Auto-Precision Scaling for Distributed Deep Learning

Ruobing Han
EECS
Peking University
hanruobing@pku.edu.cn

Min Si
Mathematics and Computer Science Division
Argonne National Laboratory
msi@anl.gov

James Demmel
Computer Science Division
UC Berkeley
demmel@cs.berkeley.edu

Yang You
Department of Computer Science
National University of Singapore
youy@comp.nus.edu.sg

Abstract

In recent years, distributed deep learning is becoming popular in industry and academia. Although researchers want to use distributed systems for training, it has been reported that the communication cost for synchronizing gradients can be a bottleneck, which limits the scalability of distributed training. Using low-precision gradients is a promising technique for reducing the bandwidth requirement. In this work, we propose Auto Precision Scaling (APS), an algorithm that can improve the accuracy when we communicate gradients by low-precision floating-point values. APS can improve the accuracy for all precisions with a trivial communication cost. Our experimental results show that for both image classification and segmentation, applying APS can train the state-of-the-art models by 8-bit floating-point gradients with no or only a tiny accuracy loss (<0.05%). Furthermore, we can avoid any accuracy loss by designing a hybrid-precision technique. Finally, we propose a performance model to evaluate the proposed method. Our experimental results show that APS can get a significant speedup over the state-of-the-art method. To make it available to researchers and developers, we design and implement the CPD system, which can simulate the training process using an arbitrary low-precision customized floating-point format. We integrate CPD into PyTorch and make it open-source to the public.

CCS Concepts **•Software and its engineering → General programming languages; •Social and professional topics →** *History of programming languages;*

Keywords low precision, distributed training

1 Introduction

State-of-the-art deep learning models are becoming deeper and larger, which take an extremely long time to train. As a result, distributed memory systems are becoming popular to train these huge models. Most researchers are using synchronous SGD for data-parallel training [2, 10, 14, 29, 31]. However, we can not always improve the training speed by just using more processors, as the communication cost is a non-trivial overhead for distributed systems and multi-GPU systems [4, 18, 23, 25]. For example, for a 8 NVIDIA GTX1080Ti GPU server, the communication can take around 40% of all wall-clock time for BERT/Wikipedia training. A potential solution is to use low-precision gradients [14, 26]. However, as far as we know, previous methods can only use 16-bit floating point for communicating gradients. One reason is that state-of-the-art communication systems only support half/single/double-precision formats. To solve this problem, we build a system that allows researchers to use an arbitrary low precision format (<32 bits) to communicate gradients. We refer to it as CPD: A High-Performance System for Customized-Precision Deep Learning. We integrate CPD into PyTorch and users can easily apply CPD in their own models. We have put CPD into github [1].

We find that directly using low-precision gradients can easily hurt testing accuracy and even make the training diverge. One reason is that the values in gradients may easily underflow or overflow as the numerical range of the low precision is quite narrow compared to that of the high precision. So there are lots of zeros and INF values, which can make the training process diverge. To solve this problem, we propose the APS (Auto-Precision-Scaling) algorithm, which is a layer-wise adaptive scheme for efficient gradients communication. With APS, we can make the large-batch training converge with only 8 bits or even 4 bits totally for the sign, exponent (exp) and mantissa (man). In our experiments, APS can improve the accuracy for any precision with a minor overhead. Compared to previous methods, the main contributions of our paper include:

- we propose APS, a layer-wise adaptive scheme, that can improve the accuracy for arbitrary low-precision formats;
- we are able to use several 8-bit precision formats to train state-of-the-art classification models and segmentation models on an 8-node distributed system;
- we are able to use 8 bits for gradients to train ResNet-50 on a 256-node distributed system;

Technical Report, 2020
2021. ...$15.00
DOI:

[1]https://github.com/drcut/CPD

- we build a system that can use arbitrarily customized low-precision floating-point operations and make it open-source to the public.

2 Related Work

2.1 Gradient synchronization optimization

There are several approaches that are widely used to reduce the communication cost for gradient synchronization. [14] uses special network topology for communication in large scale distributed system. They split the whole system in multiple groups, and use three phases to implement an all-reduce operation. [24, 26] do not synchronize a gradient as soon as it has been calculated, instead, they copy the calculated gradients into a buffer and all-reduce this buffer as a whole when its size is larger than a target threshold. This way can reduce the negative effect caused by frequently all-reduce of small size gradients (i.e. reduce the latency overhead). Instead of accelerating the process of a single all-reduce communication operation, there are also some previous works focusing on overlapping the computation and communication in the backward propagation process. The most common way is to calculate the former layers' gradients and communicate the calculated gradients concurrently [32].

These works relief the communication cost on system level, these optimization are transparent for users which maintain the communication cost. Besides, there are also some works use special algorithm for gradient synchronization which reduce the communication amount and may change the hyper-parameter of the Neural Network model.

2.1.1 Gradient Sparsification

Nowadays, a state-of-the-art DNN (Deep Neural Networks) model typically has millions of parameters and it will generate the same amount of gradients in back-propagation at each iteration. However, researchers found that some values in gradients are much more important than others [1, 8, 25]: larger values in gradients will have a greater impact on the parameter updating and the training process. Based on this finding, some works only synchronize a part of gradients at each iteration: they set a threshold and only communicate the gradient elements that are larger than it. Other gradient elements are stored locally and accumulated with future gradient elements until they are finally selected to communicate. There are several methods to choose the threshold and accumulate the stale gradients with new gradients [1, 8, 19, 26]. For example, [26] proposed sparse grain all-reduce, which used L1 norm to measure the importance of each gradient element, and only communicate the top 10% important gradient elements at each iteration. [19] proposed DGC, which also communicates a fraction of the gradients each iteration, but has a different way to accumulate the local gradients. All of these methods depend on gradients' magnitude rather than gradients' precision, so our method is orthogonal to these methods.

