Reducing Underflow in Mixed Precision Training by Gradient Scaling

Ruizhe Zhao[1] *, **Brian Vogel**[2], **Tanvir Ahmed**[2] and **Wayne Luk**[1]
[1]Imperial College London
[2]Preferred Networks, Inc.
{ruizhe.zhao15,w.luk}@imperial.ac.uk, {vogel,tanvira}@preferred.jp

Abstract

By leveraging the half-precision floating-point format (FP16) well supported by recent GPUs, mixed precision training (MPT) enables us to train larger models under the same or even smaller budget. However, due to the limited representation range of FP16, gradients can often experience severe underflow problems that hinder backpropagation and degrade model accuracy. MPT adopts loss scaling, which scales up the loss value just before backpropagation starts, to mitigate underflow by enlarging the magnitude of gradients. Unfortunately, scaling once is insufficient: gradients from distinct layers can each have different data distributions and require non-uniform scaling. Heuristics and hyperparameter tuning are needed to minimize these side-effects on loss scaling. We propose gradient scaling, a novel method that analytically calculates the appropriate scale for each gradient on-the-fly. It addresses underflow effectively without numerical problems like overflow and the need for tedious hyperparameter tuning. Experiments on a variety of networks and tasks show that gradient scaling can improve accuracy and reduce overall training effort compared with the state-of-the-art MPT.

1 Introduction

Training deep neural networks (DNNs) consumes significant amounts of time, memory, and energy [Strubell *et al.*, 2019]. This situation motivates methods and hardware that make training more efficient. A common solution is using data types with lower precision. Lowering data precision can reduce the computational cost, but the representable range will be narrower and the round-off error will be larger. Mixed precision training (MPT) [Micikevicius *et al.*, 2018] that uses a mixture of 32-bit single-precision (FP32) and 16-bit half-precision (FP16) floating point, provides the opportunity to gain computation efficiency from low-precision data types without losing accuracy. MPT adopts FP16 for compute-intensive yet precision-insensitive operations, e.g., general matrix multiplication (GEMM), while employing FP32 for precision-sensitive functions, such as batch normalization [Ioffe and Szegedy, 2015]. Meanwhile, activations and gradients, which contribute to most of the memory consumption, are represented in FP16, while the weights are in FP32 to reduce the round-off error that appears in long-term accumulation of gradient updates. Nevertheless, since gradients being backpropagated normally have small magnitude, storing them in FP16 may cause severe *underflow*. Typically, gradients in DNN can be smaller than 10^{-10}, and the smallest value that FP16 can represent is only around 6×10^{-8}. That is, values below 6×10^{-8} will be rounded to zero, which further indicates that some critical information may be discarded.

Micikevicius *et al.* [2018] propose loss scaling to address this underflow issue. A loss value will be scaled up before backpropagation, and consequently, gradients being propagated will have larger magnitude. Specifically, suppose the loss is multiplied by scale α (> 1), then every gradient element in backpropagation will be α times larger, compared with training without loss scaling. Meanwhile, when updating model parameters, the gradient update will be correspondingly scaled down by α to keep the same magnitude as normal training. In another words, only the activation gradients will be scaled up, and weight gradients will be the same. If α is chosen properly and gradients are distributed similarly among all layers, underflow rate of gradients will be reduced and MPT can achieve the same as standard full precision training, according to evidence provided by recent papers [Micikevicius *et al.*, 2018; Mellempudi *et al.*, 2019; Kuchaiev *et al.*, 2018]. If not, numerical issues other than underflow may arise, e.g., overflow, which can hurt MPT performance. For example, detection models with a pre-trained backbone network [Liu *et al.*, 2016] have different data distribution between gradients in the detection part and the backbone, which are hard to work with existing MPT (Section 5.3). Some heuristic rules can mitigate these side-effects, but the core issue caused by a single uniform scale remains unresolved.

This paper presents *gradient scaling*, a novel method that aims to provide an effective loss scaling approach for MPT. Instead of scaling the loss value once before backpropagation, we scale each of the gradient tensors appropriately while propagating them. Each scale is calculated on-the-fly, based on the criterion that the estimated underflow rate of the scaled

*Work was mainly performed during an internship at Preferred Networks, Inc.

gradient is just below a given threshold without causing overflow. This gradient-wise scaling method together with this criterion allow us to control the distribution of gradient values in a fine-grained manner, which minimizes the chance of causing numerical problems with a global scale which is too large or too small. Experiments show that our approach, in various scenarios, stabilizes the training behavior and improves the accuracy compared with the state-of-the-art MPT. Our contributions are as follows:

- Proposing and formulating gradient scaling, a method that scales each gradient appropriately during backpropagation to address gradient underflow in MPT.
- Deriving the statistical model that estimates the underflow rate of gradients, and the criterion that decides each scale based on this estimation.
- Evaluating MPT with gradient scaling, revealing its potential for higher accuracy compared with existing MPT on recognition, detection, and segmentation tasks, on a diverse set of networks.

