

AdaComp: Adaptive Residual Gradient Compression for Data-Parallel Distributed Training

Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal,
Wei Zhang, Kailash Gopalakrishnan

IBM Research AI

1101 Kitchawan Rd. Yorktown Heights, New York 10598
{cchen, choij, danbrand, ankuragr, weiz, kailash}@us.ibm.com

Abstract

Highly distributed training of Deep Neural Networks (DNNs) on future compute platforms (offering 100 of TeraOps/s of computational capacity) is expected to be severely communication constrained. To overcome this limitation, new gradient compression techniques are needed that are computationally friendly, applicable to a wide variety of layers seen in Deep Neural Networks and adaptable to variations in network architectures as well as their hyper-parameters. In this paper we introduce a novel technique - the Adaptive Residual Gradient Compression (**AdaComp**) scheme. AdaComp is based on localized selection of gradient residues and automatically tunes the compression rate depending on local activity. We show excellent results on a wide spectrum of state of the art Deep Learning models in multiple domains (vision, speech, language), datasets (MNIST, CIFAR10, ImageNet, BN50, Shakespeare), optimizers (SGD with momentum, Adam) and network parameters (number of learners, minibatch-size etc.). Exploiting both sparsity and quantization, we demonstrate end-to-end compression rates of $\sim 200\times$ for fully-connected and recurrent layers, and $\sim 40\times$ for convolutional layers, without any noticeable degradation in model accuracies.

Introduction

Over the past decade, Deep Learning (DL) has emerged as the dominant Machine Learning algorithm showing remarkable success in a wide spectrum of application domains ranging from image processing (He et al. 2016), machine translation (Wu et al. 2016), speech recognition (Xiong et al. 2017) and many others. In each of these domains, DNNs achieve superior accuracy through the use of very large and deep models - necessitating up to 100s of ExaOps of computation during training (on GPUs) and GBs of model and data storage.

In order to improve the training time over single-node systems, distributed algorithms (Chilimbi et al. 2014), (Ho et al. 2013), (Lian et al. 2015) are frequently employed to distribute the training data over multiple CPUs or GPUs - using data parallelism (Gupta, Zhang, and Wang 2016), model parallelism (Dean et al. 2012) and pipeline parallelism (Wu et al. 2016) techniques. Data parallelism techniques are widely applied to distribute convolutional layers

while model and pipeline parallelism approaches are more effective for fully-connected and recurrent layers of a neural net. All 3 techniques necessitate very high interconnect bandwidth between the GPUs (in order to communicate the necessary parameters) and impose limits on peak system utilization and model training time. A variety of system topologies have also been explored to efficiently implement distributed training using data parallelism techniques ranging from simple parameter-server based approaches to the recently proposed Wild-Fire technique (Nair and Gupta 2017) that allows direct exchange of weights during reduction. Recently, ring-based system topologies (Luehr 2016) have been proposed to maximally utilize inter-accelerator bandwidths by connecting all accelerators in the system in a ring network. The accelerator then transports its computed weight gradients (from its local mini-batch) directly to the adjacent accelerator (without the use of centralized parameter servers). However, as the number of learners increases, distribution of the minibatch data under strong scaling conditions has the adverse effect of significantly increasing the demand for communication bandwidth between the learners while proportionally decreasing the FLOPs needed in each learner - creating a severe computation to communication imbalance.

Simultaneously, there has been a renaissance in the computational throughput (TeraOps per second) of DL training accelerators - with accelerator throughputs exceeding 100s of TeraOps/s expected in the next few years (Durant et al. 2017), (Merriman 2017). Exploiting hardware architectures based on reduced precision (Gupta et al. 2015), (Courbariaux, Bengio, and David 2014), these accelerators promise dramatic reduction in training times in comparison to commercially available GPUs today. For a given DL network and distribution approach, the bandwidth needed for inter-accelerator communication (in GB/s) scales up directly with raw hardware performance as well as the number of learners. In order to guarantee high system performance, radically new compression techniques are therefore needed to minimize the amount of data exchanged between accelerators. Furthermore, the time required for compression needs to be significantly smaller than the computational time required for back-propagation. Section 2 discusses various prior compression techniques - most of which were applied to Fully Connected (FC) layers of DL networks. But

high computational capacity and wide distribution necessitates compression of convolutional layers in addition to the fully-connected ones. Extending this need to networks that have a mix of fully-connected, convolutional and recurrent layers, it is desirable to have a universally applicable and computationally-friendly compression scheme that does not impact model convergence and has minimal new hyper-parameters. In this paper, we propose a new gradient-weight compression scheme for distributed deep-learning training platforms that fully satisfies these difficult constraints. Our primary contributions in this work include:

- We explore the limitations of current compression schemes and conclude that they are not robust enough to handle the diversity seen in typical neural networks.
- We propose a novel computationally-friendly gradient compression scheme, based on simple local sampling, called AdaComp. We show that this new technique, remarkably, self-adapts its compression rate across mini-batches and layers.
- We also demonstrate that the new technique results in a very high net compression rate ($\sim 200\times$ in FC and LSTM layers and $\sim 40\times$ in convolution layers), with negligible accuracy and convergence rate loss across several network architectures (CNNs, DNNs, LSTMs), data sets (MNIST, CIFAR10, ImageNet, BN50, Shakespeare), optimizers (SGD with momentum, ADAM) and network-parameters (mini-batch size and number of learners).