2.1.2 Gradient Quantization

Recently, researchers are able to use IEEE754 half-precision floating-point to communicate gradients in distributed AlexNet/ResNet training with hundreds of nodes [14, 21, 26]. The underflow/overflow issue is a common problem in low-precision computation. To solve this problem, [21] suggests researchers should carefully select a constant scalar (i.e. loss scaling factor) to scale the loss value, which in turn will scale the gradient value. The constant scalars typically are different for different models and different precisions.

Instead of using low-precision floating point for gradients, some researchers [3, 28] proposed algorithms that quantize the gradients. Both QSGD [3] and TernGrad [28] use the same idea: they encode the gradients to unbiased estimate gradients represented by fewer bits and communicate these gradients with some extra information, and finally decode these communicated results into the normal gradients.

Although these two algorithms also use fewer bits to represent gradients, APS is significantly different from them. Instead of using a customized data structure to represent gradients with fewer bits, APS uses low precision floating-point format to communicate gradients. APS is able to mitigate the round-off error so that we can have close numerical values as the high precision. APS is transparent for high level users, which means they can use the same hyper-parameters and training strategies but with less time spent on communication. For large scale distributed systems, it is extremely expensive to fine-tune the hyper-parameters as it will require lots of computing resources. Thus, it is highly necessary to maintain the same hyper-parameter set. Although QSGD can also maintain the hyper-parameter set, it introduces an extra hyper-parameter, the bucket size, which may significantly affect the accuracy. Ternary can not maintain the same hyper-parameter set because it asks users to decrease dropout ratio to keep more neurons, use smaller weight decay and disable ternarizing in the last classification layer while training on distributed systems. Besides, compared to training on small-scale distributed systems, training on large-scale distributed systems will require lots of accumulation operations, which requires a high numerical precision. Otherwise, the results will be significantly different due to the accumulative effect. The validation of Ternary is only verified on small distributed systems with no more than eight nodes. QSGD is verified on a distributed system that has only 16 nodes. APS does not require any additional hyper-parameters, and it can maintain the hyper-parameter set used for FP32. Besides, we have verified the validation of APS on a large scale distributed system (256 nodes) with state-of-the-art deep learning models. Table 2 summarizes the difference between APS and other related methods.

2.2 Low-Precision for Deep Learning

There are several research projects that explored the possibility of using a lower precision for DNN. However, most of them were focused on the inference stage. Recently, Micikevicius et al. [21] used the half-precision floating-point format (IEEE 754 16-bit) in DNN training. With the help of loss-scaling (the scale factor is a hyper-parameter manually tuned by the researchers), they could achieve a similar accuracy as the FP32 format. After that, Wang et al. [27] were able to use 8 bits in DNN training (they use 16 bits for some parts of the data) and achieve a comparable accuracy to the baseline. The specific design of 8 bits and 16 bits are based on the information of data distributions. Johnson [15] looked into older representations of FP to produce faster silicon. Kalamkar et al. [16] did a comprehensive study on bfloat16 format in DNN training.

Instead of floating-point, some researchers have tried using fixed-point and its variants. [6] used a dynamical fixed point (DFXP) format for parameters, activations and gradients. DFXP will change the scaling factor if overflow occur during training. Instead of changing the scaling factor after overflow happen, [6] designed a predictor to change the scaling factor in advance to avoid overflow.

However, all of the previous low precision DNN training studies are focused on single node (i.e. small-batch training). If we want to finish the training in a short time, we need distributed training on clusters and supercomputers. Although some works use low precision gradient[21, 27], they do not communicate these low precision gradient. Low precision gradient synchronization will involve in round-off error dilemma and will hurt the accuracy, and we will describe this problem in Sec 4.2. In addition to saving bandwidth for gradient synchronization, APS can also be regard as an algorithm that can improve the accuracy for any given precision. We believe this is an important property as many new floating point formats have been proposed these years [16, 27]. Please see Table 1 for more detail.

2.3 Customized-Precision System

Most state-of-the-art systems only support a fixed number of bits in a floating-point format. For example, CUDA only supports floating-point formats with 16, 32, and 64 bits for fixed exponent/mantissa bits. QPyTorch [33] is a recent system that allows users to assign customized numbers of bits to exponent/mantissa in DNN training. QPyTorch is built on top of the PyTorch framework. However, QPyTorch has several limitations that hinder users from using it in real-world applications. When users design a format with only a few bits for exponent, the cast results from IEEE FP32 to the low precision format are numerically incorrect, which will lead to a serious bug in a deep learning system. Besides, it only support using IEEE 754 single-precision for reduce/all-reduce operations, which are being used at each iteration for distributed training.

format	exp bits	man bits	range
IEEE 754 FP32	8	23	$[2^{-149}, 2^{127}]$
IEEE 754 FP16	5	10	$[2^{-24}, 2^{15}]$
BFloat16	8	7	$[2^{-133}, 2^{127}]$
FP16 in [27]	6	9	$[2^{-39}, 2^{31}]$
FP8 in [27]	5	2	$[2^{-16}, 2^{15}]$

Table 1. Different floating-point formats have different representation ranges.

Due to the above limitation, we develop a system for customized-Precision Distributed Deep Learning and call it CPD. We describe the features of CPD in Sec 5.

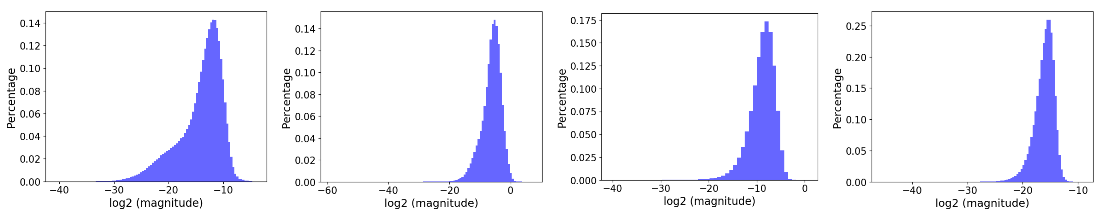
3 APS: Auto-Precision-Scaling

3.1 The limitation of the loss scaling algorithm

The loss scaling algorithm is being used in recent large-scale systems [14, 21, 26]. The key idea of the loss scaling algorithm is: as the ranges that can be presented by low precision and high precision are different, users can scale all layers' gradients with a factor to potentially solve the overflow/underflow problem. Because of the chain rule, users can easily scale all gradients by multiplying loss value with this factor (See Fig. 3 (b)). The loss scaling algorithm requires researchers to find a suitable loss scaling factor for each model, as the gradient distributions for different models are quite different in real-world applications (Fig. 1). Besides, there are several widely used precision formats [15, 16, 27]. For different precisions, the representation ranges are also different (Table 1). Therefore, even for the same model, the suitable loss scaling factors are different when training with different precisions. To make things more complicated, even within a single model, the distributions of different layers are quite different as well (Fig. 2). Previous researchers also reported the gradient distribution for a single layer also changes during the training process [6, 17]. These inconsistencies may make the loss scaling algorithm extremely unreliable in real-world applications. We will discuss this issue in depth in the next section.