2 Related Work

There are several recent works on low-precision floating-point DNN training. MPT [Micikevicius *et al.*, 2018] mainly explores numerical representation, utilizing a mixture of low and high precision data types. Within the MPT framework, Kuchaiev *et al.* [2018] showcase the advantage of MPT on sequence modelling tasks over FP32, and Mellempudi *et al.* [2019] explore MPT with a mixture of 8-bit floating-point (FP8) and FP16. Both papers mention *dynamic loss scaling*, a variant of the original [Micikevicius *et al.*, 2018] that actively increases the loss scale at a fixed rate, and passively reduces the scale only if overflow occurs. Although it relieves the burden of picking a loss scale manually, dynamic loss scaling should be accompanied with a group of heuristic rules dedicated to specific models, such as back-off [Kuchaiev *et al.*, 2018] and increasing minimum threshold [Mellempudi *et al.*, 2019], to work properly. Tuning these heuristics inevitably brings extra workload. Gradient scaling, on the other hand, can automatically scale gradients without any heuristic, which makes MPT much easier to use in practice.

Several other papers focus on reducing the round-off error in low-precision arithmetic. Wang *et al.* [2018] devise a chunk-based accumulation mechanism to address swamping, a phenomenon that appears when adding a small floating-point number to a much larger one, the smaller operand will be eliminated [Higham, 1993]. Sakr *et al.* [2019] address the same problem, and they propose to find the optimal bit width of floating point accumulation that minimizes the effect from swamping, by maximizing the variance retention ratio, a statistical property that reflects the quality of computation. Interestingly, we find their approach insightful on answering why large loss scale fails. Besides accumulation, Hoffer *et al.* [2018] identify the numerical issues in batch normalization that happen when computing the variance, and propose to replace it by numerically stabler alternatives. Outside deep learning, Higham *et al.* [2019] and Blanchard *et al.* [2019] provide theoretical analysis on mixed precision arithmetic units in NVIDIA GPU typically for linear systems.

3 Estimate Gradient Scale

Underflow happens when non-zero FP32 values are too small to be represented by FP16, which implies that the FP32-to-FP16 *typecasting* should be the target to address underflow. In MPT, gradient is often typecast at two places:

1. Mixed-precision GEMM that computes gradients during backpropagation implicitly typecasts intermediate results in FP32 to FP16 before output.
2. Gradients can be typecast from FP32 to FP16 during the backward pass of an explicit FP16-to-FP32 operator, which is typically inserted before precision-sensitive operators like softmax.

Typically, if the FP32 input is scaled up before typecasting, the underflow rate of the typecast result will be reduced. This trick, which is often used in other mixed-precision scenarios [Blanchard *et al.*, 2019], inspires us to devise gradient scaling for MPT.

The objective of gradient scaling is to adopt scaling before typecasting to reduce the underflow rate. To formulate and then solve this objective, we need to quantify to what extent the underflow rate will be reduced by a given scale on actual gradient data. In this section, we construct a statistical model to estimate the mapping between the underflow rate and gradient scale, which helps us decide the gradient scale for various typecasting operators.

3.1 Preliminaries

An N-bit floating-point number x has one signed bit, N_e exponent bits, and N_m mantissa bits. Here N equals to $1+N_e+N_m$. Suppose these bits are $\{s, e_1, \ldots, e_{N_e}, m_1, \ldots, m_{N_m}\}$, then x equals to $(-1)^s \times M \times 2^E$, where M is the significand equals to $1 + \sum_{i=1}^{N_m} m_i 2^{-i}$, and E is the exponent equals to $\sum_{i=1}^{N_e} e_i 2^{N_e - i} + 1 - 2^{N_e - 1}$. FP32 has $N_e = 8$ and $N_m = 23$ and represents magnitude ranging from 1.2×10^{-38} to 3.4×10^{38}. FP16 only has 5 exponent bits and 11 mantissa bits, resulting in the representation range of $[6 \times 10^{-8}, 65504]$. Obviously, FP16 has a much narrower range of representation than FP32, which is more likely to cause underflow and overflow. Micikevicius *et al.* [Micikevicius *et al.*, 2018] explicitly show how gradients in SSD training cannot be fully represented by FP16.