Residual gradient compression

Background

Given the popularity of distributed training of deep networks, a number of interesting techniques for compressing FC weight gradients have been proposed (Seide et al. 2014), (Strom 2015), (Dryden et al. 2016). Siede (Seide et al. 2014) proposed a one-bit quantization scheme for gradients, which locally stores quantization errors, and reconstruction values are computed using the mean of the gradient values. This scheme achieves a fixed compression rate of 32x applicable only to FC layers. Strom (Strom 2015) proposed a thresholding technique for FC layers that is somewhat similar in approach but can provide much higher compression rates. Only gradient values that exceed a given threshold are quantized to one bit and subsequently propagated to the parameter server along with their indices. These schemes preserve quantization error information in order to reduce the impact of thresholding on the accuracy of the trained model. Furthermore, these papers do not discuss techniques for determining an optimal threshold value. Dryden (Dryden et al. 2016) proposed a “best of both worlds” approach, by combining the one-bit quantization and thresholding ideas. Instead of using a fixed threshold, they propagate a fixed percentage of the gradients, and use the mean of the propagated values for reconstruction. This technique requires sorting of the entire gradient vector which is a computationally expensive task, particularly on a special-purpose accelerator. All the three techniques above are aimed at reducing weight update traffic in deep multi-layer perceptrons composed of

fully-connected layers. One common principle that allows these compression techniques to work without much loss of accuracy is that each learner maintains an accumulated gradient (that we refer to as **residual gradients**) comprising of the gradients that have not yet been updated centrally. We exploit a similar principle in our work.

Related work

For convolutional neural networks (CNNs), the primary motivation for reducing the size of the network has stemmed from the desire to have small efficient models for inference and not from training efficiency point of view. Han et al. (Han, Mao, and Dally 2015) combine network pruning along with quantization and Huffman encoding to decrease the total size of the weights in the model. This is useful as a fine-tuning technique after a trained network is available. Similarly, Molchanov (Molchanov et al. 2016) presented a technique for pruning feature maps while adapting a trained CNN model for distinct tasks using transfer learning.

Recently, a ternary scheme was proposed (Wen et al. 2017) to compress communicated gradients while training CNNs for image processing. Without the use of sparsity, the compression rate in their approach is limited to 16x for both fully connected and convolutional layers. Much higher compression rates ($\sim 100\times$) will be needed in the future to balance computation and communication for emerging deep learning accelerators (Durant et al. 2017). Furthermore, their technique showed significant ($\sim 1.5\%$) degradation in large networks including GoogLeNet. In our work, we apply a novel scheme that exploits both sparsity and quantization to achieve much higher end-to-end compression rates and demonstrate that $<1\%$ degradation is achievable in many state of art networks (e.g. ResNet50 for ImageNet) as well as other application domains (including speech and language). In addition, our approach focuses on compression, which is complimentary to the techniques used by Goyal (Goyal et al. 2017) and Minsik (Cho et al. 2017) on their use of large minibatches for highly distributed training of deep networks.

Observations

We first note that most of the residual gradient compression techniques discussed in Background section do not work well for convolutional layers during training - especially when we also compress the fully connected layers of the same model. As Fig. 1 shows, even for a simple dataset like CIFAR10 (Krizhevsky and Hinton 2009) (Jia et al. 2014), when the fully connected layer is compressed, a simple 1-bit quantization based compression scheme (Seide et al. 2014) for the convolutional layer significantly worsens model accuracy. What is desired is a universal technique that can compress all kinds of layers (including convolutional layers) without degrading model convergence.

Our first observation, inspired by (Molchanov et al. 2016), is that the best pruning techniques consider not only the magnitude of the weights, but also the mini-batch data (and their impact on input feature activities) to maximize pruning efficiency. However, prior work in the compression space assumes that the magnitude of the residue is the only metric indicative of its overall importance. For example, in (Strom

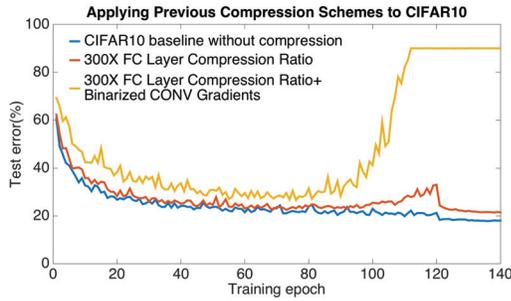


Figure 1: For the CIFAR10 dataset and a network similar to the one in Caffe (Jia et al. 2014), compressing the FC layer only by sending the top 0.3% of the gradients (Dryden et al. 2016) results in a modest degradation in test accuracy (20% vs. 18%). Furthermore, additionally compressing the convolutional layer using a 1-bit quantization scheme (Seide et al. 2014) results in complete model divergence.

