose \( \Delta G \) always corrects \( G_w \) to the "right" direction, we can replace \( G_w \) with \( \tilde{G}_w \) when updating the network weights during training.

We optimize the GC-W module in an asynchronous policy. On one hand, during the \( t \)-th iteration of training, we get the current gradient \( G_w^t \). We perform forward propagation of GC-W to generate the new gradient \( \tilde{G}_w \), which is used to update the weights of the mainstream network in the \( t \)-th iteration. One the other hand, we use the data pair \((G_w^{t-\tau}, \tilde{G}_w^{t-\tau})\) with gradient reference \( G_{ref}^{t-\tau} \) to train the GC-W module. Here, \( \tilde{G}_w^{t-\tau} \) is the moving average of \((G_w^{t-\tau})\) at the \((t-\tau)\)-th iteration. \( G_{ref}^{t-\tau} \) is the accumulated weight displacement of the past \( \tau \) iterations:

\[
G_{ref}^{t-\tau} = \sum_{t'=t-\tau}^{t-1} w^{t'+1} - w^{t'} = w^t - w^{t-\tau} \tag{1}
\]

The loss function we use here is kernel-wise angle loss \( L_{ang} \):

\[
L_{ang} = \sum_k \cos(<\tilde{G}_w^t(k), G_{ref}^{t-\tau}(k)>) \tag{2}
\]

This loss function minimizes the angle between the corrected gradient \( \tilde{G}_w \) and the reference gradient \( G_{ref}^{t-\tau} \) kernel-wisely. By doing so, \( \Delta G \) corrects \( G_w \) to the "right" direction.

\( L_{ang} \) controls the gradient direction. We also use another loss function \( L_{amp} \) to constrain the gradient amplitude:

\[
L_{amp} = (\max\{\alpha^{t-\tau} - 2|\alpha_{ref}^{t-\tau}|, 0\})^2 \tag{3}
\]

Here, \( \alpha^{t-\tau} \) and \( \alpha_{ref}^{t-\tau} \) is the \( l2 \)-norm of \( \tilde{G}_w^{t-\tau} \) and \( G_{ref}^{t-\tau} \). The optimization algorithm is demonstrated in Algorithm 1.

### 3.3 Gradient Correction for Hidden States

#### 3.3.1 Gradient Correction and Neural ODE

We can view a neural network as a dynamic system: \( \phi(x(t), t) \), which is a function that describes the time dependence of a point \( x \) in a geometrical space. More specifically, we take a neural network as an example. \( x(0) = x_0 \) represents the initial state (i.e., the input tensor) of the network. \( x(1) = x_1 \) represents the final state (i.e., the output tensor) of the network. As we can see in Observation 3.1.
Algorithm 1 Optimization Algorithm for GC-W Module.

Input: Gradient $G_w$, Gradient bank $\tilde{G}_w$, reference Gradient $G_{ref}$
Output: The updated GC-W Module

1: $\triangleright$ Training the main network:
2: $\triangleright\triangleright$ Calculate the new gradient at $t$-th iteration:
3: $G_w^t = GC-W(G_w^t, \tilde{G}_w^t)$
4: Update $w$ using gradient $G_w^t$
5: $\triangleright$ Training the GC-W module:
6: Prepare training data: $G_w^{t-\tau}, \tilde{G}_w^{t-\tau}, G_{ref}^{t-\tau}$
7: $\triangleright\triangleright$ Calculate the new gradient at $t$-th iteration:
8: $L_{ang} = \sum_k \cos(<G_w^{t-\tau}(k), G_{ref}^{t-\tau}(k)>)$
9: $\triangleright\triangleright$ Calculate the amplitude loss:
10: $L_{amp} = (\max\{\alpha l^{t-\tau} - 2\alpha l_{ref}^{t-\tau}, 0\})^2$
11: $\triangleright\triangleright$ Update the GC-W Module according to $L_{ang}$
12: $\triangleright\triangleright$ Update the SE structure according to $L_{amp}$

When we train a neural network, there is some incoherence or discontinuity in gradients of certain layers. To reveal the reason beneath Observation 3.1, we associate the neural network with neural ODE. Similar to [2], we parameterize the derivative of the hidden state using a neural network.

$$\frac{dx(t)}{dt} = \phi(x(t), t, \theta)$$  \hspace{1cm} (4)

Here, $\phi$ is a differentiable continuous function. $\theta$ is the parameter of $\phi$. As claimed in [2], an ODE network defines a vector field, which continuously transforms the state. The derivative of the hidden state $\frac{\partial x(t)}{\partial t}$ is also continuous due to the property of $\phi(x(t), t, \theta)$. Therefore, we propose to improve the between-layer gradient coherence of hidden states through a Neural ODE module.