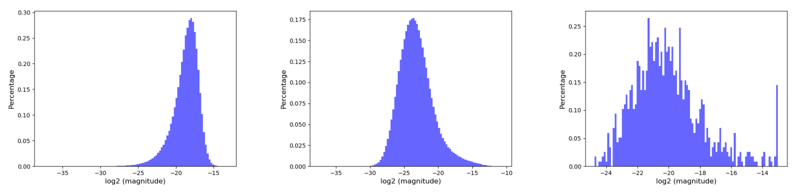
3.2 Layer-wise precision for scaling the gradients

To solve these problems, we propose Auto Precision Scaling algorithm (APS), which uses a layer-wise scheme to scale the gradients. To synchronize the gradients for a given layer, the algorithm first gets the number with the maximum absolute value in this gradient and computes its exponent values, denoted as *max_exp*. Then the algorithm does an all-reduce operation for this scalar to get the maximum value in the whole system. After getting the global maximum value, the algorithm shifts the local gradients on each node according to this value, casts the shifted gradients into low precision, denoted as *low_precision_grad*. Let us refer to the layer ID as i and this number as a_i. Then the algorithm computes the exponent values of $|a_i|$ as e_i. Assume the model has n layers, the algorithm stores a vector $E = \{e_1, e_2, ..., e_i, ..., e_n\}$ in the



(a) LeNet **(b)** DavidNet **(c)** Resnet18 **(d)** Resnet50

Figure 1. Gradients distributions for different Neural Networks. The ranges of gradients in different models are quite different.



(a) res5c_2a_weight **(b)** fc1000_weight **(c)** fc1000_bias

Figure 2. The gradients distributions of different layers in ResNet50 training with 8K batch size. The distributions are totally different. This inconsistency may make the loss scaling algorithm extremely unreliable in the real-world applications.

memory and does an all-reduce operation for this vector to get the maximum value in the whole system. Then the algorithm shifts the gradients of each layer and cast them to a lower precision based on the information of vector E. After finishing an all-reduce operation for these low precision gradients, the algorithm casts them to a higher precision and then shifts them to the original exponent. For more details, please see Algorithm 1. Besides, in a real-world application, we can synchronize the gradients for several consecutive layers as a whole tensor, which can speed up communication process by reducing the latency. The details are presented in Section 4.3.

Figure 3 shows the comparison between the loss scaling algorithm and APS algorithm. When we use 8 bits (exp: 5 bits, man: 2 bits), we can only represent values with exponents in [-16, 15], shown as the area between the two black lines. Values greater than 2^{15} will overflow and cast to INF, while values smaller than 2^{-16} will underflow and cast to 0. The blue curve and green curve represent the gradients' distribution of two layers separately. The loss scaling algorithm will scale all layers' gradients with a given constant number, which is carefully selected by hand to avoid the overflow for the maximum gradients. In this case, the loss scaling algorithm will scale all gradients by 2^{-5}. The scaled gradients are represented by dashed curves (Fig. 3 (b)). Although it can avoid overflow, it will cause some small values to underflow, which will be cast to 0. APS algorithm will scale each layer with a different constant. In other words, the algorithm will automatically scale each layer's gradient with the greatest factor that does not cause overflow. As for the situation the figure shows, we will scale the blue layer by 2^{10}, and the green layer by 2^{-5} (Fig. 3 (c)). We highlight the difference between APS and other widely-used techniques in Table 2.

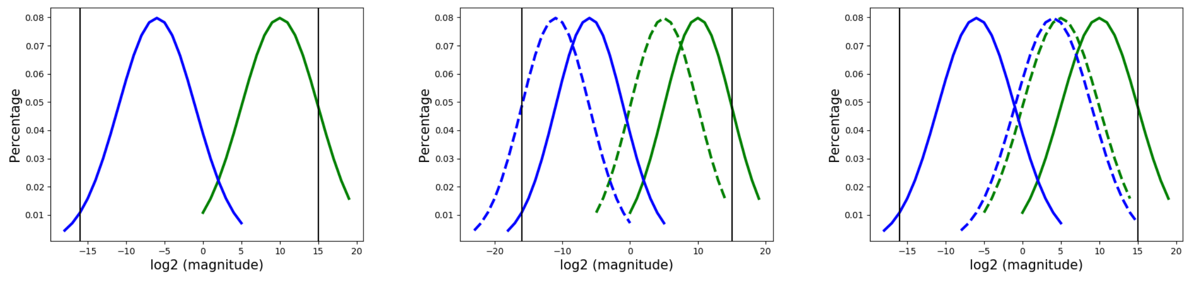
methods	same hyper parameter as FP32	communication cost with gradient size L	extra hyper parameter
APS	yes	allreduce(8 bits) + allreduce(8L bits)	no
loss scaling [21]	yes	allreduce(L * 16 bits)	scaling factor
TernGrad [28]	no	uses special distributed system	no
QSGD [3]	no	depends on coding algorithm	bucket size
flex16+5 [17]	yes	Single node. Gradients: (16L+5) bits	no

Table 2. The difference between APS and other methods.