2015) a fixed gradient value is used as the threshold, and in (Dryden et al. 2016) a fixed portion of the top gradients are communicated. Intuitively, picking residues with maximum value for any layer may miss lower-magnitude (but critically important) residual gradients that are connected to high activity input features. We also note (empirically) that accumulated residual gradients show little to no correlation with feature activations - which raises further questions about the efficacy of simply picking high valued gradients in any layer.

The second key observation is that neural networks have different activities in different layers depending on overall network architecture, layer types, mini-batch sizes and other parameters, and all of these factors have a direct impact on compression rate. Since compression during training has the potential to dramatically affect convergence, we note that new techniques are desired that automatically and universally adapt to all of these variations without necessitating a whole new set of hyper-parameters.

Finally, it is critical to minimize the computational overhead for any new compression technique given the dramatic reduction in the training time per mini-batch with new high TeraOp/s accelerators. This constraint automatically precludes techniques that require globally sorting of the residue vectors, and instead drives us towards techniques that are accelerator friendly and localized (in terms of memory accesses).

Adaptive Residual Gradient Compression (AdaComp) Technique

Given the lack of correlation between the activity of the input features and the residual gradients in any layer, we conjecture that it is important to have a small enough sampling window that can effectively capture the right residues across the entire layer. To facilitate this, we divide the entire residue vector for each layer (laid out as $NumOutMaps \times NumInpMaps \times KernelRows \times KernelCols$) uniformly into several bins - where the fixed length bin size, L_T is a new hyper-parameter. In each bin, we first find the maximum of the absolute value of the residues. In addition to

this value, we found that it was also important to send several other residues that are relatively similar in magnitude to this maximum. There are several ways to find such important gradients inside each bin. In this paper, we propose a relatively simple self-adjusting scheme. Recall that in each mini-batch, the residue is computed as the sum of the previous residue and the latest gradient value obtained from back-propagation. If the sum of its previous residue plus the latest gradient multiplied by a scale-factor exceeds the maximum of the bin, we include these additional residues in the set of values to be sent (and centrally updated). Empirically, we studied a range of choices for the scale factor (from 1.5 - 3.0 \times) and chose 2 \times primarily for computational ease (simple additions vs. multiplications). The primary intuition here is that since the residues are empirically much larger than the gradients, this scheme allows us to send a whole list of important residues close to the local maximum. Furthermore, we quantize the compressed residue vector in order to increase the overall compression rate. AdaComp is applied to every layer separately - and each learner sends a scale-factor in addition to the compressed sparse vector.

This approach to “threshold” the selection is self-adjusting in 3 ways. First, it allows some bins to send more gradients than others - as many as are needed to accurately represent each bin. Secondly, since the residues are small in the early epochs, more gradients are automatically transmitted in comparison with later epochs. Third, as will be shown in later sections, in comparison to other schemes, this technique minimizes the chances of model divergence that result from an explosion in the residual gradient values and gradient staleness. Thus, AdaComp adaptively adjusts compression ratios in different mini-batches, epochs, network layers and bins. These characteristics provide automatic tuning of the compression ratio, resulting in robust model convergence. We observe that just one hyper-parameter (L_T) is sufficient to achieve high compression rates without loss of accuracy. Finally, it should also be noted that AdaComp, unlike (Dryden et al. 2016), does not require any sorting (or approximations to sorting) and is therefore computationally very efficient ($O(N)$) for high-performance systems.

Pseudo code

The following pseudo code describes two algorithms. Algorithm 1 shows the gradient weight communication scheme we used to test AdaComp, and algorithm 2 is the AdaComp algorithm we propose. Note that our algorithm is not limited to a particular quantization function. In this work, the quantization function uses a sign bit and a scale value to represent the original number. In addition, we use a single scale value for the entire layer - calculated as the average of the absolute values of all the elements in the g_{max} vector. For simplicity of exposition Algorithm 2 assumes that the threshold T evenly divides the $length(G)$.

Experiments

We performed a suite of experiments using the AdaComp algorithm. That algorithm is encapsulated within two functions, pack() and unpack(), inserted into the standard DL

Algorithm 1 Computation Steps

```
learningNoUpdate()      ▷ Forward/Backward only
serializeGrad()         ▷ Collect grad of each layer as a vector
pack()                  ▷ AdaComp Compression for each layer
exchange()              ▷ Learner receives packed grads from others
unpack()                 ▷ AdaComp Decompression for each layer
averageGradients()      ▷ Average among all learners
updateWeights()         ▷ Performed locally at each learner
```

