Large-Batch Training for LSTM and Beyond

Yang You[2], Jonathan Hseu[1], Chris Ying[1], James Demmel[2], Kurt Keutzer[2], Cho-Jui Hsieh[3]
Google Brain[1], UC Berkeley[2], UCLA[3]

ABSTRACT

Large-batch training approaches have enabled researchers to utilize distributed processing and greatly accelerate deep neural networks training. However, there are three problems in current large-batch research: (1) Although RNN approaches like LSTM have been widely used in many applications, current large-batch research is principally focused on CNNs. (2) Even for CNNs, there is no automated technique for extending the batch size beyond 8K. (3) To keep the variance in the gradient expectation constant, theory suggests that a Sqrt Scaling scheme should be used in large-batch training. Unfortunately, there are not many successful applications. In this paper, we propose Dynamic Adaptive-Tuning Engine (DATE) for better large-batch training. DATE achieves a 5.3x average speedup over the baselines for four LSTM-based applications on the same hardware. We finish the ImageNet training with ResNet-50 in two minutes on 1024 v3 TPUs (76.7% top-1 accuracy), which is the fastest version as of June of 2019.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**;

KEYWORDS

large-batch training, distributed computing, neural networks

ACM Reference Format:
Yang You[2], Jonathan Hseu[1], Chris Ying[1], James Demmel[2], Kurt Keutzer[2], Cho-Jui Hsieh[3]. 2019. Large-Batch Training for LSTM and Beyond. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356137

1 INTRODUCTION

Speeding up Deep Neural Network (DNN) training is important because it can improve the productivity of machine learning researchers and developers. Since the acceleration of training through exploiting model parallelism is limited, current research principally focuses on data parallelism. Specifically, a large-batch training approach has enabled us to successfully exploit large-scale distributed processing [1, 13, 20, 27, 35, 47, 49]. For example, by scaling the batch size from 256 to 32K [48], researchers are able to reduce the training time of ResNet50/ImageNet from 29 hours [14] to 2.2 minutes [47]. However, there are three problems with current large-batch approaches:

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356137

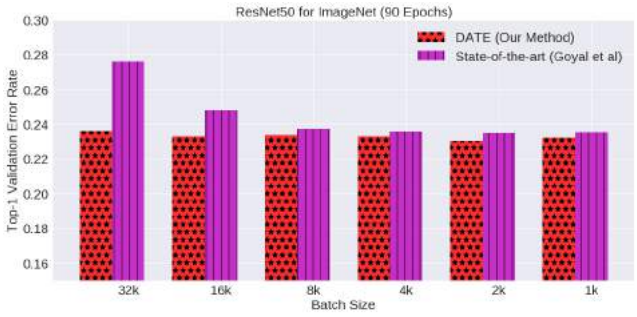


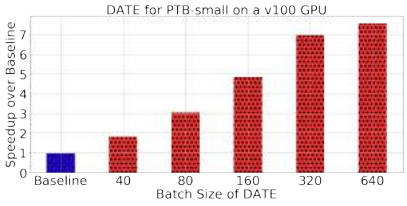
Figure 1: DATE achieves the constant accuracy when we scale up the batch size without tuning the parameters (learning rate, weight decay, and momentum). DATE works better than previous large-batch techniques [13].

- Although RNN techniques like LSTM [15] have been widely used, the current large-batch study is mostly focused on CNN applications. On the other hand, adaptive solvers like Adam [23] do not beat well-tuned Momentum SGD for ImageNet training. We want to evaluate Adam for large-batch LSTM training.
- Even for CNN applications, significant hyper-parameter tuning is required to increase the batch size beyond 8K with no loss in accuracy. For batch sizes lower than 8K, linear scaling usually works well for most applications. However, for batch sizes beyond 8K, even solvers like LARS [48] require users to manually tune the hyper-parameters (including learning rate, warmup, weight decay, and momentum).
- Prior successful large-batch training relies on a linear scaling scheme [13]. However, to keep the variance in the gradient expectation constant, theory [24] suggests a Sqrt Scaling scheme should be used. Currently there are not many successful large-batch applications using Sqrt Scaling. Our goal is to show how to make Sqrt Scaling effective in practice.

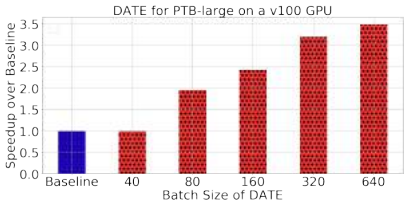
To solve these problems, we propose a new approach called Dynamic Adaptive-Tuning Engine (DATE). DATE enables Sqrt Scaling to perform well in practice and as a result we achieve a much better performance than the previous Linear Scaling learning rate scheme. DATE also includes other techniques like efficient warming-up, auto LR (learning rate) decay, and runtime LR adaptive updating. For the GNMT application (Seq2Seq) with LSTM, we are able to scale the batch size by a factor of 16 without losing accuracy and without tuning the hyper-parameters mentioned above. For the PTB dataset with LSTM, we are able to scale the batch size by a factor of 32 without losing accuracy and without tuning the hyper-parameters. Beyond RNN applications, we also successfully applied DATE in ImageNet training with ResNet-50. DATE is able to achieve a constant accuracy when we scale the batch size to 32K. DATE works better than previous large-batch tuning techniques (Figure 1). We also provide some theoretical explanations for the key techniques

of DATE. DATE achieves a 5.3× average speedup over the baselines for 4 LSTM-based applications on the same hardware (Figure 2). DATE achieves 119% weak scaling efficiency (super-linear speedup) as we increase the number of TPUv2 chips from 8 to 512. DATE also achieves a good scaling efficiency for several state-of-the-art deep learning models on latest TPUv3 chips (Figure 4 - Figure 9). We finish the ImageNet training with ResNet-50 in two minutes on 1024 v3 TPUs (76.7% top-1 accuracy), which is the fastest version with 76.5+% accuracy as of June of 2019. We also did not tune the hyper-parameters.

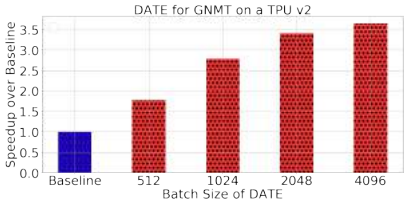
2.1



2.2

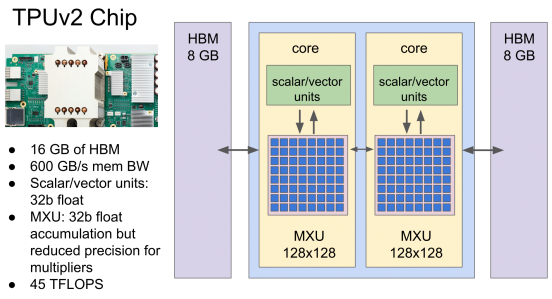


2.3

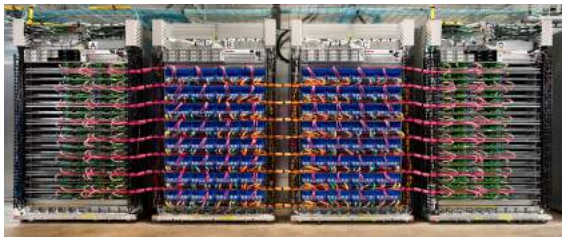


2.4

Figure 2: The speedups over the baseline, which are achieved by DATE with different batch sizes on the same hardware. The leftmost bar is the baseline.