3.3 Technical details for APS

3.3.1 Using the power of 2 as scaling factors

We cover the technical details of APS in this section to give the readers a better understanding. For loss scaling [21], users can choose arbitrary values as scaling factors. However, in APS, the algorithm will only choose a scaling factor that is the power of two. This choice can take advantage of the properties of the floating-point numbers. By doing so, we can minimize the round-off error. For example, Fig. 4 shows an example of using value 10 or 8 for the scaling factor. We use 8 bits precision (exp: 5 bits, man: 2 bits). The gray box denotes the sign bit, the yellow box denotes the exponent bit, and green box denotes the mantissa bit. For a normal floating-point format, when multiplied by 8 (a value that is the power of 2), only the exponent part will be changed, and the mantissa part will remain the same. So after it is multiplied and divided by 8, the output value is still the same as the input value. While using 10 as the scaling factor, both the exponent and mantissa part will be changed, which may



(a) normal case **(b)** loss scaling **(c)** Auto Precision scaling

Figure 3. These figures show the comparison between loss scaling and APS. When we use 8 bits (exp: 5 bits, man: 2 bits), we can only represent values with exponents in [-16, 15], shown as the area between the two black lines. The data distributions of two layers are represented by blue/green curves separately. Values greater than 2^{15} will overflow and cast to INF, while values smaller than 2^{-16} will underflow and cast to 0. The dashed curves represent the data distributions after scaled.

Algorithm 1 Auto Precision scaling algorithm

```
Input: Gradient: gradient (high precision)
Input: exp_bit: bits of low precision exponent
Input: man_bit: bits of low precision mantissa
Input: world_size: numbers of distributed nodes
upper_bound_exp ← $2^{exp\_bits-1}-1$
for all grad ∈ Gradient do
    max_exp ← FindMaxExp(grad ∗ world_size)
    t ← upper_bound - AllReduce(max_exp, max)
    factor_exp ← t
    grad ← grad∗$2^{factor\_exp}$
    low_precision_grad ←Cast(grad, exp_bit, man_bit)
  ▷ cast to low precision
    t ← AllReduce(low_precision_grad, sum)
    low_precision_grad ← t
    grad ←Cast(low_precision_grad, 8, 23)
            ▷ cast back to high precision (exp: 8, man: 23)
    grad ← grad/$2^{factor\_exp}$
end for

function FindMaxExp(Tensor)
    max_exp ← −INF
    for all i ∈ Tensor do
        if i! = 0 then
            tmp_exp ← ceil(log2(abs(i)))
            if tmp_exp > max_exp then
                max_exp ← tmp_exp
            end if
        end if
    end for
    return max_exp
end function
```

truncate the numerical value. Either multiplied by 10 or divided by 10 will cause a round-off error.

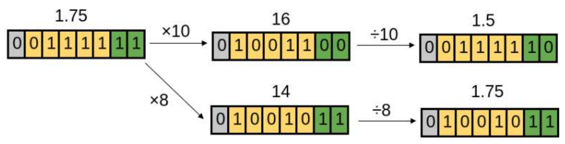


Figure 4. Using 10 and 8 as scaling factor separately for 8-bit precision (exp: 5 bits, man: 2 bits). The gray box denotes the sign bit, the yellow box denotes the exponent bit, and the green box denotes the mantissa bit. Using 8 (the power of 2) as the scaling factor, the output is equal to the input. However, using 10 will lead to a round-off error, which means the output is different from the input.

3.3.2 Trade-off between underflow and overflow

In most cases, numbers represented by high precision formats are out of the ranges low precision formats can represent. So the scaling technique can be a trade-off between underflow and overflow. An example is shown in Figure 5. The original distribution is shown in the green curve, it has both an underflow part and an overflow part. Using a scaling factor larger than 1 will move the green curve to the red curve, which is affected by overflow. In contrast, the blue curve, shifted by a scaling factor smaller than 1, is affected by underflow. However, overflow often can be much more harmful than underflow for deep neural networks training. There are two reasons behind this. Firstly, in backward propagation, the gradients of latter layers are used to calculate the gradients of previous layers. When the gradients of latter layers are overflow and cast to INF, all the gradients in previous layers that depend on them will also be INF. According to the rules of floating point, in most cases, the operators' outputs will be INF if there is an INF for operand. And this domino effect will make the training process diverge as we will lose lots of important information.

Secondly, in [1, 8, 19, 26], the authors do not discard small gradients. They accumulate it with future gradients locally, and communicate the accumulation when it is larger than a threshold. However, for APS algorithm, we discard the small values (the values that will underflow) permanently. If we use 8 bits (exp: 5 bits, man: 2 bits), the maximum gradient is around 2^{15} after scaling, which is the upper bound of this precision. The value smaller than the lower bound (2^{-16}) will underflow. If we store a gradient value locally and accumulate it with the future gradient, there are two possible situations: (1) All future gradients have the same exponent. We have to accumulate $2^{3}1$ times so that this gradient has the same degree of effect as the maximum gradient. (2) The future gradients have a large exponent. In this case, we have to add a small value with a large value. As we have only two bits for mantissa, large values that are 4 times larger than it will incur round-off error, and the small gradient will be discard as well. Taking the above two situations into account, we believe it is unnecessary to store the small gradients. Therefore, our experiments and analysis indicate that we should choose a scaling factor that can avoid overflow. Among all these working values, we choose the largest one, which makes the smallest fraction fall into the underflow range.

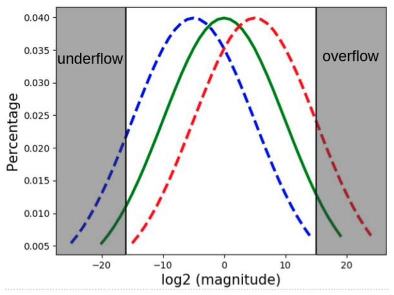


Figure 5. The green curve is the original data distribution. The blue/red dashed curves are distributions scaled by factors smaller/greater than 1.0, which leads to different underflow/overflow fractions.