Notations used in this paper are listed as follows:

- The range of FP16 representation is $[\lambda_{\text{min}}, \lambda_{\text{max}}]$, where $\lambda_{\text{min}} = 6 \times 10^{-8}$ and $\lambda_{\text{max}} = 65504$.
- ∇X, X, and W denote tensors of gradient, activation, weight respectively. ∇x, x, and w are random variables representing their elements.
- f_G stands for mixed-precision GEMM and f_T is the type cast function from FP32 to FP16.
- α denotes the gradient scale to be calculated, and γ represents the underflow rate.

3.2 Assumptions about Gradient Distribution

Intuitively, the underflow rate of ∇X after casting from FP32 to FP16 can be formulated as $P(|\nabla x| \leq \lambda_{\text{min}})$, the CDF

of the absolute gradient values evaluated at λ_{min}. If ∇X is scaled by α, the underflow rate γ then becomes:

$$\gamma = P(\alpha|\nabla x| \leq \lambda_{\text{min}}) \tag{1}$$

Equation 1 relates the underflow rate and the gradient scale. Our model should characterize $P(|\nabla x|)$ to reveal the underlying mapping between α and γ. However, precisely characterizing the distribution of all gradients is intractable. Therefore, we adopt the following assumptions that are commonly used in prior papers to derive mathematically tractable models.

Assumption 1. *∇X, X, and W are independent of each other, and their elements are mutually independent and identically distributed, which can be characterized by random variables ∇x, x, and w, respectively.*

Assumption 1 is used in recent papers [He *et al.*, 2015; Glorot and Bengio, 2010] to derive the statistical relationship among variables during forward and backward propagation for scenarios like network initialization.

Assumption 2. *Product terms in GEMM are mutually independent and identically distributed in zero-mean normal distribution, i.e., $p \sim \mathcal{N}(0, \sigma^2)$, where p is the random variable representing any product term.*

A linear layer, e.g., convolution, applies GEMM on X and W in the forward pass to produce output $Y = f_G(X, W)$. During backpropagation, ∇Y is multiplied by the Jacobian $\frac{\partial Y}{\partial X} = W$ to propagate the gradient to the previous layer, i.e., $\nabla X = f_G(\nabla Y, W)$. Assumption 2 is needed to model the distribution of the output from this GEMM routine. Sakr *et al.* [2019] assume that product terms are *i.i.d.* with zero mean. Based on our empirical observation, we additionally suppose that these product terms share a normal distribution.

Assumption 3. *Gradients propagated from softmax has a log-normal distribution.*

We need to typecast softmax input from FP16 to FP32, and therefore, gradients propagated from softmax will be typecast from FP32 to FP16. Assumption 3 is applied to estimate the underflow rate of gradients in such cases. Since softmax gradients are exponentially related[1] to the normally distributed activations (Assumption 2) output from the last linear layer, it is plausible to assume that they are log-normally distributed.

3.3 Statistical Model

We first introduce the following lemma, which can be directly derived from the assumptions given above.

Lemma 1. *Gradients ∇X calculated from $f_G(\nabla Y, W)$ are normally distributed with zero mean and standard deviation*

$$\sigma_{\nabla x} = \sqrt{N}\sigma_{\nabla y}\sigma_w, \tag{2}$$

where N is the accumulation length in this GEMM.

Proof. Under Assumption 2 each element in ∇X is an accumulation of N product terms $\{p_i\}_{i=1}^N$ from $\mathcal{N}(0, \sigma_p^2)$. Given Assumption 1, σ_p should be $\sigma_{\nabla y}\sigma_w$.[2] We can then derive Equation 2 since $\{p_i\}_{i=1}^N$ are independent of each other. □

[1]When using cross-entropy loss, softmax gradient is in the form of $e^x / \sum e^x - y$, where y is the one-hot label vector. Ignoring the multiplier $\sum e^x$ and bias y, we can get the exponential relationship.

[2]We ignore the mean terms to simplify the expression.

Lemma 1 characterizes the distribution of a gradient produced by GEMM. The following theorem adopts this lemma to provide the underflow rate for such scenario.

Theorem 1. *Given scale α, the underflow rate of $\nabla X = f_G(\alpha\nabla Y, W)$ when casting results from FP32 to FP16 is*

$$\gamma = P(\alpha|\nabla x| \leq \lambda_{min}) = \text{erf}\left(\frac{\lambda_{min}}{\alpha\sqrt{2N}\sigma_{\nabla y}\sigma_w}\right), \tag{3}$$

where erf *is the error function.*

Proof. $|\nabla x|$ follows a *half-normal distribution* since ∇x is normally distributed (Lemma 1). Equation 3 can be straightforwardly derived from the CDF of half-normal distribution and Equation 2. [3] □

We also propose the following theorem to estimate the underflow rate when type-casting softmax gradients.