Algorithm 2 Details of pack()

```
 $G \leftarrow residue + dW$       ▷  $dW$  is from serializeGrad()
 $H \leftarrow G + dW$           ▷  $H = Residue + 2*dW$ 
Divide  $G$  into bins of size  $T$ 
for  $i \leftarrow 1, length(G)/T$  do      ▷ Over all bins
    Calculate  $g_{max}(i)$ ; ▷ Get largest absolute value in each bin
end for
for  $i \leftarrow 1, length(G)/T$  do      ▷ Over all bins
    for  $j \leftarrow 1, T$  do              ▷ Over all indices within each bin
         $index \leftarrow (i - 1) * T + j$ 
        if  $|H(index)| \geq g_{max}(i)$  then ▷ Compare to local max
             $Gq(index) \leftarrow Quantize(G(index))$ 
            add  $Gq(index)$  to a pack vector (sent in exchange())
             $residue(index) \leftarrow G(index) - Gq(index)$ 
        else
             $residue(index) \leftarrow G(index)$  ▷ No transmission
        end if
    end for
end for
```

flow between the backward pass and the weight-update step (Algorithm 1). The pack/unpack algorithms are independent of the exchange() function which depends on the topology (ring-based vs. parameter-server based), and therefore the exchange() function is not a subject of this paper. AdaComp impacts 4 critical parameters during DL training: 1) communication overheads, 2) extra time spent executing the pack() and unpack() functions, 3) convergence, and 4) compression ratios. In this paper, we evaluate convergence and compression (items 3 and 4) - but do not report the impact on runtime (items 1 and 2).

Methodology

Experiments were done using IBM SoftLayer cloud servers where each server node is equipped with two Intel Xeon E5-2690-V3 processors and two NVIDIA Tesla K80 cards. Each Xeon processor has 12 cores running at 2.66GHz and each Tesla K80 card contains two K40 GPUs each with 12GB of GDDR5 memory. The software platform is an in-house distributed deep learning framework ((Gupta, Zhang, and Wang 2016), (Nair and Gupta 2017)). The exchange of gradients is done in a peer-to-peer fashion using MPI. In addition, we use synchronous SGD - where all the learners always have identical weights at each step.

Table 1 records the details of the datasets and neural network models we use in this paper. Here we briefly describe the network architectures used in our experiments.

- MNIST-CNN (LeCun et al. 1998): 2 convolutional layers (with 5x5 filters and Relu activation functions), 2 FC layers, and a 10-way softmax.

- CIFAR-CNN (Krizhevsky and Hinton 2009): 3 convolutional layers (with 5x5 filters and Relu activation function), 1 FC layer, and a 10-way softmax.
- AlexNet (Krizhevsky, Sutskever, and Hinton 2012): 5 convolutional layers and 3 FC layers. The output layer is a 1K softmax layer.
- ResNet18 (He et al. 2016): 8 ResNet blocks totaling 16 convolutional layers with 3x3 filters, batch normalization, Relu activation and a final FC layer with a 1K softmax.
- ResNet50 (He et al. 2016): 16 bottleneck ResNet blocks totaling 48 convolutional layers with 3x3 or 1x1 filters, batch normalization, Relu activation and a final FC layer with a 1K softmax.
- BN50-DNN (van den Berg, Ramabhadran, and Picheny 2017): 6 FC layers (440x1024, 1024x1024, 1024x1024, 1024x1024, 1024x1024, 1024x5999) and a 5999-way softmax.
- LSTM (Karpathy 2015): 2 LSTM layers (67x512, 512x512), 1 FC layer (512x67) and a 67-way softmax.

Experiment Results

To demonstrate the robustness as well as the wide applicability of the proposed AdaComp scheme, we tested it comprehensively on all 3 major kinds of neural networks: CNNs, DNNs, and LSTMs. For CNN, five popular networks for image classification were tested: MNIST-CNN, CIFAR10-CNN, AlexNet, ResNet18, and ResNet50. We also included in our tests two pure DNNs (BN50-DNN for speech and MNIST-DNN (not shown)), and an RNN (LSTM). In all these experiments we used the same hyper-parameters as the baseline (i.e., no compression). The selection of L_T is empirical and is a balance between communication time and model accuracy; the same values are used across all models: L_T is set to 50 for convolutional layers and to 500 for FC and LSTM layers.

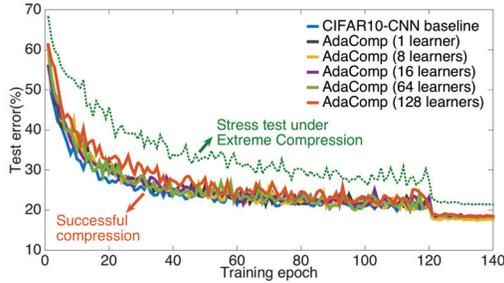
The experimental results are summarized in Table 2, while the detailed convergence curves are shown in Fig. 2. The proposed AdaComp scheme, on every single network (with different datasets, models and layers - CNNs, DNNs and LSTMs), tested under a wide range of distributed system settings (from 1 to 128 learners), achieved almost identical test errors compared with the non-compressed baseline.