3.3.2 Neural ODE Module

We now introduce an auxiliary Neural ODE module for gradient correction (GC-ODE). For a neural network block $h_{m+1} = f(h_m)$ with input $h_m \in \mathbb{R}^{h \times w \times c}$ and output $h_{m+1} \in \mathbb{R}^{h \times w \times c}$, we add an auxiliary neural ODE branch:

$$\begin{align*}
\frac{dx_m(t)}{dt} &= \phi(x_m(t), t, \theta), \\
x_m(0) &= h_m
\end{align*}$$  \hspace{1cm} (5)

As we shown in Figure 4, we use an auxiliary branch for neural ODE block for training. During forward propagation, we have:

$$h_{m+1} = \frac{1}{1 + \lambda_0} f(h_m) + \frac{\lambda_t}{1 + \lambda_t} x_m(1)$$  \hspace{1cm} (6)

Here, $\lambda$ is a combination coefficient: $\lambda_t = \frac{\lambda_0 e^{-\alpha t/t_{max} - \beta}}{1 + \lambda_0 e^{-\alpha t/t_{max} - \beta}}$. $\lambda_0$, $\alpha$ and $\beta$ are hyperparameters. $t$ is the training iterations. We use an exponential decay formula here, as we hope to gradually remove the ODE branch as the training goes on.

We take ResNet as an example. According to the resolution of hidden state, ResNet family consists of 4 main stages, i.e., 56 × 56, 28 × 28, 14 × 14, 7 × 7. We add four auxiliary Neural ODE modules, one for each stage.

3.3.3 Forward and Backward Propagation

In this section, we describe how to do optimization and back-propagation to the Neural ODE Module.

To do forward propagation of the Neural ODE Module, we can apply any ODE solver:

$$x(t_1) = ODESolver(x(t_0), \phi, t_0, t_1, \theta)$$  \hspace{1cm} (7)

with $t_0 = 0$ and $t_1 = 1$.

To do back-propagation, suppose we get the gradient of $x(1)$ with respect to the final loss $L$: $\frac{\partial L}{\partial x_0}$. To do backward propagation, we have to compute: 1) $\frac{\partial L}{\partial x_0}$ to optimize the Neural ODE module; 2) $\frac{\partial L}{\partial x_1}$ to pass the gradient to previous layers.

To solve this problem, We follow the Reverse-mode differentiation (RMD) algorithm [2]. As we get $\frac{\partial L}{\partial x_1}$, we can easily get gradients: $\frac{\partial L}{\partial \theta}$ and $\frac{\partial L}{\partial x_0}$:

$$\left[ \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial x_0} \right] = RMD(x_1, \frac{\partial L}{\partial x_1}, t_1, t_0, \theta)$$  \hspace{1cm} (8)

3.4. Combining Gradient Correction Modules

We have presented two gradient correction modules: GC-W and GC-ODE. We can combine these two modules (i.e., GC-W, GC-ODE) together as GCGD to do compound gradient correction. The overall framework is shown in Figure 5. We take ResNet-18 as an example. For all layers
before down-sampling, we insert the GC-W module. For hidden states \( \{h_0, h_1, h_2, h_3, h_4\} \), we insert GC-ODE module between adjacent pairs. Therefore, there are four pairs of GC-W and GC-ODE modules when applying GCGD to ResNet-18. Similarly, when applying GCGD to ResNet-20 for Cifar experiments, we use three pairs of GC-W and GC-ODE modules.

4. Experiment

In this section, we conduct experiments to validate the effectiveness of the proposed gradient correction framework. Firstly, we describe the details to implement the proposed gradient correction modules. Then, ablation studies are conducted on each component of our framework. Then, we report experiment results on several datasets. At last, we carry on some discussion on related topics.

4.1. Implementation Details

Generally, we conduct experiments on both GC-W and GC-ODE modules. Experiments are conducted on CIFAR [12] and ImageNet 2012 classification dataset [24].