3.1 Tensor Core (figure credit: Jeff Dean's NIPS'17 talk)



3.2 TPU v2 Pod (figure credit: Jeff Dean's NIPS'17 talk)

Figure 3: Tensor Core is the most basic building block of the TPU system. A TPU-v2 Chip has two tensor cores. 16-by-16 TPU-v2 chips connected via 2D torus is a TPU-v2 Pod.

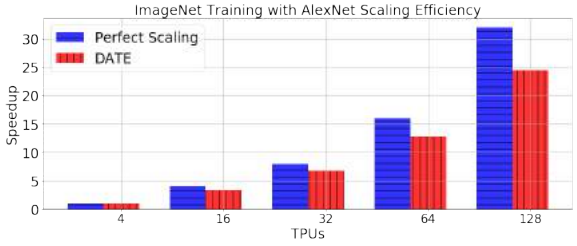


Figure 4: 76.66% weak scaling efficiency.

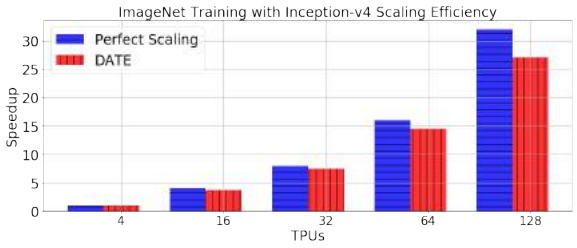


Figure 5: 84.76% weak scaling efficiency.

2 BACKGROUND AND RELATED WORK

2.1 Data-Parallelism Mini-Batch SGD

Let us refer to w as the DNN weights, X as the training data, n as the number of samples in X, and Y as the labels of X. Let us also denote x_i as a sample of X and $l(x_i, y_i, w)$ as the loss computed by x_i and its label y_i ($i \in \{1, 2, ..., n\}$). A typical loss function is cross-entropy [12]. The goal of DNN training is to minimize the loss defined in Equation (1).

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} l(x_i, y_i, w) \quad (1)$$

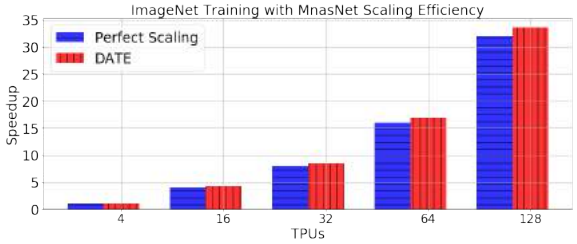


Figure 6: 100.05% percent weak scaling efficiency.

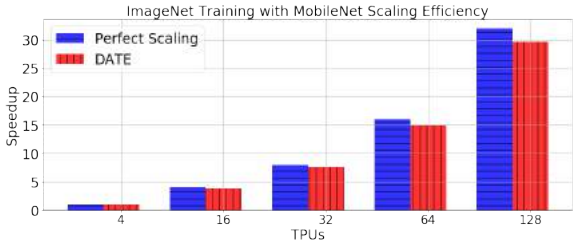


Figure 7: 92.82% percent weak scaling efficiency.

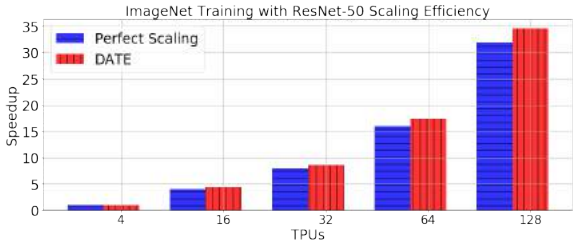


Figure 8: 100.08% weak scaling efficiency.

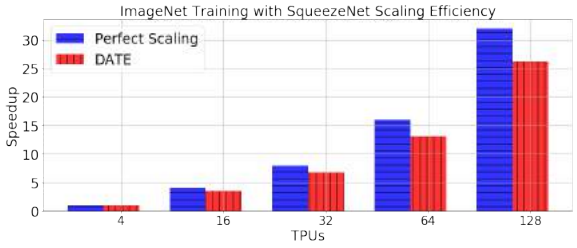


Figure 9: 81.89% weak scaling efficiency.

At the t-th iteration, we use forward and backward propagation to get the gradients of weights based on the loss. Then we use the gradients to update the weights, which is shown in Equation (2):

$$w_{t+1} = w_t - \eta \nabla l(x_i, y_i, w) \tag{2}$$

where η is the learning rate (LR). This method is called as Stochastic Gradient Descent (SGD). Usually, people do not use a single sample to compute the loss and the gradients. Instead, they use a batch of samples at each iteration. Let us refer to the batch of sample at t-th iteration as B_t. The size of B_t is b. Then we update the weights based on Equation (3).

$$w_{t+1} = w_t - \frac{\eta}{b} \sum_{x \in B_t} \nabla l(x, y, w) \tag{3}$$

This method is called as Mini-Batch SGD. To simplify the notation, we define the gradient estimator as $\nabla w_t := \frac{1}{b} \sum_{x \in B_t} \nabla l(x, y, w)$ and the updating rule in Equation (4) can be denoted as

$$w_{t+1} = w_t - \eta \nabla w_t \tag{4}$$

where we use the gradients ∇w_t to update the weights w_t.

2.2 Model Parallelism

2.2.1 Limitation. Due to the data dependency between different layers in forward propagation and backward propagation, the developers can not parallelize across different layers. Thus, model parallelism requires the developers to parallelize within each layer. In this way, a wider neural network provides a higher parallelism. However, modern deep learning researchers prefer deep neural networks to wide neural networks. The reason is that, given the fixed number of parameters, the deep model can achieve better results than the wide model. For example, a typical layer of BERT (the state-of-the-art NLP model) [8] is a 1024-by-1024 matrix. The widest layer of BERT is a 1024-by-4096 matrix. If we disable the data parallelism (i.e. set batch size as one), we can not even make full use of the computational power of one GPU or CPU chip to accelerate a 1024x1024x1024 matrix multiply.