Table 1: Dataset and Model

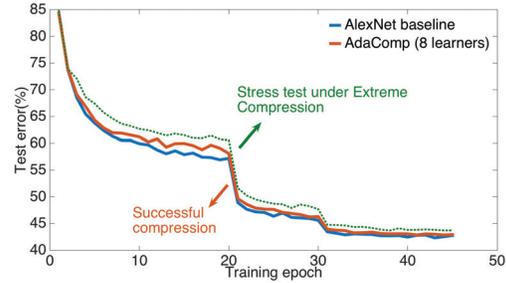
Name	Dataset		Model	
	#Sample	Size	Name	Size
MNIST	60k	181MB	CNN	1.7MB
CIFAR10	50k	1GB	CNN	0.3MB
ImageNet	1.2M	140GB	AlexNet	288MB
			ResNet18	44.6MB
			ResNet50	98MB
BN50	16M	28GB	DNN	43MB
Shakespeare	50k	4MB	LSTM	13MB

Table 2: CNN, MLP, and LSTM results

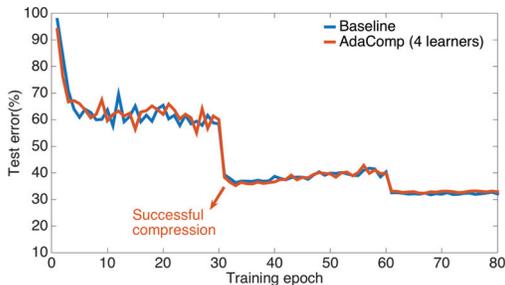
Compression hyper-parameters: convolution layer L_T : 50 and fully connected layer L_T : 500							
Model	MNIST-CNN	CIFAR10-CNN	AlexNet	ResNet18	ResNet50	BN50-DNN	LSTM
Dataset	MNIST	CIFAR10	ImageNet	ImageNet	ImageNet	BN50	Shakespeare
Mini-Batch size	100	128	256	256	256	256	10
Epochs	100	140	45	80	75	13	45
Baseline (top-1)	0.88%	18%	42.7%	32.41%	28.91%	59.8%	1.73%
Our method (top-1)	0.85% (8L)	18.4%(128L)	42.9%(8L)	32.87%(4L)	29.15%(4L)	59.8% (8L)	1.75% (8L)
Learner number	1,8	1,8,16,64,128	8	4	4	1,4,8	1,8



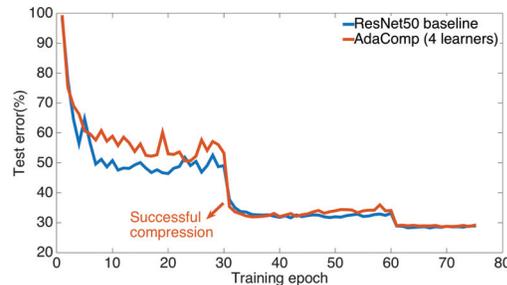
(a) CIFAR10-CNN for CIFAR10 dataset: For Stress test under Extreme Compression, $L_T = 800$ is used for CONV and 8000 is used for FC layers



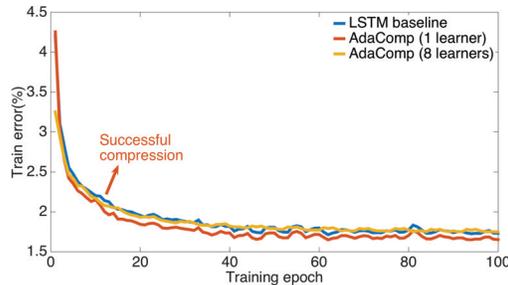
(b) AlexNet for ImageNet dataset: For Stress test under Extreme Compression, $L_T = 500$ was used for both CONV and FC layers.



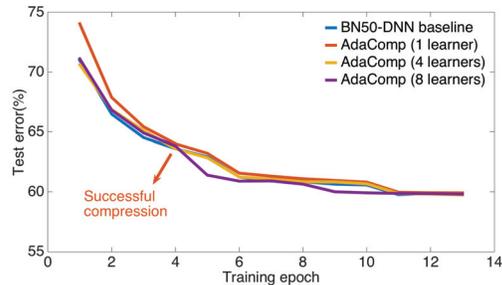
(c) ResNet18 for ImageNet dataset



(d) ResNet50 for ImageNet dataset



(e) LSTM for Shakespeare dataset



(f) BN50-DNN for BN50 speech dataset

Figure 2: Model convergence results for different networks, datasets and learner numbers. In all tests (except for the Stress Tests) the same compression hyper-parameters are used: L_T of 50 for CONV layers and 500 for FC/LSTM layers. Excellent compression ratios of $40\times$ for CONV layers and $200\times$ for FC/LSTM layers are obtained with no degradation in model convergence or accuracy. Stress tests are also shown for CIFAR10-CNN and AlexNet to demonstrate the limits of compression.