For Cifar experiments, we use a weight decay of 0.0001. We train the models with a mini-batch size of 128 on two GPUs. We use vanilla SGD [22] with learning rate 0.1 and divide it by 10 at 100 and 150 training epochs and finish training at 200 epochs. For each training image, we pad 4 pixels on each side and perform a 32×32 random crop for simple data augmentation. The backbone network is based on ResNet-20 [5] with feature map resolution \( \{32 \times 32, 16 \times 16, 8 \times 8\} \). The detailed architecture is shown in Table 1. For each resolution, the GC-W and GC-ODE modules are inserted before the three down-sampling layers.

For ImageNet experiments, we follow a common practice: we perform a scale augmentation followed by a 224×224 random crop to get the training images. Color augmentation is applied. We train all models using Adam [11] with \( \beta_1 = 0.9, \beta_2 = 0.999 \) with a mini-batch size of 2048 on 4 GPUs. The backbone network is based on ResNet [5]. Using resnet-18 as an example, the detailed architecture is illustrated in Figure 5. For each resolution in \( \{56 \times 56, 28 \times 28, 14 \times 14, 7 \times 7\} \), the GC-W and GC-ODE module are inserted before the four down-sampling layers.

The basic implementation details for each module are described as follows:

Gradient Correction for Weights (GC-W) At the beginning of the training, we perform warm-up without doing weight gradient correction for 500 iterations. At the t-th iteration, we calculate the weight gradient moving average with the window length \( \tau = 10 \) of as \( \tilde{G}_w^t \) and collect the training data for GC-W modules: \( \langle G_w^t, \tilde{G}_w^t, G_{ref}^t \rangle \). As long as the training data is collected, we start training the GC-W module, together with the backbone network in an asynchronous way. Starting from the 500-th iteration, we start doing weight gradient correction, as described in Algorithm 1.

Gradient Correction for Hidden States (GC-ODE) The gradient correction modules for Hidden States (GC-ODE)
are trained together with the backbone network. As a plug-in branch described in Section 3.3.2, the GC-ODE module is merged into the backbone network via Equation 6. Here, we set \( \lambda_t = \frac{\lambda_0 e^{-\alpha (t/t_{\text{max}} - \beta)}}{1 + e^{-\alpha (t/t_{\text{max}} - \beta)}} \), with \( \lambda_0 = 1, \alpha = 19.2, \beta = 0.83 \). \( \lambda_t \) gradually decreases from \( \lambda_0 \) to 0 during training.

- Baseline. The backbone network trained with the vanilla SGD.
- GC-W*. Only use GC-W module without SE.
- GC-W. Only use GC-W module with SE.
- GC-ODE. Only use GC-ODE module without GC-W.
- GCGD. Combine GC-ODE module with GC-W.

We use ResNet-18 as the backbone network. For each setting, we train the corresponding network on Cifar-10 and ImageNet. We extract image features from different hidden states, corresponding to different resolutions. For Cifar-10, three hidden states are included: \( \{h_1, h_2, h_3\} \) and for ImageNet, four hidden states are included: \( \{h_1, h_2, h_3, h_4\} \). Then, we train a supervised linear classifier (a fully-connect layer followed by softmax) on those fixed representations.

### Table 1. The network structure we used in Cifar Experiments. The backbone network is ResNet-20 [5]. More details can refer to the ResNet-18 in Figure 5.

| Block | Output size | Layer | W | ODE |
|-------|-------------|-------|---|-----|
| conv1 | 32 x 32     | 7 x 7, 64, 2 | - | - |
| conv2 | 32 x 32     | 3 x 3, 16, 1 | ✓ | ✓ |
| conv3 | 16 x 16     | 3 x 3, 32, 1 | ✓ | ✓ |
| conv4 | 8 x 8       | 3 x 3, 64, 1 | ✓ | ✓ |

### Table 2. Cifar-10 linear classification results. We test the linear evaluation performance of three hidden states under four kinds of settings. The three hidden states \( \{h_1, h_2, h_3\} \) correspond to the output resolution \( \{32 \times 32, 16 \times 16, 8 \times 8\} \) respectively.

| Method | Arch | \( h_1 \) | \( h_2 \) | \( h_3 \) |
|--------|------|----------|----------|----------|
| Baseline | R20 | 65.3 | 81.2 | 91.3 |
| GC-W* | R20 | 65.5 | 81.3 | 91.4 |
| GC-W | R20 | 65.9 | 82.1 | 92.1 |
| GC-ODE | R20 | 66.0 | 81.9 | 91.7 |
| GCGD | R20 | 66.2 | 82.3 | 92.3 |