2.2.2 Work with Data Parallelism. For applications with an extremely wide model or a large single sample, model parallelism can work with data parallelism(e.g. Mesh-tensorflow [38]). Assume we have P nodes in this situation, we partition these P nodes into G groups (e.g. G=256, P=1024). We use model-parallelism within each group and data-parallelism across different groups. In this paper, $G = P$ works well for all of our applications. Thus, we only enable model parallelism within a single node. We focus on maximizing the data parallelism (i.e. maximizing the batch size).

2.3 Large-Batch Training Difficulty

Increasing the batch size allows us to scale to more nodes without reducing the workload on each node. On a modern architecture like TPUs, reducing the workload often leads to a lower efficiency. However, when we increase the batch size after a certain point (e.g. 1024) without a careful optimization scheme, the algorithm usually suffers from slow convergence. The test accuracy of the converged solution becomes significantly lower than the baseline [13, 16, 22, 28]. Keskar et al [22] suggested that there is a generalization problem when training with large-batches. The algorithm usually converges to sharper local minimums, so the test accuracy will be much lower even when the training accuracy remains high. For small batches, the training accuracy and test accuracy are closer to each other. Hoffer et al [16] and Li et al [28] suggests that training longer will help the algorithm to generalize better and keep the accuracy higher. On the other hand, Goyal et al [13] can scale the batch size to 8K without losing accuracy by using a LR scheduling technique.

2.4 Large Batch Training Techniques

When we increase the batch size (B), we need to increase the peak LR to prevent losing accuracy [13]. There are two rules for increasing the peak LR:

Sqrt Scaling Rule [24]. When we increase the batch size by k times, we should increase the LR by $\sqrt{k}$ times to keep the variance of the gradient estimator constant.

Linear Scaling Rule [24]: When we increase the batch size by k times, we should increase the LR by k times based on the assumption that $\nabla l(x, y, w_t) \approx \nabla l(x, y, w_{t+j})$, where $j < B$.

Warmup Scheme [13] Usually, under linear scaling rule, $k\eta$ is extremely large, which may make the algorithm diverge at the beginning. Therefore, people set the initial LR η to a small value and increase it gradually to $k\eta$ in a few epochs (e.g. 5 or 10). This method is called **Gradual Warmup Scheme**. There is another method called as **Constant Warmup Scheme**, which uses a constant small LR during the first a few epochs. Constant warmup scheme works efficiently for prototyping object detection and segmentation [10], [29]. Goyal et al. [13] showed that gradual warmup performs better than constant warmup for ResNet-50 training. Bottou et al. [3] showed that there should be an upper bound of LR regardless of batch size. Our experimental results are in line with these findings. Chen et al. [4] also used linear scaling LR scheme in their experiments when they increase the batch size from 1600 to 6400. However, they did not show the accuracy of the small-batch baseline.

Krizhevsky [24] reported 1 percent loss in accuracy when he increased the the batch size from 128 to 1024. He achieved 56.7% accuracy for using a batch size of 1024 in Imagenet training with AlexNet. Iandola et al. [19] also scaled the batch size to 1K for AlexNet and GoogLeNet. Li [27] used a batch size of 5120 for ResNet-101 to train Imagenet dataset on 160 GPUs. Goyal et al. [13] scaled the batch size to 8K for ImageNet training with ResNet-50. They used data parallelism to process ResNet-50 model on 256 NVIDIA P100 GPUs (equal to 32 NVIDIA DGX-1 stations). The LARS algorithm [48] was proposed to scale the batch size to 32K for ImageNet training. The LARS algorithm was implemented on 2048 Intel KNL chips and finished the ImageNet/ResNet-50 training in 14 minutes [49]. The LARS algorithm was also implemented on TPU-v3 Pod to finish the ImageNet/ResNet-50 training in 2.2 minutes [47]. Codreanu et al. [5] scaled DNN training on 1024 SkyLake CPUs and finished ImageNet training with ResNet50 in 44 minutes. Akiba et al. [1] scaled the batch size to 32K and finished the ImageNet training with ResNet50 in 15 minutes. However, their baseline's accuracy was missing. Jia et al. [20] combined LARS algorithm with mixed-precision training [33] and finished the ImageNet training with ResNet50 in 8.6 minutes. The other related directions include K-FAC [32] and dynamic batch size [7, 39]. However, there is no auto-tuning technique for CNN with a batch size beyond 8K. Moreover, there is no existing autotuning work for large-batch RNN applications like LSTM. The hand-tuned process is painful and time-consuming. In this paper, we study the large-batch algorithms for LSTM applications. We propose an auto-tuning framework for large batches called DATE (Dynamic Adaptive-Tuning Engine), which not only performs well for LSTM applications, but also performs well for CNN applications.

2.5 Distributed TPU Systems (TPU Pod)

TPU (Tensor Processing Unit) [21] is a powerful architecture targeting machine learning applications. TPU v1 is focused on inference tasks like speech and image recognition. TPU v2 and TPU v3 provide the high floating-point performance for the training process. TensorCore is the most basic building block of the TPU system. A TPU-v2 Chip consists of two compute nodes called TensorNodes. Each TensorNode can be thought of as a core in a CPU. The TensorNode has a dense compute unit called TensorCore (Figure 3) and a sparse compute unit. The MXU (Matrix Multiplier Unit) within each TensorCore features a drastically different architecture than typical CPUs and GPUs, called a systolic array. In matrix multiply, a MXU reuses the inputs several times as part of producing the output. Each value can be inputted once, but used for various operations without being moved back to a register. Arithmetic Logic Units (ALUs) are energy-efficient because they are only connected by adjacent wires. The design is simplified because the ALUs perform only multiplications and additions in the constant patterns. The MXU design is called systolic because the data flows in waves through the chip, which is the same way that the heart pumps blood. The specific type of systolic array in the MXU is optimized for reducing power and improving area efficiency in matrix multiply (i.e. the dominate operation in deep learning). It is not optimized for the general-purpose computing. This brings an engineering trade-off: the higher operation density and energy efficiency comes from the reduced control, registers and operational flexibility.

A group of 16-by-16 TPU chips connected via 2D torus is a TPU Pod (Figure 3). The Cloud TPU-v2 server includes four TPU chips. Each Cloud TPU-v2 server provides 180 TFLOPS and 64 GB High Bandwidth Memory (HBM). A TPU-v2 Pod is made up of 64 TPU-v2 servers. In theory, a TPU-v2 Pod can provide 11.5 petaflops performance (16/32 bit mixed precision) and 4 terabytes of HBM memory. TPU's 16-bit format is called bfloat16, which is different form IEEE 754 16 bit (IEEE 754 has 1 sign bit, 5 exponent bits, and 10 mantissa bits. When combined with the 1 "hidden bit", there are effectively 11 mantissa bits. bfloat16 has 1 sign bit, 8 exponent bits, and 7 mantissa bits).