In addition to the conventional SGD with momentum optimizer, we also applied the AdaComp technique to other optimizers such as Adam (Kingma and Ba 2014). We ran experiments in Adam for the CIFAR10-CNN model and found

that our compression scheme achieved similar compression ratios with Adam. In addition, the compression technique had no impact on model convergence or test error (Adam: baseline:18.1% vs compression:18.3%). As shown in Fig. 3,

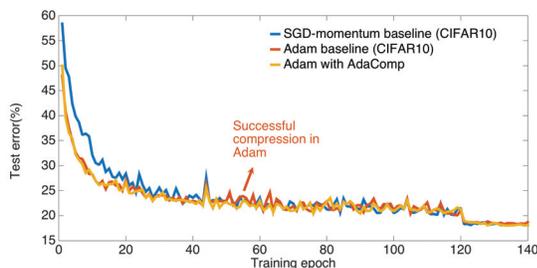


Figure 3: This work (AdaComp) achieves similar effective compression rates ($40\times$ for CONV layers and $200\times$ for FC/LSTM layers) with Adam, and had no impact on convergence or test error (Adam: baseline:18.1% vs compression:18.3%). In comparison to SGD, Adam exhibited faster initial convergence but similar final accuracy.

in comparison to SGD, as expected, Adam exhibited faster initial convergence but similar final accuracy. Intuitively, this result is consistent with the AdaComp algorithm which should be agnostic to the optimizer used (SGD with momentum vs. Adam vs. rmsprop) - with a detailed analysis presented in the next section.

Overall, our experimental results indicate that the AdaComp scheme is remarkably robust across application domains, layer types, learner numbers, and the choice of the optimizer. For the above L_T choices of 50 and 500, the AdaComp algorithm typically selects only up to 5 elements within each bin (through sparsity). For L_T sizes <64 , a sparse-indexed representation of 8-bits could be used effectively, while 16-bits of representation would be needed for larger L_T sizes (up to 16K elements) - where 2-bits (out of 8 or 16) would be used to represent the ternarized data values. Therefore, in comparison with traditional 32-bit floating-point representations, the AdaComp scheme achieves an excellent **Effective Compression Rate** of $40\times$ for convolutional layers and $200\times$ for fully connected and recurrent layers.

Discussions

Robustness of the AdaComp Technique

To understand the robustness of the different Residual Gradient compression schemes, 3 different methods are compared in Fig. 4 - Dryden (Dryden et al. 2016), Local Selection (LS), and this work (AdaComp). The LS technique refers to a scheme similar to AdaComp’s local selection scheme, but without applying a soft-threshold to self-adjust the compression rate. This allows us to evaluate the importance of the self-adjustable nature of AdaComp. For each compression scheme, the CIFAR10-CNN model is trained using SGD with momentum ($=0.9$) and varying L_T (and hence compression rates), reporting the final test error after 140 epochs. The 3 convolutional layers and the fully connected layer are all compressed at the same rate. As shown in Fig. 4, when the compression rate is less than 250, all methods achieve test errors close to (or slightly above) the

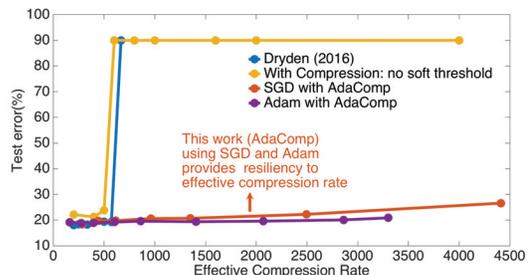


Figure 4: Comparing CIFAR10 test-errors vs. effective compression rates for 3 different schemes - Dryden’s method, Local Selection (LS) and the AdaComp technique trained by SGD. AdaComp is remarkably resilient to the compression rate while high compression rates cause LS and Dryden’s schemes to diverge. AdaComp is also tested using Adam and exhibits even higher resiliency.

baseline. While the test errors for Dryden’s method and LS increase significantly as the effective compression rate increases, the AdaComp method is remarkably robust to ultra-high compression rates (reaching only 22% test-errors for compression rates exceeding $2000\times$). We also show that this result extends to other optimizers including Adam - which shows even higher resilience (when compared to SGD) at high compression rates.

To understand why the LS scheme fails when the compression rate is high, we focus on experiments where the FC layer alone is compressed using LS. In Fig. 5, we plot the value of the 95 percentile of the gradient (dW) and the Residual Gradient (RG) during the training process. When the compression rate is small (e.g. LS with $L_T=200$), the magnitude of RG and dW is stable during training, leading to successful convergence with test errors (17.84%) close to the baseline. However, as the compression rate increases (e.g., LS with $L_T=300$), the magnitude of RG and dW appears to increase exponentially, resulting in complete divergence. This exponential increase can be understood as a positive feedback effect - where an increase in RG results in higher training error, leading to further increase in dW . Since dW is accumulated into RG, this feedback loop further expedites the growth of RG. As a result, both RG and dW grow exponentially over epochs.