3.3.3 Find the maximum scaling factor

The above section suggests that we should choose the maximum scaling factor which does not incur overflow. This condition is described by Equation (1), we have to find the maximum value that meets this condition. In this section, we define g as gradients, f as the scaling factor, $\hat{p}$ as the upper bound of the required floating point precision, N as the number of nodes in the system, $\hat{g}$ as the maximum element of the gradients, and $\tilde{f}$ as $\log_2 factor$ ($factor\_exp$ in pseudo-code). Thus, we have the summation over all the distributed nodes:

$$\left|\sum(g \times f)\right| \leq \hat{p} \tag{1}$$

However, as each node only knows its local gradients, it is hard to exactly get the maximum factor with negligible communication cost. So in APS, we use a heuristic algorithm to find a suitable scaling factor. We relax the bound in Equation (1) as Equation (2).

$$\left|\sum(g \times f)\right| = f \times \left|\sum g\right| \leq f \times \sum |g| \leq f \times N \times |\hat{g}| \tag{2}$$

A straightforward approach is to just communicate each node's largest gradient to get the global maximum gradient and then calculate the factor. On top of that, we want to do further optimizations to speed up the communication process. The condition can be written as Equation (3):

$$f \leq \frac{\hat{p}}{|N \times \hat{g}|} \tag{3}$$

As Section 3.3.1 suggests that we should use only the power of 2 as the scaling factor, we can further transform Equation (3) ($f = 2^{\tilde{f}}$ and $\tilde{f}$ is an integer):

$$\tilde{f} < \lceil \log_2(\frac{\hat{p}}{|N \times \hat{g}|}) \rceil = \lceil \log_2(\hat{p}) - \log_2(|N \times \hat{g}|) \rceil \tag{4}$$

So we will assign $\tilde{f} = \log_2(\hat{p}) - \lceil \log_2(|N \times \hat{g}|) \rceil$ to meet the requirements. For a given floating-point number, the logarithm is exactly equal to the exponent part. So instead of communicating $|N \times \hat{g}|$, we only communicate $\lceil \log_2(|N \times \hat{g}|) \rceil$. If we use IEEE 754 floating-point precision and communicate the former value, we have to communicate 32-bit floating point numbers. While using the latter one, we only need to communicate 8 bits, as IEEE 754 floating-point format has 8 bits for exponent.

4 Experiments

It is hardware friendly to use a power of 2 as the number of bits. This is efficient for both memory access and computational operations. So we tried using 4 and 8 bits for gradients in distributed training. We provide an emulator of CPD implementation to make sure our experiments can be reproduced on any device. So we do not emphasize our hardware platform. The major concern for using mixed precision is the casting from high precision to low precision. The standard IEEE floating-point format uses the rounding-to-nearest method, while some researchers prefer stochastic rounding [3, 27, 28] which can get an unbiased estimate for high precision values. Although stochastic rounding has nice mathematical properties, its randomness makes it hard to reproduce. Also, in some situations, it is slower than the rounding-to-nearest method. So in the following experiments, we use round-to-nearest even method, which is a special case of the round-to-nearest method. We fix the number of epochs as the same for all precisions for a given model. As mentioned before, we not only focus on reducing the communication cost, but also want to make APS algorithm transparent for users, which means APS will not change the training process. Therefore, unless otherwise noted, the low precision training will use the same hyper-parameter as IEEE FP32. Besides, we also compare the training curves between APS and the baseline. In this way, we are able to show that using APS does not affect the training process. Most importantly, we also compare the training curves and accuracies with/without APS, to show that APS can improve the accuracy for a given precision.

4.1 Training on small-scale distributed systems

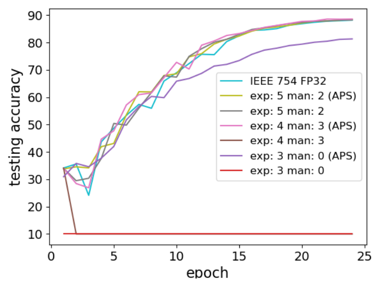
In this section, we pick state-of-the-art deep learning models (DavidNet and Resnet18 for classification and FCN for segmentation) and train them on an 8-node distributed system, and each node has a NVIDIA V100 GPU. We use ring all-reduce [9, 22] for all experiments in this distributed system.

For classification models, we use CIFAR10 dataset and set the batch size as 4K for both models, which means the local batch size is 512 per node in the distributed system. For ResNet18, we set the learning rate as 1.6 and use 5 epochs for learning rate warming up [10] from 0.1. We decay the learning rate with a factor of 0.1 at 40th and 80th epoch. We use Momentum SGD with 0.9 for m. Besides, we use a weight decay γ of 0.0001. For DavidNet, we use Nesterov momentum with m of 0.9 and set γ as 0.256 for weight decay. We first increase the learning rate from 0 to 0.4 linearly in the first 5 epochs and then decrease it to zero linearly in the last 20 epochs. We summarize the relationship between gradient precisions and the accuracy for DavidNet/ResNet18 in Table. 4. We also show the comparison for the training curves of different precisions in Fig. 6. These results show that APS can make a significant difference in low-precision learning.

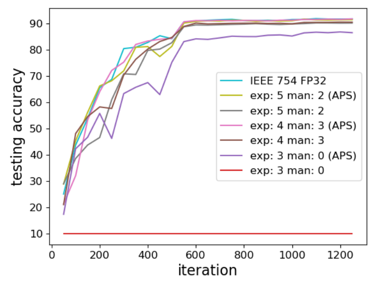
In addition to traditional models, we also select a state-of-the-art segmentation model, FCN [20] (with pre-trained ResNet50 for backbone), for experiments. We use cityscape [5] for dataset. We do our experiments on MMSegmentation[2] and use its hyper-parameter. In the training, we set crop size as 769×769 and train 40K iterations. The experimental results in Table 3 show that we can use 8 bits (exp:4 man:3) to maintain the testing accuracy by APS. We not only compare the numerical values and training curves (Fig. 7), but also compare the segmentation result (Fig. 8). As there is almost no difference between these pictures, it shows that using low precision with APS does not only achieve the same accuracy, but also get a similar model as IEEE FP32. Once again, APS is transparent for high-level users.