3 DATE: DYNAMIC ADAPTIVE-TUNING ENGINE (FOR LARGE-BATCH TRAINING)

In this section, we introduce DATE framework for large-batch training. We present the main features of DATE one-by-one.

3.1 Linear Epoch Gradual Warmup (LEGW)

The warmup technique has been successfully applied in the CNN applications [13, 48]. However, before this paper, most of the RNN implementations did not use warmup techniques. On the other hand, warmup has become an additional parameter that requires developers to tune, which further increases the efforts of DNN system implementation. To make things worse, large-batch training usually converges to a sharp local minimum, so a tiny change in the hyper parameters may have a significant influence on the test accuracy (Table 1). We propose the Linear-Epoch Gradual Warmup (LEGW or Leg-Warmup) scheme. When we increase the batch size by k times, we also increase the warmup epochs by k times. The intuition is that larger batch size usually needs a large LR. However,

Table 1: Large-batch training is a sharp minimal problem. It is easy to miss the global minimum. Tuning the hyper-parameters requires a lot of efforts. In this example (ImageNet/ResNet-50 training by LARS solver), we only slightly changed the LR, the accuracy dropped below the target 76.3% accuracy.

Batch Size	Init LR	Warmup	Epochs	Top-1 Test Accuracy
2048	9.94	0.6875 epochs	90	76.97%
2048	10.0	0.6875 epochs	90	75.59%

a larger LR may make the training algorithm more easily diverge because the gradient changes dramatically in the beginning of neural network training. We use a longer warmup to avoid the divergence of larger LR. Linear epoch warmup means fixing the warmup iterations as we increase the batch size, which helps us to stabilize the chaotic early-learning state. It worth noting the users do not need to change anything. DATE will work like a black box.

3.2 Explanation of LEGW

In general, it is hard to prove why a specific learning rate schedule works. However, some experimental findings on the change of local Lipschitz constant during iterations partially explain why LEGW works better than previous methods.

Consider the update along the gradient direction $g = \nabla f(x)$. Assume the update is $x \leftarrow x - \eta g$, the question is: how to choose learning rate η? One classical idea is to form a second order approximation around current solution x. $f(x + \Delta) \approx$

$$\tilde{f}(x + \Delta) := f(x) + \Delta^T \nabla f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x)\Delta, \tag{5}$$

and then find Δ to minimize the approximation function. If we assume Δ is in the form of $-\eta g$ and Hessian is positive definite along the direction of g ($g^T \nabla^2 f(x) g > 0$), then the optimal η^* is

$$\arg\min_{\eta} \tilde{f}(x - \eta g) = \frac{1}{g^T \nabla^2 f(x) g / \|g\|^2} := \frac{1}{L(x, g)}.$$

Therefore, ideally the learning rate should be inversely proportional to $L(x, g)$. Moreover, it is known [11] that the update $-\eta g$ will decrease the objective function in a small compact region S if $\eta < \min_{x' \in S} \frac{1}{L(x', g)}$. The optimal learning rate is also called the local Lipchitz constant along the gradient direction, and $L(x, g)$ can be viewed as its approximation. In Figure 11, we plot the values of $L(x, g)$ for all the iterations in MNIST training with LSTM. It is hard to compute $L(x, g)$ exactly since $\nabla^2 f(x)$ involves all the training samples. So we approximate it using a small batch and compute the Hessian-vector product by finite difference. For the same reason it is hard to apply a second order method exactly, but the plots in the figures show an interesting phenomenon that explains why linear warmup works. We observe that the value of $L(x, g)$ usually has a peak in the early iterations, implying a smaller step size should be used in the beginning (which implies warmup is needed). Furthermore, the peak tends to shift toward right (almost linearly) as batch size grows. This intuitively explains our linear warm-up strategy: when batch size increases, the warm up should be longer to cover the "peak region".

3.3 Sqrt Learning Rate Scaling

To keep the variance in the gradient expectation constant, theory [24] suggests that Sqrt Scaling should be used in large-batch training. In practice, however, researchers observe that Linear Scaling performs much better than Sqrt Scaling [13, 24, 27, 49]. The constant-epoch warmup scheme was used together with Linear Scaling in previous applications. For example, Goyal et al. [13] manually set the warmup length as five epochs. The efficiency of Linear Scaling only works up to 8K batch size, although researchers are able to scale the batch size to 32K with signifiant hyper-parameter tuning (tuning learning rate, warmup, weight decay and momentum for different batch sizes). With LEGW, the Sqrt Scaling scheme can work well in practice, and is able to match the expectations of the theoretical analysis. The results are shown in Section 4. We assume that Sqrt Scaling is built on top of LEGW. The reason is that the constant number of warmup steps performs better with the constant variance.

3.4 Roller Coaster Schedule

The adaptive solvers like AdaGrad [9] use the sum of all historical gradients to decay the learning rate (e.g. $\frac{\eta}{\sqrt{\sum_t g_t \odot g_t}}$), which is easily out of control at runtime because of the vanishing and exploding gradient problems [2]. Thus, in the real-world systems, the state-of-the-art approach uses a manual way to reduce the learning rate. For example, in ResNet-50 training, the authors manually reduce the learning rate by a factor of 10 at 30th, 60th, and 80th epoch [13]. In ResNet-101 training, the authors manually reduce the LR by factor of 10 at 50th epoch and 100th epoch [27]. The way of manual tuning makes the decay scheme too complicated to be used by amateurs. In this paper, we use an automatic way to decay the learning rate. Let us assume t is the current number of iterations we have finished and T is the total number of iterations we need to finish. We use a roller-coaster way to decay the learning rate after the warmup stage:

$$\eta = max\{\frac{(T - t)}{(1 - w/E) \times T} \times \sqrt{\frac{B}{B_0}}\eta_0, \hat{\eta}\}$$

where $\hat{\eta}$ is the lower bound of the LR. There is no need to tune $\hat{\eta}$, we set 10^{-6} as the default. This way maintains a stable decay all the way from the post-warmup point to the final stage, which helps the algorithm converges to the minimum. In our experiments, this approach is consistently better than the polynomial decay (power=0.5, 1.0, 2.0) in more than 10 repeated runs.