The key factor that makes the AdaComp technique robust is the self-adjustable threshold. The positive feedback in LS occurs because insufficient number of gradients are sent after each mini-batch. The difficult challenge with the LS and Dryden schemes is to find which and how many gradients need to be sent - since this number can be layer, network and hyper-parameter dependent. Recall that the AdaComp scheme sends a few additional residual gradients close to the local maximum in each bin - and can therefore automatically adapt to the number of important gradients in that set. Even if the compression rate is high, soft-threshold avoids the exponential increase in RG and dW - by being adaptive to the number of gradients being sent. This is demonstrated in Fig. 5, where RG for AdaComp with $L_T=5000$ slightly increases

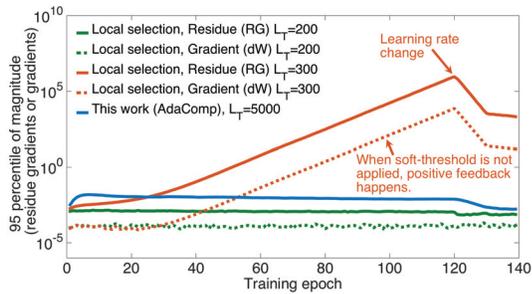


Figure 5: Comparison of the magnitude of Residual Gradients (RG) for the local selection scheme (without adaptive soft-threshold) and the proposed AdaComp scheme. RG values are very sensitive to L_T when local selection is used: larger L_T (corresponding to higher compression rate) causes exponential increases in RG and results in model divergence. In comparison, AdaComp is very resilient to high L_T values.

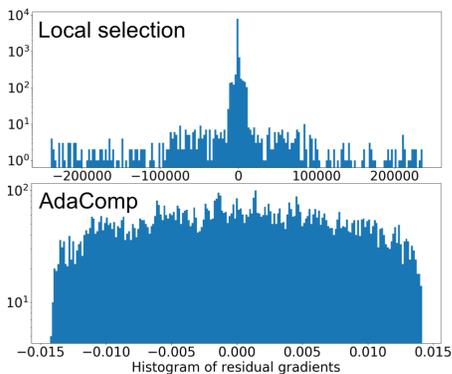


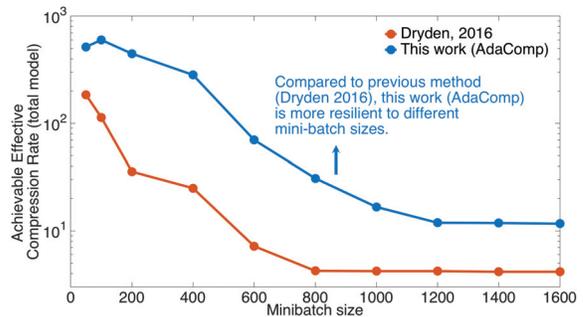
Figure 6: Comparing histograms of the Residual Gradients (RG) at epoch 120: Local Selection (LS) technique (top) and the AdaComp technique (bottom). AdaComp reduces RG by many orders of magnitude in comparison to LS.

in the beginning, but stabilizes after that. Note that the compression rate using AdaComp ($L_T=5000$) is $\sim 2000\times$, which is much higher than $\sim 600\times$ obtained with the LS scheme ($L_T=300$).

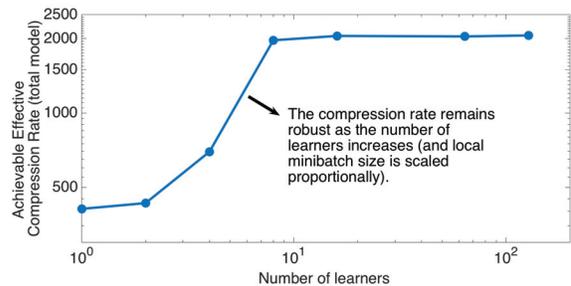
To illustrate how the self-adjustable threshold impacts the magnitude of RG, we plot the histogram of RG for LS and AdaComp at epoch 120 in Fig. 6. For the LS scheme, only a finite number of gradients are sent, increasing the RG of the remaining gradients. Therefore, it is observed that RG exponentially increases (ranging from -240K to 239K), shaping the histogram to have a very long tail at both ends. However for the AdaComp scheme, the gradients with large RG are all sent altogether, drastically shortening the tails. Therefore, RG for AdaComp does not increase over epochs.

Impact of mini-batch size and number of learners

Empirically, as shown in Fig. 7(a), we observe that increasing the mini-batch size reduces the achievable compression rate (while preserving model fidelity); this is true for previous work (Dryden et al. 2016) as well as the AdaComp



(a) Compression rate comparisons between previous work (Dryden et al. 2016) and this work (AdaComp) for different minibatch sizes.



(b) Achievable compression rate for AdaComp with varying number of learners. Super-minibatch size (across learners) = 128.

Figure 7: Impact of mini-batch size and the number of learners on the compression rate for CIFAR10-CNN. Compared to previous work (Dryden et al. 2016), AdaComp is more resilient to large minibatch size (a) and shows higher compression rate for large number of learners (b) - while maintaining test error degradation within the acceptable range ($< 1\%$).

compression scheme. Recall that the advantage of AdaComp over Dryden’s scheme is its ability to locally sample the residue vector and thereby effectively capture high activities in the input features. When the mini-batch size increases, we expect the input features at every layer to see higher activity. This causes the compression rate for Dryden’s technique to suffer dramatically, because relatively small residues (or gradients) now become increasingly important. In contrast, AdaComp locally selects elements, so while the compression rate does degrade, it still does a far better job at capturing all of the important residues. This results in a $\sim 5 - 10\times$ improved compression rate for AdaComp (Fig. 7(a)).