Theorem 2. *Given scale α and gradients ∇Y from softmax, the underflow rate of $\nabla X = f_T(\alpha\nabla Y)$ is*

$$\gamma = \frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{\ln\lambda_{min} - \mu - \ln\alpha}{\sqrt{2}\sigma}\right), \tag{4}$$

where μ and σ are measured on $\ln\nabla Y$.

Proof. ∇y has a log-normal distribution from Assumption 3, which implies that $\nabla y > 0$ and $\ln\nabla y \sim \mathcal{N}(\mu, \sigma^2)$. Scaled ∇y in the log space, i.e., $\ln(\alpha\nabla y) = \ln\nabla y + \ln\alpha$, has distribution $\mathcal{N}(\mu + \ln\alpha, \sigma^2)$. Meanwhile, since $P(\ln(\alpha\nabla y) \leq \ln\lambda_{\text{min}}) = P(\alpha\nabla y \leq \lambda_{\text{min}})$, we can calculate the CDF of $\ln(\alpha\nabla y)$ evaluated at $\ln\lambda_{\text{min}}$ to get $P(\alpha\nabla y \leq \lambda_{\text{min}})$. Given that $\gamma = P(\alpha\nabla y \leq \lambda_{\text{min}})$, we can figure out Equation 4. □

These equations together form the statistical model for gradient scaling. In the next section, we will present how to incorporate this model with the backpropagation algorithm to implement gradient scaling.

4 The Gradient Scaling Algorithm

The statistical model from the previous section relates the underflow rate and gradient scale, such that we can estimate the required scale from a given underflow rate. That is the core idea of gradient scaling: this algorithm inserts gradient scale calculation for each type cast on the backpropagation path to reduce the rate of underflow. This section provides more details about this algorithm, including:

- How to scale down gradients before updating weights to keep its original magnitude by *propagating scales*.
- The *optimization problem* to be solved for each scale to reduce underflow rate without causing overflow.
- Besides f_G and f_T, how other operators, e.g., concatenation and branching, affect the scales being propagated.

Algorithm 1 covers our gradient scaling algorithm. We explain it piece by piece in the following sections.

[3]Note that if ReLU is applied to ∇Y before f_G, $\sigma_{\nabla x}$ becomes $\sqrt{N/2}\sigma_{\nabla y}\sigma_w$ as argued by [He *et al.*, 2015], and we should update Equation 3 accordingly. We will not explicitly mention the difference in the following discussion.

Algorithm 1: The gradient scaling algorithm.

Input: A DAG of backpropagation operators, of which $op(i)$ denotes the type of the i-th operator; a threshold γ'; and a list of initial gradients in FP32.

1 **for** $i \leftarrow$ *ID of operators in their reversed topological order of the given computational graph* **do**
2 **if** $op(i)$ *is* f_G **then**
3 **variables:** scaled input gradient $\langle \alpha, \nabla Y \rangle$, weights W, and the accumulation length N
4 **statistics:** $\sigma \leftarrow \sqrt{N}\sigma_{\nabla y}\sigma_w$
5 **scale:** local gradient scale $\beta \leftarrow \min\left(\frac{\lambda_{\min}}{\sqrt{2}\sigma \mathrm{erf}^{-1}(\gamma)}, \frac{\lambda_{\max}}{N \max|\nabla X| \max|W|}\right)$.
6 **update:** $W \leftarrow W + f_G(X, \nabla Z)/\alpha$
7 **propagate:** $\langle \beta\alpha, f_G(\beta \nabla Z, W) \rangle$
8 **else if** $op(i)$ *is* f_T *applied to softmax* **then**
9 **variable:** scaled input gradient $\langle \alpha, \nabla Y \rangle$
10 **statistics:** compute μ and σ on $\ln \nabla y$
11 **scale:** $\beta \leftarrow \min\left(e^{[\ln \lambda_{\min} - \mu - \sigma\sqrt{2}\mathrm{erf}^{-1}(2\gamma - 1)]}, \frac{\lambda_{\max}}{\max|\nabla Y|}\right)$
12 **propagate:** $\langle \beta\alpha, \beta \nabla Y \rangle$
13 **else if** $op(i)$ *is concatenation* **then**
14 Duplicate the gradient scale to each partition.
15 **else if** $op(i)$ *is branching or splitting* **then**
16 **input:** a list of gradients $\{\langle \alpha_i, \nabla X_i \rangle\}_{i=1}^M$
17 **scale:** $\beta \leftarrow \max(\{\alpha_j\}_{j=1}^M)$, such that $\beta \max|\nabla X_i|/\alpha_i < \lambda_{\max}, \forall\, 1 \le i \le M$
18 **propagate:** $\{\langle \beta, \beta \nabla X_i/\alpha_i \rangle\}_{i=1}^M$
19 **else**
20 Pass through the received scaled gradient.
21 **end**
22 **end**