LARS[30] is a state-of-the-art method being widely used for distributed training which can significantly improve the testing accuracy. As LARS will set the local learning rate for each layer separately based on gradients, we want to study the relationship between LARS and low-precision gradients. We suspect LARS maybe sensitive to gradients. So we try using LARS with low precision gradients to see if the round-off error caused by the low precision communication hurts

[2]https://github.com/open-mmlab/mmsegmentation



(a) Davidnet



(b) Resnet18

Figure 6. Training with 4K batch size on CIFAR10 dataset using synchronous SGD on an 8-node distributed system.

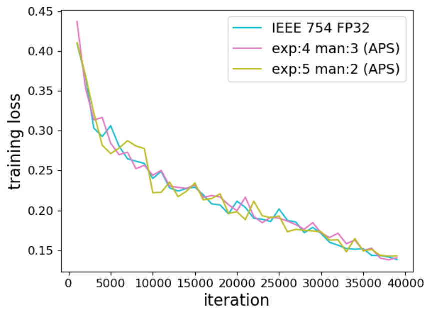
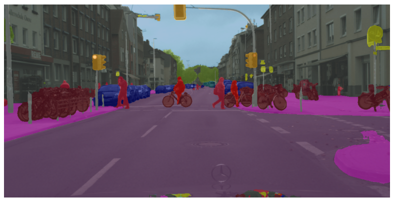


Figure 7. Training FCN on cityscapes dataset with batch size 16 using 8 nodes distributed system. Using 8 bits with APS, we can have close training curve as IEEE FP32

Precision (exp, man)	Using APS	mIOU	mAcc
(8, 23): 32bits	/	75.16	82.84
(4, 3): 8bits	yes	75.88	84.34
	no	74.60	82.55
(5, 2): 8bits	yes	74.76	82.62
	no	74.41	82.30

Table 3. FCN model is trained on cityscapes dataset with batch size 16 by 8 nodes for 40K iterations. Using APS can help improve the mIoU and mAcc for both precisions. While we can achieve even high accuracy using APS with low precision (exp: 4 man:3) comparing with IEEE FP32.

(a) IEEE754 FP32


(b) 8 bits (exp: 4, man: 3)

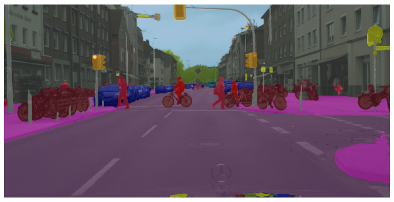

(c) 8 bits (exp: 5, man: 2)

Figure 8. We visualize a segmentation application for the model trained by different precisions. It shows APS can not only achieve the same accuracy, but also get a similar model as IEEE FP32. APS is also transparent for high-level users.

Model	Precision (exp, man)	Using APS	accuracy
DavidNet	(8, 23): 32bits	/	88.2
	(5, 2): 8bits	yes	88.4
		no	88.3
	(4, 3): 8bits	yes	88.6
		no	10.0
	(3, 0): 4bits	yes	81.3
		no	10.0
ResNet18	(8, 23): 32bits	/	91.4
	(5, 2): 8bits	yes	91.4
		no	90.1
	(4, 3): 8bits	yes	91.6
		no	90.4
	(3, 0): 4bits	yes	86.7
		no	10.0

Table 4. Models are trained on CIFAR10 dataset with 4K batch size by 8 nodes. For all precisions, even by 4 bits, APS can make the training processes converge with little or no accuracy loss.

the accuracy or not. We train ResNet18 on CIFAR10 dataset with 8K batch size using 8 nodes and find low precision will hurt the accuracy. On the other hand, by using APS, we can maintain the same accuracy and even improve accuracy. The results are shown in Table 5 and Fig. 9.

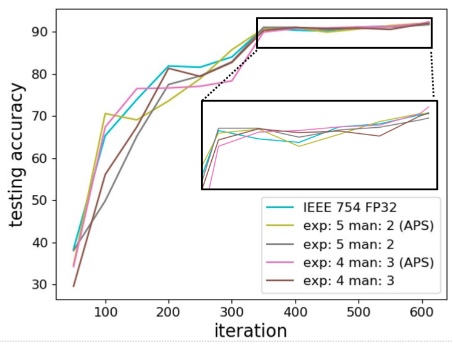


Figure 9. Training ResNet18 on CIFAR10 dataset with 8K batch size using LARS algorithm. APS allows LARS algorithm maintain the same accuracy as 32 bits while using low precision communication.

Precision (exp, man)	Using APS	testing accuracy
(8, 23): 32bits	/	92.072
(4,3): 8bits	yes	92.44
	no	92.036
(5,2): 8bits	yes	92.015
	no	91.737

Table 5. Training ResNet18 with LARS. APS can improve the accuracy for both (exp:5, man:2) and (exp:4, man:3). It can even get a higher accuracy than 32-bit precision.

4.2 Training on large-scale distributed systems

We train ResNet50 [12] on a 256-node distributed system. Instead of ring all-reduce used in Section 4.1, we use the Hierarchical all-reduce [14, 26]: we partition the nodes into 16 groups, and assign a *master* node for each group. Each all-reduce operation will finish 3 steps: (1) within each group, all worker nodes send their local gradients to the master node; (2) we conduct the ring all-reduce across all the master nodes; (3) within each group, the master node broadcasts the global gradients to all the worker nodes. There are two reasons why we use the hierarchical all-reduce approach:

- **Performance**: the ring all-reduce with p nodes need to finish $2(p-1)$ steps (each step transfers the same amount of data). The hierarchical all-reduce with a group size of k only needs $4(k-1)+2(p/k-1)$ steps. In our experiments with 256 nodes and a group size of 16, we only need to finish 74 steps, instead of 510 steps for using ring all-reduce.
- **Round-off error** : when we use a low precision floating point to add a small number with a large number, the smaller number may be truncated and cast as zero in this addition operation. This situation is common in all-reduce process. To avoid this problem, we should try to minimize the number of large-and-small additions. If we use ring all-reduce, we have to add a local gradient with the summation of all other nodes' local gradients in the last step. The summation may be 255x larger than this local gradient if we have 256 nodes. When we use Hierarchical all-reduce, we have only 16 nodes for intra-group reduction. In this situation, the last step will add a local gradient with

a 15x larger gradient. The situation is the same as inter-group all-reduce among 16 master nodes.