3.5 Dynamic Per-Layer Stabilized Learning

One successful idea of the LARS solver [48] is to use the ratio between the L2 norm of the weight and the L2 norm of the gradient at each iteration to adaptively update the LR. In this way, different layers will have different speeds of learning. However, one weakness of LARS is that it requires the users to build on top of a momentum solver. The users need to manually input the hyper-parameters like LR and weight decay. If we switch the kernel from momentum to RMSprop [43], we observe LARS does not converge in some situations. We use MNIST with LeNet to illustrate the idea here. For small-batch baseline (batch size = 256), RMSprop achieves 1% testing error rate (i.e. 99% testing accuracy). When we scale the batch

size to 8K, RMSprop only achieves 2.8% error rate. After adding LARS correction to RMSprop, the accuracy becomes even worse. It only achieves a 21.8% error rate. For these adaptive solvers, they already gave a larger learning rate to the weights with a smaller historical gradient. LARS gives them an even larger learning rate (Layer 4 in Figure 12). For the slowly-learning layer (i.e. with a small learning rate), LARS gives them an even smaller learning rate (Layer 5 in Figure 12). Thus, we correct LARS by a dynamic upper limit and a dynamic lower limit for LARS ratio at runtime. We also removed the weight decay from LARS. In this way, we can reduce the error rate from 21.8% to 1.0% for the 8K batch size.

3.6 Minimal Tuning Effort

By using DATE, the users do not need to manually tune any hyper-parameters for scaling batch size. For example, the users only need to input the hyper parameters of a baseline (e.g. batch size = 256) to the framework. Then the system can automatically scale to the batch size the users want. The users can treat DATE as a black-box large-batch training tool. Here, we use a specific example (i.e. MNIST training by LeNet) to illustrate the details inside DATE framework.

First, we explain the way used in state-of-the-art deep learning system [13], which is a manually-tuned approach. The LeNet/MNIST training has 30 epochs. The baseline's batch size, number of iterations, learning rate, and warmup epochs are B_0=1024, I_0=1755, η_0=1.25×10^{-3}, and w_0=2, respectively (notations are explained in Algorithm 1). The baseline works in this way:

- In the initial two epochs (117 iterations), the users gradually increase LR from 0 to 1.25×10^{-3} in a linear way.
- In the (2, 10] epochs (469 iterations), the constant learning rate of 1.25×10^{-3} will be used.
- In the (10, 20] epochs (586 iterations), the constant learning rate of 1.25×10^{-4} will be used.
- In the (20, 27] epochs (410 iterations), the constant learning rate of 1.25×10^{-5} will be used.
- In the (27, 30] epochs (176 iterations), the constant learning rate of 1.25×10^{-6} will be used.

If the user scales up the batch size by k times, they need to increase the learning rate by k times. For example, the 8K batch size uses a peak learning rate of 0.01. In the same way, the users first warm up the LR in the first two epochs and then reduce the LR by multiplying it by 0.1 at 10th, 20th, and 27th epoch. The idea is illustrated in Figure 10.1. Some users may feel there are too many parameters to tune in this scheme. The users may need to decide in which epoch to decay the learning rate, and how much the learning rate should be reduced each time.

In DATE, the user only needs to input the baseline information. DATE automatically do the following for a batch size of 8K (input of B in Algorithm 1):

- In the initial 16 epochs (117 iterations), DATE gradually increases LR from 0 to 1.25×10^{-3}×$\sqrt{\frac{8192}{1024}}$ (line 6 of Algorithm 1). DATE also uses adaptive method to update LR at the runtime.
- From 16th epoch to 30th epoch (from 118th to 220th iteration), DATE uses a LR of max{$\frac{(220-t)}{(1-2/30)\times 220}$ × 1.25 × 10^{-3} × $\sqrt{\frac{8192}{1024}}$, 10^{-6}} at t-th iteration. DATE also uses adaptive method to update LR at the runtime.

The idea is illustrated in Figure 10.5. Figure 10 includes the main features of DATE framework and a comparison to the state-of-the-art approach. Algorithm 1 is an overview of DATE framework.

Algorithm 1: Framework of DATE

Input:
n labeled data points (x_i, y_i) for training;
Another k labeled data points $(\hat{x}_j, \hat{y}_j)$ for testing;
$i \in \{1, 2, \ldots, n\}$, $j \in \{1, 2, \ldots, k\}$;
A baseline with Batch Size B_0, learning rate η_0, warmup epochs w_0 and total number of iterations I_0;
A target large batch size B;

Output:
Trained Model of large batch B;
Test Accuracy of large batch B

1 The warmup epochs $w = \frac{Bw_0}{B_0}$
2 The number of iterations $I = \frac{B_0 I_0}{B}$
3 **for** $i \in 1 : I$ **do**
4 $E = \frac{iB}{n}$ (the current epoch)
5 **if** $E < w$ **then**
6 $\eta = \frac{E}{w}\sqrt{\frac{B}{B_0}}\eta_0$ or $(\frac{E}{w})^2\sqrt{\frac{B}{B_0}}\eta_0$
7 **else**
8 $\eta = max\{\frac{(I-i)}{(1-w/E)\times I} \times \sqrt{\frac{B}{B_0}}\eta_0, 10^{-6}\}$
9 L = {the number of layers}
10 **for** $j \in 1 : L$ **do**
11 w = {the weight of layer-j}
12 g = {the gradient of layer-j}
13 **if** $||g||_2 == 0$ *or* $||w||_2 == 0$ **then**
14 r = min{max{1.0, lower_limit}, upper_limit}
15 **else**
16 r = min{max{$\frac{||w||_2}{||g||_2}$, lower_limit}, upper_limit}
17 $\eta = r\eta$ (runtime correction)
18 apply_gradient_update(w, g, η) based on the optimizer (SGD, momentum, AdaGrad, or RMSProp)

4 EXPERIMENTAL RESULTS

In all the comparisons of this paper, different methods will use the same hardware and run the same number of epochs (i.e. the same number of floating point operations). We use several real-world applications to evaluate our approach. The models and datasets are shown in Table 2. Besides the LSTM applications, we also include the traditional CNN applications like MNIST/LeNet training and ImageNet/ResNet-50 training.

Table 2: The applications we used to evaluate our method.

Model	Dataset	Type	Samples	Metric & Reference
LeNet	MNIST	Small	60K/10K	99.2% accuracy[1]
1-layer LSTM	MNIST	Small	60K/10K	98.7% accuracy[2]
PTB-small	PTB	Medium	930K/82K	116 perplexity[3]
PTB-large	PTB	Medium	930K/82K	78 perplexity[4]
GNMT	wmt16	Large	3.5M/3K	21.8 BLEU[5]
ResNet50	ImageNet	Large	1.3M/5K	75.3% accuracy[6]

4.1 The LSTM applications

4.1.1 **Handwritten Digits Recognition for MNIST.** We use the MNIST dataset [26] to train a pure-LSTM model. We partition each image as 28-step input vectors. The dimension of each input vector is 28-by-1. Then we have a 128-by-28 transform layer before the LSTM layer, which means the actual LSTM input vector is 128-by-1. The hidden dimension of LSTM layer is 128. Thus the cell kernel of LSTM layer is a 256-by-512 matrix. The state-of-the-art single-layer LSTM achieved 97.27% accuracy for MNIST dataset [34, 44, 50]. After a careful model design, we achieved 98.7% accuracy in 25 epochs training. The baseline uses a momentum solver (momentum=0.9) and constant learning rate. The baseline's batch size is 128. Our goal is to scale the batch size to 8K without losing accuracy. A batch size over 8K on a V100 GPU or a TPU-v2 server will get no additional speedup, so we stop at 8K. The effect of DATE is shown in Figures 16, 17 and 18. These figures show that DATE is able to beat the comprehensive tuning solver and the Adam solver.