4.1 Propagating Scaled Gradient

We keep track of the scale value during backpropagation in the form of a 2-tuple $\langle \alpha, \nabla X \rangle$, in which α is the product of all the scales applied before propagating to ∇X. If we further scale $\langle \alpha, \nabla X \rangle$ by a local scale β, suppose the operator is f_T, the new tuple we will get is $\langle \alpha\beta, \beta \nabla X \rangle$. $\langle \alpha, \nabla X \rangle$ also indicates that without gradient scaling, the magnitude of the gradient calculated at the same point will be α times smaller.

Weight updates, which are gradients propagated to weight variables, should have the same magnitude as what we get without gradient scaling. Otherwise, it may affect gradient descent optimizers, e.g., SGD and Adam [Kingma and Ba, 2015], in the sense that these gradient scales change the *step size* differently for weight update. If the scales are being tracked and updated correctly, then the weight update, in the form of $\langle \alpha, \nabla W \rangle$, can be correctly applied by using $\nabla W/\alpha$. Line 6 in Algorithm 1 shows how weights are updated.

4.2 Optimizing Gradient Scale

Gradient scale should minimize the underflow rate after type casting without causing overflow. This requirement can be formulated as a constrained optimization problem, in which the objective function can be expanded through Equation 3 and Equation 4, and the constraint ensures the largest value will not exceed $\lambda_{\max}$. Note that even if the statistical model for γ is biased, the overflow constraint is strong enough to prevent numerical problems.

$$\min_{\alpha} \gamma = P(\alpha|\nabla X| \le \lambda_{\min}) \quad \mathbf{s.t.} \quad \alpha \max|\nabla X| < \lambda_{\max}$$

Because γ monotonically decreases with α, we end up with deciding α by $\lambda_{\max}/\max|\nabla X|$. In cases that $\max|\nabla X| \ll \lambda_{\max}$, making such a decision can result in very large scale value, which is not necessary and even harmful, since $\alpha > 1$ will enlarge the variance of gradients, i.e., $\mathrm{Var}(\alpha \nabla X) = \alpha^2 \mathrm{Var}(\nabla X) > \mathrm{Var}(\nabla X)$, and may trigger gradient explosion [Glorot and Bengio, 2010; He *et al.*, 2016; Hanin, 2018].

A safer way to decide the scale is by reducing the underflow rate only below a predetermined threshold γ', that is, finding the lower bound on α given by $P(\alpha|\nabla X| \le \lambda_{\min}) \le \gamma'$. By default, we set γ' to 10^{-3}, which indicates our intention to keep the underflow rate below 0.1%. In practice, we constrain the scale value to be a power of two, such that scaling up and down will be simply shifting the exponent bits. It is also worth to mention that α can be smaller than 1, if the type cast will cause overflow for any $\alpha \ge 1$.

4.3 Handling Specific Operators

The input to Algorithm 1 is a DAG representing backpropagation, in which each node denotes an operator of different types, e.g., type cast, GEMM, etc., and each edge specifies dependency. This algorithm loops over each node in the order of dependency, and its body is a big switch conditioned on the operator type. For each case, besides executing the operator itself, we need to make extra effort to scale gradients:

- **GEMM and type casting.** Simply solve the optimization problem based on Equation 3 and 4.
- **Concatenation.** Since the backward pass of a concatenation operator only splits a gradient into multiple parts, we can just distribute the scale of the original gradient.
- **Branching and splitting.** Both operators need to combine multiple gradients, possibly with different scales, together into one scaled gradient. Input gradients should be rescaled to one common value, which can be selected from their current scales in a descending order until one that will not cause overflow to any gradient. Details can be found in Line 17 of Algorithm 1.

5 Experiments

This section empirically evaluates the effectiveness of gradient scaling.