Consistent with the explanation above, we also observe (in Fig. 7(b)) that the compression rate for CIFAR10-CNN dramatically scales with the number of learners in the distributed system (while proportionally reducing the mini-batch size per learner). With more learners, each learner sees a smaller local mini-batch size and therefore compression rate is enhanced due to lower feature activity.

Conclusions

A new algorithm, Adaptive Residual Gradient Compression (**AdaComp**), for compressing gradients in distributed training of DL models, has been presented. The key insight behind AdaComp is that it is critical to consider both input feature activities as well as accumulated residual gradients for maximum compression. This is accomplished very effectively through a local sampling scheme combined with an adjustable (soft) threshold, which automatically handles variations in layers, mini-batches, epochs, optimizers, and distributed system parameters (i.e., number of learners). Unlike previously published compression schemes, our technique only needs 1 new hyper-parameter, is more robust, and allows us to simultaneously compress different types of layers in a deep neural network. Exploiting both sparsity and quantization, our results demonstrate a significant improvement in end-to-end compression rates: $\sim 200\times$ for fully-connected and recurrent layers, and $\sim 40\times$ for convolutional layers; all without any noticeable degradation in test accuracy. These compression techniques will be foundational as we move towards an era where the communication bottlenecks in distributed systems get exacerbated due to the availability of specialized high-performance hardware for DL training.

Acknowledgments

The authors would like to thank Naigang Wang, Vijayalakshmi Srinivasan, Swagath Venkataramani, Prithvi Narayanan, and I-Hsin Chung for helpful discussions and supports. This research was supported by IBM Research AI, IBM SoftLayer, and IBM Cognitive Computing Cluster (CCC).

References

- Chilimbi, T. M.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, 571–582.
- Cho, M.; Finkler, U.; Kumar, S.; Kung, D.; Saxena, V.; and Sreedhar, D. 2017. Powerai ddl. *arXiv preprint arXiv:1708.02188*.
- Courbariaux, M.; Bengio, Y.; and David, J.-P. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*.
- Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.; Senior, A.; Tucker, P.; Yang, K.; Le, Q. V.; et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*, 1223–1231.
- Dryden, N.; Jacobs, S. A.; Moon, T.; and Van Essen, B. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, 1–8. IEEE Press.
- Durant, L.; Giroux, O.; Harris, M.; and Stam, N. 2017. Inside volta: The worlds most advanced data center gpu. In <https://devblogs.nvidia.com/parallelforall/inside-volta/>.
- Goyal, P.; Dollár, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; and He, K. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.
- Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; and Narayanan, P. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 1737–1746.
- Gupta, S.; Zhang, W.; and Wang, F. 2016. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, 171–180. IEEE.
- Han, S.; Mao, H.; and Dally, W. J. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.
- He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.
- Ho, Q.; Cipar, J.; Cui, H.; Lee, S.; Kim, J. K.; Gibbons, P. B.; Gibson, G. A.; Ganger, G.; and Xing, E. P. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, 1223–1231.
- Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; and Darrell, T. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, 675–678. ACM.
- Karpathy, A. 2015. Multi-layer recurrent neural networks (lstm, gru, rnn) for character-level language models in torch. <https://github.com/karpathy/char-rnn>.
- Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- Krizhevsky, A., and Hinton, G. 2009. Learning multiple layers of features from tiny images.
- Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.
- LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11):2278–2324.
- Lian, X.; Huang, Y.; Li, Y.; and Liu, J. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, 2737–2745.
- Luehr, N. 2016. Fast multi-gpu collectives with nccl.
- Merriman, C. 2017. Google announces tpu 2.0 with 180 teraflop max out for ai acceleration. In <https://www.theinquirer.net/inquirer/news/3010365/google-announces-tpu-20-with-180-teraflop-max-out-for-ai-acceleration>.
- Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; and Kautz,

- J. 2016. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*.
- Nair, R., and Gupta, S. 2017. Wildfire! approximate synchronization of parameters in distributed deep learning. In *IBM Journal of Research and Development*. IBM.
- Seide, F.; Fu, H.; Droppo, J.; Li, G.; and Yu, D. 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*.
- Strom, N. 2015. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.
- van den Berg, E.; Ramabhadran, B.; and Picheny, M. 2017. Training variance and performance evaluation of neural networks in speech. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, 2287–2291. IEEE.
- Wen, W.; Xu, C.; Yan, F.; Wu, C.; Wang, Y.; Chen, Y.; and Li, H. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878*.
- Wu, Y.; Schuster, M.; Chen, Z.; Le, Q. V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. 2016. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.
- Xiong, W.; Droppo, J.; Huang, X.; Seide, F.; Seltzer, M.; Stolcke, A.; Yu, D.; and Zweig, G. 2017. The microsoft 2016 conversational speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, 5255–5259. IEEE.