4.1.2 **Language Modeling for PTB Dataset.** The Penn Treebank (PTB) [31] dataset selected 2,499 stories from a three year Wall Street Journal (WSJ) collection of 98,732 stories for syntactic annotation. The vocabulary has 10,000 words. After word embedding, the input vector length is 200 and 1500 for PTB-small model and PTB-large model[7], respectively. The sequence length is 20 and 35 for PTB-small and PTB-large. Our LSTM model has two layers. The hidden dimensions of both these two layers are 200 for PTB-small and 1500 for PTB-large. For both layers, the LSTM Cell Kernel is an 400-by-800 matrix for PTB-small and 3000-by-6000 matrix for PTB-large. We use perplexity to evaluate the correctness of our LSTM model (a lower perplexity means a better result). After a 13-epoch training, PTB-small can achieve a perplexity[8] of 116. After a 55-epoch training, PTB-large can achieve a perplexity[9] of 78. For PTB-small, the baseline uses a momentum optimizer (momentum=0.9) and exponential learning rate decay. The model uses constant learning rate in the first seven epochs. Then the learning rate will be decayed by 0.4 after each epoch. For PTB-large, the baseline uses the LARS solver [48] and poly decay (power=2.0). The baseline's batch size is 20. Our goal is to scale the batch size to 640 without increasing perplexity. The effect of DATE is shown in Figure 16, which is able to beat the Adam solver. The batch size over 640 will lead to an out-of-memory error on a V100 GPU, so we stop at 640.

4.1.3 **Google Neural Machine Translation (GNMT).** GNMT or seq2seq [30, 46] is a state-of-the-art machine translation technique. We use WMT16 English-German translation dataset for training. The encoder and decoder are using shared embeddings. The encoder includes 4 LSTM layers. The hidden dimension is 1024. The first layer is bidirectional, the rest are undirectional. The residual connections start from 3rd layer. The decoder includes 4 unidirectional LSTM layers with hidden size 1024 and a fully-connected classifier. The residual connections start from 3rd layer. We use normalized Bahdanau attention (gnmt_v2 attention mechanism). We use BLEU score on newstest2014 dataset as the quality metric (higher is better). The BLEU score is reported by sacrebleu package. The baseline achieves a BLEU score[10] of 21.80. The effect of DATE is shown in Table 3 and Figure 16, which demonstrates that we can scale the batch size to 4K without losing accuracy. A batch size over 4K will lead to an out-of-memory error on a TPU, so we stop at 4K.

4.2 Compared to Adaptive Solvers

Our goal is to minimize the tuning effort for large-batch training. To evaluate this we need to pick an adaptive solver as a baseline for comparison. We fully evaluate a total of seven solvers: SGD [37], Momentum [36], Nesterov [40], Adagrad [9], RMSprop [12], Adam [23], Adadelta [51]. We pick Adam and Adadelta as the baseline for adaptive solvers because they do not require the users to input hyper-parameters. For MNIST and PTB datasets, we observe Adam performs much better than Adadelta (Figure 13). Moreover, Adam is able to beat the existing tuning techniques (Figure 15). Thus, we use Adam as the adaptive solver baseline for comparison.

The comparison between Adam and DATE is shown in Figure 16. We observe DATE performs better than Adam for PTB and GNMT applications in the same number of epochs. DATE's accuracy changes less than Adam when we scale up the batch size. It is worth noting that we carefully tuned the learning rate of Adam solver and made sure it gets the best performance. For MNIST application, the tuning space is {0.0001, 0.0002, 0.0003, ..., 0.0010}. For PTB application, the tuning space is {0.001, 0.002, 0.003, ..., 0.020} and {0.0001, 0.0002, 0.0003, ..., 0.0020}. For GNMT application, the tuning space is {0.001, 0.002, 0.003, ..., 0.020} and {0.0001, 0.0002, 0.0003, ..., 0.0020}. Figure 14 also shows that DATE performs better than the tuned Adam solver for PTB-large and GNMT applications. Therefore, we conclude that DATE is a better auto-tuning scheme compared to state-of-the-art approaches.

4.3 Comparison to Comprehensive Tuning

To prove the effectiveness of DATE, we make a comparison between DATE and the comprehensive tuning baseline for the largest batch size. For the MNIST dataset, since the model uses a constant learning rate for momentum solver. We comprehensively tune the learning rate and find only the range of [0.01, 0.16] is effective. After tuning the learning rate from 0.01 to 0.16, we observe that DATE's accuracy is higher than the best tuned version (Figure 17.1). For PTB dataset, the baseline uses the SGD optimizer. We comprehensively tune the initial learning rate for baseline and we find only the range from 0.1 to 1.6 is effective. Then we tune the learning rate within the effective range, the baseline's highest accuracy is still lower than DATE's accuracy (Figure 17.2). We also run the training algorithms long enough to make sure all of them are converged. For MNIST dataset, we increase the number of epochs from 25 to 100. For PTB dataset, we increase the number epochs from 13 to 50. Even when comprehensive turning versions are allowed to run longer, DATE

[1] https://github.com/tensorflow/models/tree/master/official/mnist
[2] https://medium.com/machine-learning-algorithms
[3] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py
[4] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py
[5] https://github.com/mlperf/training/tree/master/rnn_translator
[6] https://github.com/KaimingHe/deep-residual-networks
[7] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py
[8] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py
[9] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py
[10] https://github.com/mlperf/training/tree/master/rnn_translator

Table 3: By using DATE, we can scale the batch size of GNMT training from 256 to 4K without influencing the BLEU score. The baseline's BLEU score is 21.8. Since linear warmup epochs means fixed the warmup iterations, DATE sets the warmup iterations as 200.

Batch Size	Init LR	Warmup	Epochs	BLEU
256	$2^{-0.5}/10^3$	0.0145 epochs	2	22.7
512	$2^{0.0}/10^3$	0.0290 epochs	2	22.9
1024	$2^{0.5}/10^3$	0.0580 epochs	2	22.6
2048	$2^{1.0}/10^3$	0.1160 epochs	2	22.5
4096	$2^{1.5}/10^3$	0.2320 epochs	2	22.2

Table 4: DATE scales the batch size for ImageNet training by ResNet-50 without tuning hype-parameters. According to Stanford DAWN benchmark, 93% top-5 accuracy for ImageNet is the metric of a correct ResNet50 model.