5.1 Experimental Setting

Software. We implemented gradient scaling in the Chainer [Tokui *et al.*, 2019] framework (version 6.3.0). Chainer provides a dynamic computational graph based programming paradigm similar to PyTorch. We also use ChainerCV [Niitani *et al.*, 2017] to apply our algorithm to concrete training examples. The CUDA library (version 10.1) we use supports FP16 and TensorCore [Gupta, 2019].

Model	Type	Method	Test Acc.
ResNet-18	FP32	N/A	70.76%
	FP16	N/A	71.24%
		Fixed-LS	71.39%
		Dyn-LS	71.39%
		GS	**71.44%**
ResNet-50	FP32	N/A	76.49%
	FP16	N/A	76.07%
		Fixed-LS	76.03%
		Dyn-LS	76.12%
		GS	**76.22%**

Table 1: ResNet-18/50 results on the ILSVRC 2012 dataset.

BS	Model	Type	Method	Test mAP
8	SSD-300	FP16	N/A	Diverged
			Fixed-LS	76.23% (best in 7 runs)
			Dyn-LS	69.26%
			GS	**76.35%**
	SSD-512	FP16	N/A	Diverged
			Fixed-LS	**79.29%** (best in 8 runs)
			Dyn-LS	75.33%
			GS	**79.24%**
32	SSD-512	FP32	N/A	80.50%
		FP16	N/A	Diverged
			Fixed-LS	80.01%
			Dyn-LS	80.17%
			GS	**80.31%**

Table 2: SSD-300/512 training results. **BS** denotes batch size.

Hardware. Most of our experiments run on NVIDIA RTX 2080Ti that has the latest Turing architecture [Choquette *et al.*, 2018] and enables TensorCore operations. We also have limited access to a multi-GPU cluster that has several NVIDIA Tesla V100 installed to run distributed training.

Baseline methods. We compare gradient scaling (GS) with the state-of-the-art MPT method with loss scaling, which has two variants: using fixed loss scale (Fixed-LS), or dynamically updating loss scale (Dyn-LS). We also compare with FP32 training results.

Hyperparameters. The only hyperparameter for gradient scaling is the underflow rate threshold γ', which is set as 10^{-3} for all experiments. The update frequency of the gradient scale is per 100 iterations, such that the overhead of calculating gradient scales can be ignored. Task-specific hyperparameters will be listed in the following sections.

5.2 Image Recognition

We first examine ResNet-18 and ResNet-50 [He *et al.*, 2016] on the ILSVRC 2012 dataset [Russakovsky *et al.*, 2015]. These are convolutional neural networks (CNNs) with ReLU, batch normalization [Ioffe and Szegedy, 2015], and residual connections, which sum up two outputs from distinct layers. Both networks can properly benchmark MPT since they are typically used for recognition tasks, and their architectures are representative of recently developed CNNs. We train them in the same hyperparameter setting as [Goyal *et al.*, 2017][4] with 16 GPUs.

Table 1 presents the results. Due to limited computational resources, we only train each scenario once, using 128 as the loss scale for Fixed-LS. Note that among the FP16 results, GS produces the best test accuracy for both networks. However, since the residual structure partially mitigates underflow, the benefit from using gradient scaling is not significant.

5.3 Object Detection

We choose SSD as the benchmark for measuring the performance of MPT with different scaling methods on object detection tasks. SSD is trained on a joint of the PASCAL VOC 2007 and 2012 training sets, and validated on the 2007 test set. The training method[5] is the same as the original paper, with the data augmentation in [Fu *et al.*, 2017].

SSD uses VGG [Simonyan and Zisserman, 2015] as the backbone, which is initialized by pre-trained weights instead of random values. The two SSD variants we examine, SSD-300 and SSD-512, operate on images of 300×300 and 512×512 resolution, respectively. Multiple outputs from the backbone are branched out to feed into predictors that estimate location and class of possible objects at various scales. Thes use of pre-trained weights combined with the branching structure of the architecture can make loss scaling hard to use, since they enforce the gradients at different places to have distinct distributions. However, gradient scaling is expected to be better suited at handling such distributions.

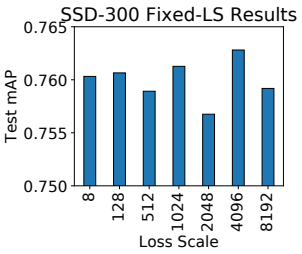
Table 2 shows the result. We first train SSD-300 and 512 with batch size 8, and the reported accuracy is averaged from multiple runs. Specifically, we use a range of loss scales and choose the best result for Fixed-LS. We see that GS performs similarly as the best result from Fixed-LS, and achieves much higher performance than Dyn-LS. We also trained SSD-512 for the same number of iterations but with batch size 32, and in this scenario, GS also gives the highest test mAP in MPT results, only worse than the FP32 baseline. Note that the results from Fixed-LS are the best among multiple runs, which implies the additional training effort that Fixed-LS requires. The following includes some details for further discussion.