### Table 3. ImageNet linear classification results. We test the linear evaluation performance of four hidden states under four kinds of settings. The four hidden states \( \{h_1, h_2, h_3, h_4\} \) correspond to the output resolution \( \{56 \times 56, 28 \times 28, 14 \times 14, 7 \times 7\} \) respectively.

| Method | Arch | \( h_1 \) | \( h_2 \) | \( h_3 \) | \( h_4 \) |
|--------|------|----------|----------|----------|----------|
| Baseline | R18 | 30.8 | 45.1 | 63.0 | 70.1 |
| GC-W* | R18 | 30.9 | 46.5 | 63.6 | 70.5 |
| GC-W | R18 | 31.1 | 46.6 | 64.2 | 70.9 |
| GC-ODE | R18 | 31.3 | 46.8 | 64.5 | 71.1 |
| GCGD | R18 | 31.6 | 47.0 | 64.9 | 71.3 |

### 4.2. Ablation Study

In this section, we perform some ablation study on each component of our gradient correction module. We verify our method by linear classification.

In this experiment, we compare five different settings:

- Baseline. The backbone network trained with the vanilla SGD.
- GC-W*. Only use GC-W module without SE.
- GC-W. Only use GC-W module with SE.
- GC-ODE. Only use GC-ODE module without GC-W.
- GCGD. Combine GC-ODE module with GC-W.

We use ResNet-18 as the backbone network. For each setting, we train the corresponding network on Cifar-10 and ImageNet. We extract image features from different hidden states, corresponding to different resolutions. For Cifar-10, three hidden states are included: \( \{h_1, h_2, h_3\} \) and for ImageNet, four hidden states are included: \( \{h_1, h_2, h_3, h_4\} \). Then, we train a supervised linear classifier (a fully-connect layer followed by softmax) on those fixed representations.
| Dataset  | Arch | Method | Accuracy(%) |
|---------|------|--------|-------------|
| Cifar   | R20  | Baseline   | 91.3        |
|         |      | GC-W     | 92.1        |
|         |      | GC-ODE   | 91.7        |
|         |      | GCGD     | **92.3**    |
| Imagenet| R18  | Baseline   | 70.1        |
|         |      | GC-W     | 71.1        |
|         |      | GC-ODE   | 70.9        |
|         |      | GCGD     | **71.3**    |

Table 4. Comparison with state-of-the-art results. We experiment on Cifar-10 and ImageNet dataset. R20 stands for ResNet-20. R18 stands for ResNet-18. For each dataset, we show results of GC-W, GC-ODE and GCGD, as well as the baseline performance.

Table 2 and Table 3 summaries the single crop top-1 classification accuracy on the Cifar-10 test set and ImageNet validation set respectively. As we can see, compared with the baseline network, the proposed GC-W module and GC-ODE module can both bring improvement to the training of the backbone network. More specifically, GC-W outperforms GC-W* in most cases, which indicates that the squeeze-and-excitation structure is useful in the GC-W modules. Also, the combination of GC-W and GC-ODE (GCGD) can further improve the performance of the backbone network.

### 4.3. Linear Classification

In this subsection, we verify our method by linear classification experiments on ImageNet. Table 4 summaries the single crop top-1 classification accuracy on Cifar-10 and ImageNet dataset. Compared with the baseline networks, the proposed GC-W module and GC-ODE module can both bring improvement. GCGD can further improve the network performance (i.e., 1.0% on Cifar-10 and 1.2% on ImageNet).

### 4.4. Memory usage Analysis

For inference, GCGD provides a practical advantage: Network performance can be improved without increasing the computation complexity of the backbone network. First of all, the GC-W module only works during backpropagation. Therefore, the inference speed will not be affected. Secondly, as a plug-in branch, the GC-ODE module is controlled by the parameter $\lambda_t$. As $\lambda_t$ tends to zero, we can get rid of the GC-ODE module after the training. Therefore, both GC-W and GC-ODE modules don’t increase the computation complexity of the backbone network during inference.

For training, GCGD uses an asynchronous training strategy for GC-W and an embedded training strategy for GC-ODE. The GC-W module occupies extra memory in the storage of $\tilde{G}_w$. The GC-ODE module uses extra computation on the forward/backward propagation for the neural ODE layers.