Taking the above two factors into account, we choose to use Hierarchical all-reduce with a group size of 16. The reason is that this group size not only can reduce the number of steps from $2(p-1)$ to $4(k-1)+2(p/k-1)$, but also can minimize the round-off error. We use 8K batch size to train ResNet50 on ImageNet dataset [7]. As APS does not require us to modify the hyper-parameters, we use the same setting and data preprocessing as [10], except the learnable scaling coefficient γ is initialized as 1 for all BN layers in our experiments. We adopt the initialization of [11] for the 1000-way fully-connected layer. Based on the suggestions of [27, 28], we use IEEE FP32 for the last layer (i.e. classification layer) and low precision for all other layers. We also have the experimental results of using low precision for all layers, please see Fig. 10 for details. We try using the APS algorithm on different precisions (Table 6), and find APS only needs 8 bits to achieve roughly the same accuracy as the standard 32-bit format. We can further improve the accuracy by hybrid precision: using FP32 for the first 30 epochs and 8 bits for the last 60 epochs. This method can help us maintain the same accuracy as IEEE FP32 (Table 7).

Precision (exp, man)	with APS	top-1 accuracy
(8, 23): 32bits	/	76.02
(5,2): 8bits	yes	75.98
	no	71.00
(4,3): 8bits	yes	75.93
	no	0.1
(8, 23) + (4, 3)	yes	76.09

Table 6. We use APS to train ResNet50 with 8K batch size (256 distributed nodes). APS can improve the accuracy in 8-bit training. With APS, we can use 8 bits instead of 32 bits for the whole training process, with only a tiny loss in testing accuracy (<0.05%). Additionally, we can further improve the accuracy by the hybrid precision: using IEEE 754 FP32 for the first 30 epochs and 8 bits for the last 60 epochs. The hybrid precision with APS can get an even higher accuracy than the FP32 baseline.

Precision for other layers	Precision for the last classification layer	top-1 accuracy
(5, 2)	(5,2)	75.08
	FP32	75.98
(4, 3)	(4, 3)	75.46
	FP32	75.93

Table 7. Training ResNet50 with APS algorithm while using low precision for different layers. We can significantly improve the accuracy by using high precision for the last classification layer.

Precision (exp, man)	group size	top-1 accuracy
(4, 3): 8bits	32	74.95
	16	75.46
(5,2): 8bits	32	74.91
	16	75.08

Table 8. For ResNet50 training on a 256-node distributed system, using a group size of 16 can improve the accuracy compared to a group size of 32 for 8 bits, as it can minimize the round-off error. For this experiment, we use low precision for all layers, including the last classification layer

group size	4	8	16	32	64	256 (ring all reduce)
round-off error	55%	44.21%	41.83%	49.62%	58.21%	85.22%

Table 9. The average round-off error for the first convolutional layer's weight in ResNet50 using 8 bits (exponent: 5 bits, mantissa: 2 bits) in a 256-node distributed system.

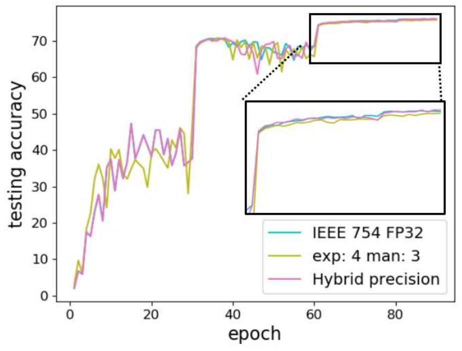


Figure 10. Training ResNet50 with 32 bits, 8 bits, and hybrid precision. We use APS for all low precision process. Using 8 bits for the whole training process can get roughly the same accuracy with FP32. We can further eliminate this tiny gap by hybrid precision: using FP32 in the first 30 epochs and 8 bits in the last 60 epochs.

We also have a comparison between different group sizes and present the results in Table 8. To further explain the difference between different group sizes, we compare the average round-off error for the first convolutional layer's weight using 8 bits (exp: 5, man: 2) and present the result in Table 9. The average round-off error is described by Equation 5 (the gradients calculated by the high precision and the low precision are denoted as $grad\_h$ and $grad\_l$ separately and we assume there are N elements in the gradient tensor). It shows using Hierarchical all-reduce can decrease the round-off error compared to ring-allreduce. It also shows that 16 is the most suitable group size for a 256-node distributed system.

$$average\_round\_off\_error = \frac{\sum_{i=0}^{N} \left| \frac{grad\_h_i - grad\_l_i}{grad\_h_i} \right|}{N} \tag{5}$$

4.3 Performance Analysis

Figure 11 shows the time cost for synchronizing gradients of some layers in ResNet50 (gradient size of each layer: res5c_branch2a: 2048*512, res5c_branch2b: 512*512*3*3, res5c_branch2c: 512*2048). The system has 32 V100 GPUs and uses NCCL for communication. The blue bars denote

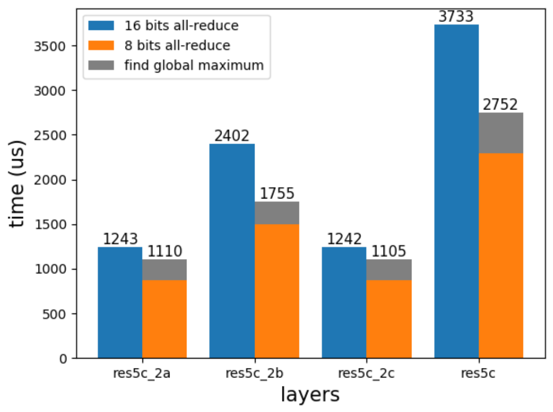


Figure 11. The time of all-reduce by 16 bits with normal all-reduce and all-reduce by 8 bits with APS in a 32-node distributed system. Although APS needs two phases: find the maximum gradient and all-reduce this values (the gray part) and all-reduce the low-precision gradients (the orange part), it is faster than the standard half-precision. If we use lazy all-reduce, which means we do not communicate a layer's gradients immediately but communicate consecutive layers' gradients as a whole, we can save more time by reducing the latency (the rightmost column).