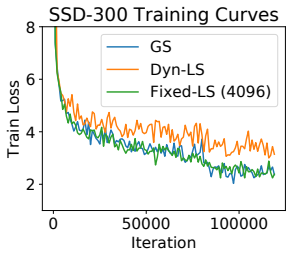
Fixed loss scaling. Fixed-LS requires searching for a loss scale value. Figure 1a shows the non-trivial differences in accuracy among loss scale choices. Taking into account the cost for searching such a hyperparameter, Fixed-LS can be multiple times slower than GS to achieve the same performance.

Dynamic loss scaling. Dyn-LS performs poorly especially when the batch size is 8 (Figure 1b). The major reason is that Dyn-LS implicitly increases the loss scale value and only decreases it when overflow occurs. The distribution of gradients in SSD is very different, and a large loss scale can easily cause overflow. With the same number of training iterations, these more frequent overflows correspond to wasted computation, such that the convergence rate of Dyn-LS slows down.

[4]The script we use is revised from https://github.com/chainer/chainercv/blob/master/examples/classification.

[5]The original training script can be found at https://github.com/chainer/chainercv/blob/master/examples/ssd/.

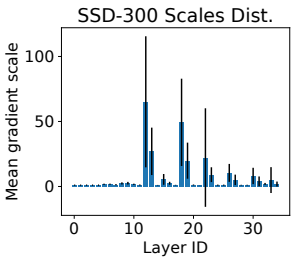




(a) Fixed-LS performance with various loss scales.

(b) Dyn-LS performs the worst.

Figure 1: SSD MPT details using different scaling methods.



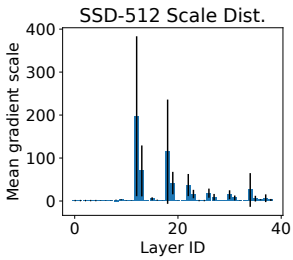


Figure 2: Mean gradient scales for all layers of SSD-300/512 across all training iterations. Layer are labeled from input to output in topological order. Black lines are error bars.

Since larger batch sizes can reduce the chance of exploding gradient, Dyn-LS performs better when the batch size is 32.

Scale distribution. Figure 2 shows the distribution of gradient scales computed for each convolutional layer. Those layers with peak scales are located in network branches that predict object positions and classes at various resolutions. Other layers use smaller scales to further adjust gradients.

5.4 Segmentation

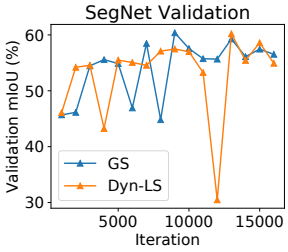
SegNet [Badrinarayanan *et al.*, 2017], a fully convolutional network with an encoder-decoder architecture, is our benchmark for image segmentation. Its encoder part is a VGG-like CNN with batch normalization, and its decoder takes the output from the encoder and processes it by multiple upsampling and convolution layers. The loss is calculated by pixel-wise softmax. When training it with mixed precision, we can cast the encoder-decoder part into FP16 and cast the output back to FP32 for the pixel-wise softmax.

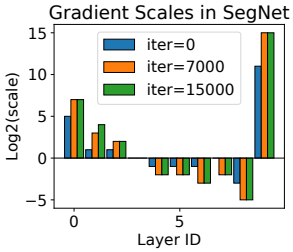
SegNet is evaluated on the CamVid road scenes dataset, which has 367 training and 233 test images at the resolution of 360×480. When training it, we use SGD with batch size 12, learning rate 0.1, and momentum 0.9 for 16K iterations. We compare gradient scaling with other loss scaling methods for MPT, as well as the FP32 training baseline[6].

[6]The original SegNet training script: https://github.com/chainer/chainercv/tree/master/examples/segnet. We also collect the FP32 baseline result through this script, which is better than the reported values from the original paper.

Model	Type	Method	mIoU	CLS	GLB
SegNet	FP32	N/A	51.1%	68.3%	83.9%
	FP16	N/A	Diverged		
		Dyn-LS	48.3%	64.1%	82.4%
		GS	**49.6%**	**64.6%**	**83.4%**

Table 3: SegNet MPT test results on CamVid.





(a) mIoU on the validation set, collected every 1k iterations.

(b) Gradient scales collected for all layers at different iterations.