Batch Size	Init LR	Warmup	Epochs	Top-5 Accuracy
1024	$2^{2.5}$	$10/2^5$ epochs	90	0.9336
2048	$2^{3.0}$	$10/2^4$ epochs	90	0.9325
4096	$2^{3.5}$	$10/2^3$ epochs	90	0.9334
8192	$2^{4.0}$	$10/2^2$ epochs	90	0.9355
16384	$2^{4.5}$	$10/2^1$ epochs	90	0.9343
32768	$2^{5.0}$	10 epochs	90	0.9318

is still able to beat them in accuracy (Figure 18). To repeat, DATE does not require hyper-parameter tuning.

4.4 ImageNet Training with ResNet-50

To show its robustness, we also apply DATE to the large-scale image classifications. We use DATE in ImageNet training with ResNet50. According to Stanford DAWN benchmark[11], 93% top-5 accuracy is the metric of a correct ResNet50 implementation. We are able to scale the batch size to 32K and achieve the target accuracy without tuning hype-parameters (Table 4). We achieved a constant performance and a higher accuracy compared to existing tuning schemes (Figure 1).

4.5 Energy-Efficient Communication

The communication is becoming a major cost for the energy consumption [6]. In our profiling result, the cost of a single floating point operation is around 100 pJ. However, the cost to move a word off-chip to a neighboring node is around 2500 pJ. In addition to the speed improvement, we want to reduce the energy cost of the communication over the network. Our results shown that DATE can significantly reduce the communication energy cost (Figure 19).

5 SPEEDUP AND SCALING

For ImageNet training with ResNet50, our auto-tuning approach is able to scale the batch size to 32K without losing accuracy. On a TPU-v2 Pod, we are able to finish the training in 7 minutes. The baseline [13] can only scale the batch size to 8K, which takes 16 minutes on the same TPU-v2 Pod. Ying et al. [47] are able to finish the ImageNet training with ResNet-50 in 2.2 minutes by LARS solver [48]; however, they use a better hardware (i.e. TPU-v3 Pod). If we port their code to TPU-v2 Pod, they achieve the same speed as us. The difference between our results and theirs is that they need to tune the hyper-parameters manually while we design an auto-tuning technique. This paper does not claim the contribution of implementing LARS on TPUs. We claim the contribution of the design of DATE and implementing DATE on TPUs, which does not decrease the system speed. We got 119% weak scaling efficiency when we scale from one cloud TPU server to one TPU Pod (equals to 64 cloud TPU severs). The training on one cloud TPU server requires 8 hours and 52.5 minutes (76.1× speedup). The reason behind the superlinear speedup is that we reduce the number of iterations linearly as we increase the batch size (line 2 of Algorithm 1). The number of communication messages is linear with the number of iterations. For the other four LSTM-based applications, DATE can also help the system utilize a much larger batch size without sacrificing accuracy. This leads to significant speedups on all the four datasets (Figure 2). For example, our GNMT baseline with a batch size of 256 needs more than 2 hours to finish the training on a cloud TPU-v2. With DATE, the GNMT with a batch size of 4096 can finish the training in 33 minutes on the same cloud TPU-v2. In summary, DATE achieves a 5.3× average speedup over the baselines for 4 LSTM-based applications on the same hardware.

5.1 Scaling on Various Models

In this section, we present more scaling results. We also want to study the impact of DATE on different architectures that achieve the same goal on the same dataset. We pick the ImageNet dataset as it has driven the deep learning and HPC communities in recent years. We pick several state-of-the-art models that were proposed in recent years: AlexNet [25], Inception-v4 [41], MnasNet [42], MobileNet [17], ResNet-50 [14], and SqueezeNet [18]. As mentioned in previous sections, DATE can achieve the consistent accuracy without hyper-parameter tuning when we scale the batch size to extremely large cases. In this section, we focus on the system scaling abilities of DATE. We focus on weak scaling study in this section because we want to make sure each node is fully utilized as we scale the number of nodes. We use 128 v3 TPU chips (latest TPUs) in the section. The data in previous sections are measured by v2 TPU chips. The baseline uses four TPU chips and a batch size of 256 (i.e. 64 per chip). We keeps workload per chip constant and increases the number of chips from 4 to 16, 32, 64, and 128. Figure 4 - Figure 9 show the scaling results. From these figures we can see that DATE can achieve good scaling results for all the models. The best scaling efficiency (100.08%) is achieved on ResNet-50 while the worst scaling efficiency is achieved on AlexNet (76.66%). The scaling efficiency of ResNet-50 in this section is different from the efficiency in Section 4.4. The reason is that they use the different hardware (chips and network).

To explain the difference in scaling efficiency for different models, let us define the computational intensity here, which is a similar concept in the Roofline study [45]. We define the computational intensity as the ratio between the computation volume and the communication volume. Here, the computation volume means the number of floating-point operations required to process each image.

[11] https://dawn.cs.stanford.edu/benchmark/

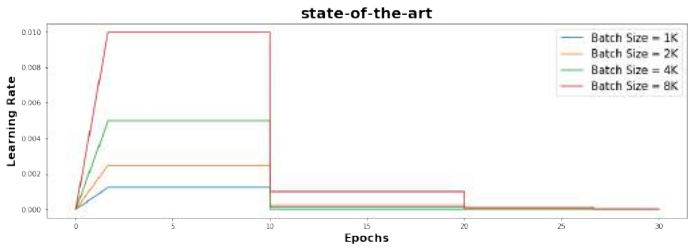
The communication volume means the number of parameters transferred in each message. For deep learning applications, the number of parameters transferred in each message is equal to the number of parameters in the gradients (i.e. the number of parameters in the model). For AlexNet, the model has 61 million parameters and it requires 1.5 billion operations to process each image. Thus, AlexNet has a computational intensity of 24.6. For ResNet-50, the model has 25 million parameters and it requires 7.7 billion operations to process each image. Thus, ResNet-50 has a computational intensity of 308. That is the reason why ResNet-50 has a much higher than scaling efficiency than AlexNet (100.08% vs 76.66%). DATE is well optimized as it can achieve a good scaling efficiency even the model has a low computational intensity. The same analysis works for other models.

6 CONCLUSION

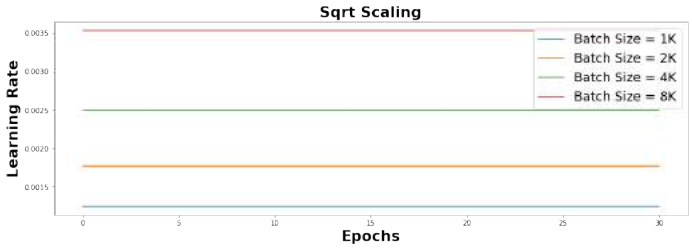
DATE is an auto-tuning method equipped with auto-tuning warmup, LR Scaling, LR decay and adaptive LR updating techniques. In practice, DATE performs well on both RNN applications and CNN applications. For LSTM applications, we are able to scale the batch size by a factor of 64× without losing accuracy and without tuning the hyper-parameters. For CNN applications, DATE is able to keep accuracy constant even as we scale the batch size to 32K, and we have demonstrated that DATE works uniformly better than previous large-batch auto-tuning techniques (Figure 1). For four LSTM applications, while running on the same hardware, DATE achieves a 5.3× average speedup. We also provide some theoretical explanations for the key techniques of DATE.