the time cost by using half precision without APS and default ring all-reduce. The bars on the right set of each blue bars show the total time for using APS to communicate the same gradient. The gray bars denote the time cost to get the global maximum gradient. The orange bars denote the time cost to communicate gradients using 8 bits. For all layers, APS with 8 bits can speed up the communication process. Our experiments show that merging short messages into a single one can reduce the overall communication time. Here, res5c_2a, res5c_2b, res5c_2c are three consecutive layers in ResNet50. We synchronize them as a whole, and present the result on the rightmost column in Figure 11. We can achieve a 1.33× speedup over half-precision.

5 CPD: Customized-Precision Deep Learning

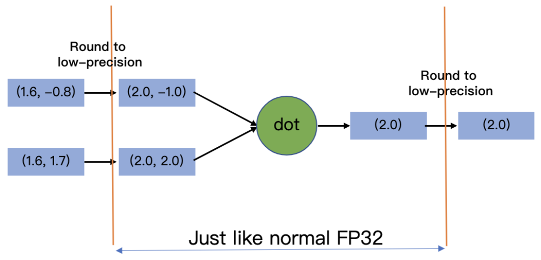
5.1 Features of CPD

According to the above limitations about QPyTorch, we built CPD to emulate the low-precision training for our experiments. CPD has the following functions, which are not supported by any previous systems: (1) arbitrary low-precision, with number of exponent bits <= 8 and number of mantissa bits <=23; (2) Kahan summation algorithm [13]; (3) finishing the GEMM computation using any low-precision accumulator; (4) reduce/all-reduce function using low-precision.

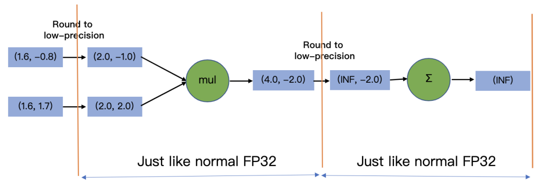
5.1.1 Low-precision accumulation

To add a floating point number to another one with a larger exponent, we need to right shift the mantissa part of the smaller number. In this way, the exponent part of the smaller number can match that of the large number. But this means we may lose the data stored in the mantissa part of the small number after right shift. The situation that a large number is added to a small number is common in two scenarios when we train a neural network model: accumulation (both in GEMM and gradient all-reduce) and parameter updating.

Although it's hard to use a low-precision accumulation for parameter updating, there is a method that can improve the accuracy of the low-precision accumulation for gradient updating and GEMM: Kahan summation algorithm [13]. To the best of our knowledge, this algorithm has never been used for DL. We introduce this algorithm into DL, and support researchers to use it for reduce/all-reduce accumulation and GEMM operation.



(a) QPytorch



(b) CPD

Figure 12. Assume we use 3-bit floating point (2 bits for exp) to do vector multiplication. QPyTorch first casts two input vectors to a low-precision value and calls default dot product to get the result. Then it casts this result to low-precision. As for CPD, instead of calling a default dot product, it calls default multiplication and summation in order and casts the intermediate result to low-precision.

In addition to the accumulation strategy, we are also not able to control the precision of the accumulator in existing systems, and this may cause implicit errors, as shown in Fig. 12. To avoid this in CPD, we implement our own GEMM on NVIDIA GPUs. We are also not able to control the process while doing all-reduce operations in existing systems. CPD allows users to store the intermediate results with a customized-precision, and do accumulation by Kahan summation algorithm. To implement an arbitrary precision all-reduce, we first ask each node to gather all data from other nodes using MPI/NCCL with IEEE 754 float precision, and then accumulate these data independently with a customized precision.

6 Conclusion

Auto Precision Scaling (APS) is a flexible low-precision floating-point technique that can reduce the communication cost. It can train several state-of-the-art applications by 8 bits for gradient communication without losing accuracy. APS can save the bandwidth and improve the accuracy for any given low-precision with almost no cost. For low-precision formats in our experiments, APS can improve the accuracy for all of them. Besides, we can train ResNet-50 by a hybrid precision to maintain the same accuracy as the baseline with the same number of epochs in a distributed system with 256 nodes. We also analyze the time APS used for gradient communication and find that the saving in time is larger than the additional cost in compute. Furthermore, we built the CPD system that allows users to simulate any arbitrary low-precision format. We integrate CPD into PyTorch and make it open source to the public.

References

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325* (2017).

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.

[4] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 571–582.

[5] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3213–3223.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[8] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. IEEE, 1–8.

[9] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. *Baidu Research, Tech. Rep.* (2017).

[10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *The IEEE International Conference on Computer Vision (ICCV)*.

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

[13] Nicholas J Higham. 2002. *Accuracy and stability of numerical algorithms*. Vol. 80. Siam.

[14] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[15] Jeff Johnson. 2018. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721* (2018).

[16] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. 2019. A Study of BFLOAT16 for Deep Learning Training. *arXiv preprint arXiv:1905.12322* (2019).

[17] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. 2017. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*. 1742–1752.

[18] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*. 19–27.

[19] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[20] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully Convolutional Networks for Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[21] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[22] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[23] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[24] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[25] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[26] Peng Sun, Yonggang Wen, Ruobing Han, Wansen Feng, and Shengen Yan. 2019. GradientFlow: Optimizing Network Performance for Large-Scale Distributed DNN Training. *IEEE Transactions on Big Data* (2019).

[27] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*. 7675–7684.

[28] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[29] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992* (2018).

[30] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6 (2017).

[31] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 1.

[32] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 181–193.

[33] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. 2019. QPyTorch: A Low-Precision Arithmetic Simulation Framework. arXiv:cs.LG/1910.04540