### 4.5. Discussion

#### Training Curve

We present the training curve of SGD, GC-W, GC-ODE, and GCGD in Figure 7. As we can see, GC-W converges faster than GC-ODE as well as the baseline network. GC-ODE accelerates the training of the baseline network. Besides, the proposed GCGD shows great performance and convergence properties during the network training. To reach the same test accuracy, GCGD reduces the overall training epochs by 20%.

#### Limitation

Though the convergence speed is raised, the proposed GC-W and GC-ODE modules introduce extra computation for each epoch during training. Besides, solving the ODE layer is time-consuming for large-dimensional data. Future work involves simplifying the extra module so that training can be further accelerated.

### 5. Conclusion

In this work, we present a gradient correction framework (GCGD) to improve the gradient quality during gradient descent optimization of neural networks. We propose two plug-in gradient correction modules: GC-W and GC-ODE, designed for weights and hidden states respectively. By the proposed gradient correction module, GCGD advances the optimization progress by reducing the overall training epochs and improving the final performance of the trained network. Our future work involves transforming the weight correction module into a weight generation module to replace the original gradient with generated gradient.
References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 265–283, 2016.

[2] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. CoRR, abs/1806.07366, 2018.

[3] Timothy Dozat. Incorporating nesterov momentum into adam. ICLR Workshop, 2016.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research, 12(7), 2011.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[6] Eduin E. Hernandez, Stefano Rini, and Tolga M. Duman. Speeding-up back-propagation in dnn: Approximate outer product with memory, 2021.

[7] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. {Cited on}, 14(8):2, 2012.

[8] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 7132–7141, 2018.

[9] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In International Conference on Machine Learning, pages 1627–1635. PMLR, 2017.

[10] Yangqiu Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Jeff Donahue, et al. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the ACM International Conference on Multimedia, MM ’14, Orlando, FL, USA, November 03 - 07, 2014, pages 675–678. ACM, 2014.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[12] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. -., 2009.

[13] Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In Joint european conference on machine learning and knowledge discovery in databases, pages 498–515. Springer, 2015.

[14] Renjie Liao, Yuwen Xiong, Ethan Fetaya, Lisa Zhang, Kihun Yoon, Xiaq Pitkow, Raquel Urtasun, and Richard Zemel. Reviving and improving recurrent back-propagation. In International Conference on Machine Learning, pages 3082–3091. PMLR, 2018.

[15] Sindy Löwe, Peter O’Connor, and Bastiaan S Veeling. Putting an end to end-to-end: Gradient-isolated learning of representations. arXiv preprint arXiv:1905.11786, 2019.

[16] Yuling Luo, Qiang Fu, Junxiu Liu, Jim Harkin, Liam McDaid, and Yi Cao. An extended algorithm using adaptation of momentum and learning rate for spiking neurons emitting multiple spikes. In International Work-Conference on Artificial Neural Networks, pages 569–579. Springer, 2017.

[17] Y. E. NESTEROV. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. Dokl. Akad. Nauk SSSR, 269:543–547, 1983.

[18] Arild Nokland. Improving back-propagation by adding an adversarial gradient. arXiv preprint arXiv:1510.04189, 2015.

[19] Yann Ollivier, Corentin Tallec, and Guillaume Charpiat. Training recurrent networks online without backtracking. arXiv preprint arXiv:1507.07680, 2015.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[21] Ning Qian. On the momentum term in gradient descent learning algorithms. Neural Networks, 12(1):145–151, 1999.

[22] Herbert Robbins and Sutton Monro. A stochastic approximation method. The annals of mathematical statistics, pages 400–407, 1951.

[23] Sebastian Ruder. An overview of gradient descent optimization algorithms. CoRR, abs/1609.04747, 2016.

[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. International journal of computer vision, 115(3):211–252, 2015.

[25] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In International conference on machine learning, pages 1139–1147. PMLR, 2013.

[26] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In International Conference on Machine Learning, pages 6105–6114. PMLR, 2019.

[27] Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable admm approach. In International conference on machine learning, pages 2722–2731. PMLR, 2016.

[28] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. CoRR, abs/1605.07146, 2016.

[29] Matthew D Zeiler. Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701, 2012.

[30] Lu Zhang, Jiani Wang, and Hui Zhang. New insights in smoothness and strong convexity with improved convergence of gradient descent, 2021.