Figure 3: Details in the SegNet training procedure.

As shown in Table 3, GS performs the best among all MPT results and a bit worse than the FP32 baseline. No exhaustive search for Fixed-LS has been performed, and we only use Dyn-LS as a counterpart. Validation results in Figure 3a shows that training by Dyn-LS can be less stable than GS.

Explaining GS benefits. Figure 3b compares the computed gradient scales in different layers and training iterations. The last layer (ID=8) is the type cast from FP16 to FP32 before pixel-wise softmax, layers 4-7 are the convolution layers in the decoder, and layers 0-3 are in the encoder. The type cast always requires large scales, otherwise the underflow rate will be around 70%. GS can correctly calculate this scale for that type cast. It can also scale down gradients to prevent overflow when the accumulation length is long, specifically, at the intersection of encoder and decoder, which is not available with a single loss scale. The accuracy gap between GS and FP32 is mainly because the extremely small pixel-wise softmax gradients, and fully mitigating underflow requires $\alpha > \lambda_{\max}$.

6 Conclusion

This paper presents gradient scaling, a novel method to tackle gradient underflow in MPT. Our empirical evaluation on a variety of vision-based model architectures shows improved accuracy compared to existing MPT methods. For future work, we plan to evaluate gradient scaling on other tasks and models, especially those for Natural Language Processing, as well as extending it to even lower-precision representations such as 8-bit floating-point, fixed-point, etc.

Acknowledgments

We thank Mitsuru Kusumoto and Kohei Hayashi for their helpful comments.

References

[Badrinarayanan *et al.*, 2017] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A deep convolutional encoder-decoder architecture for image segmentation. *PAMI*, 39(12):2481–2495, 2017.

[Blanchard *et al.*, 2019] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. Mixed Precision Block Fused Multiply-Add : Error Analysis and Application to GPU Tensor Cores. *MIMS EPrint: 2019.18*, 2019.

[Choquette *et al.*, 2018] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and Programmability. *IEEE Micro*, 2018.

[Fu *et al.*, 2017] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C. Berg. DSSD : Deconvolutional Single Shot Detector. 2017.

[Glorot and Bengio, 2010] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, pages 249–256, 2010.

[Goyal *et al.*, 2017] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia Kaiming, and He Facebook. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[Gupta, 2019] Geetika Gupta. Using Tensor Cores for Mixed-Precision Scientific Computing. https://devblogs.nvidia.com/tensor-cores-mixed-precision-scientific-computing/, 2019. "(accessed: 10.01.2019)".

[Hanin, 2018] Boris Hanin. Which Neural Net Architectures Give Rise To Exploding and Vanishing Gradients? In *NeurIPS*, pages 582–591, 2018.

[He *et al.*, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034, 2015.

[He *et al.*, 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, pages 770–778, 2016.

[Higham *et al.*, 2019] Nicholas J Higham, Srikara Pranesh, and Mawussi Zounon. Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems. *SIAM J. Sci. Comput.*, 41(5):585–602, 2019.

[Higham, 1993] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Scientific Computing*, 14(4):783–799, 1993.

[Hoffer *et al.*, 2018] Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: efficient and accurate normalization schemes in deep networks. In *NeurIPS*, pages 2160–2170, 2018.

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, pages 448–456, 2015.

[Kingma and Ba, 2015] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

[Kuchaiev *et al.*, 2018] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-precision training for NLP and speech recognition with OpenSeq2Seq. *arXiv preprint arXiv:1805.10387*, 2018.

[Liu *et al.*, 2016] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *ECCV*, pages 21–37, 2016.

[Mellempudi *et al.*, 2019] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed Precision Training With 8-bit Floating Point. *arXiv preprint arXiv:1905.12334*, 2019.

[Micikevicius *et al.*, 2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *ICLR*, 2018.

[Niitani *et al.*, 2017] Yusuke Niitani, Toru Ogawa, Shunta Saito, and Masaki Saito. ChainerCV: a library for deep learning in computer vision. In *ACM Multimedia*, pages 1217–1220, 2017.

[Russakovsky *et al.*, 2015] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *IJCV*, 115(3):211–252, 2015.

[Sakr *et al.*, 2019] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. In *ICLR*, 2019.

[Simonyan and Zisserman, 2015] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[Strubell *et al.*, 2019] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *ACL*, pages 3645–3650, 2019.

[Tokui *et al.*, 2019] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *KDD*, pages 2002–2011. ACM, 2019.

[Wang *et al.*, 2018] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *NeurIPS*, pages 7686–7695, 2018.