7 ACKNOWLEDGE

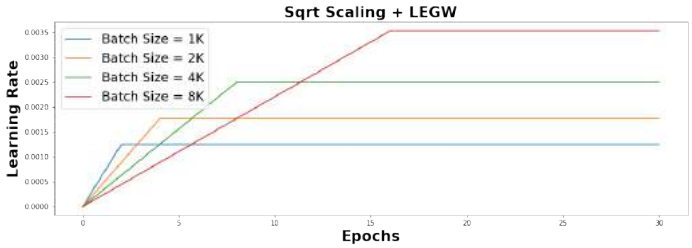
JD and YY are supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0010200; by DARPA Award Number HR0011-12- 2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and Samsung. Other industrial sponsors include Mathworks and Cray. In addition to ASPIRE sponsors, KK is supported by an auxiliary Deep Learning ISRA from Intel. CJH also thank XSEDE and Nvidia for independent support. We thank CSCS for granting us access to Piz Daint resources.



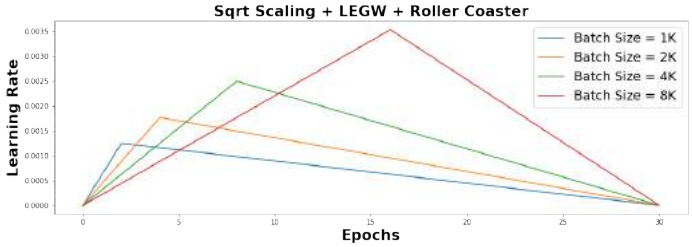
10.1



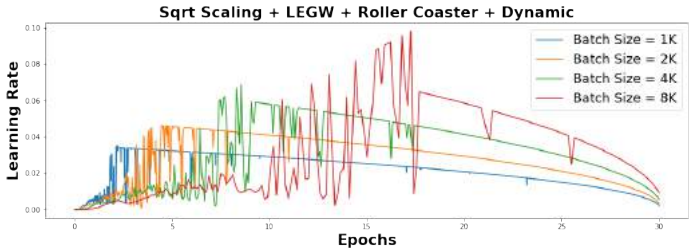
10.2



10.3

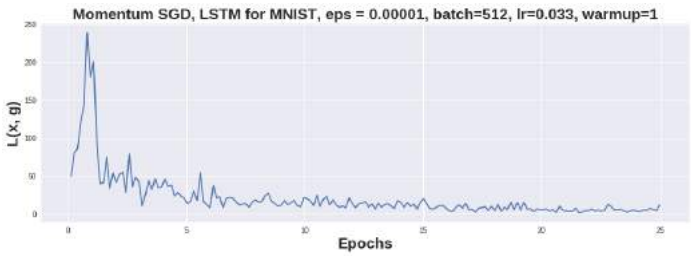


10.4

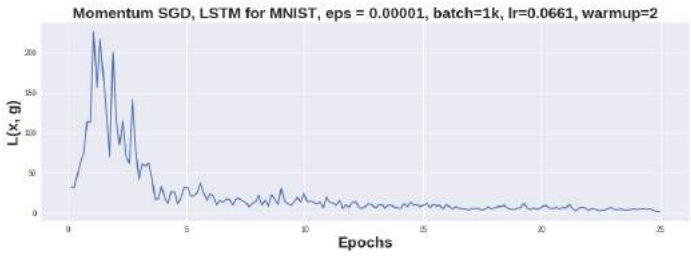


10.5

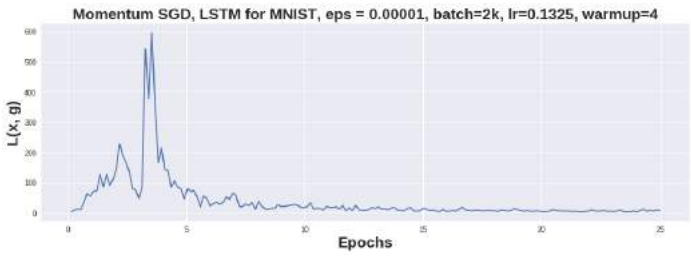
Figure 10: This figure illustrates the main features of DATE framework (Sqrt Scaling, LEGW, Roller Coaster, and Dynamic). This figure is also a summary of differences among the baseline, state-of-the-art approach, and the DATE framework. The application is LeNet/MNIST training, which totally needs 30 epochs.



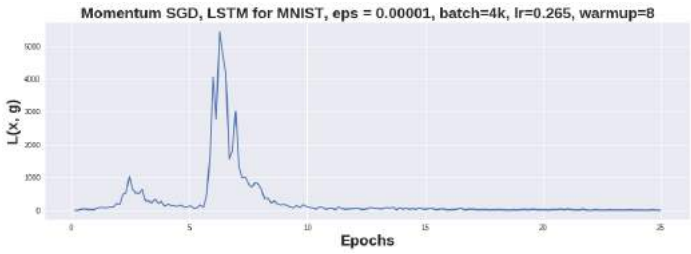
11.1 SGD with batch size 512.



11.2 SGD with batch size 1K.



11.3 SGD with batch size 2K.



11.4 SGD with batch size 4K.

Figure 11: The approximation of Lipchitz constant for different batch sizes.

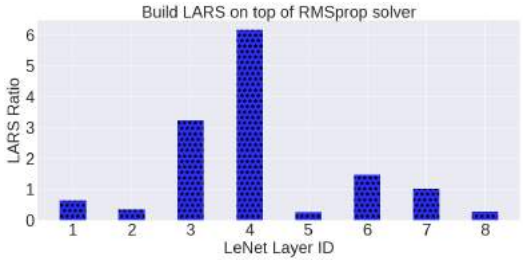
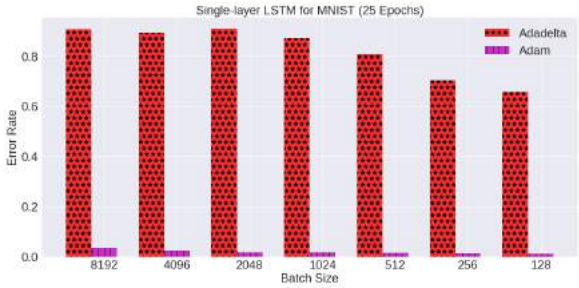
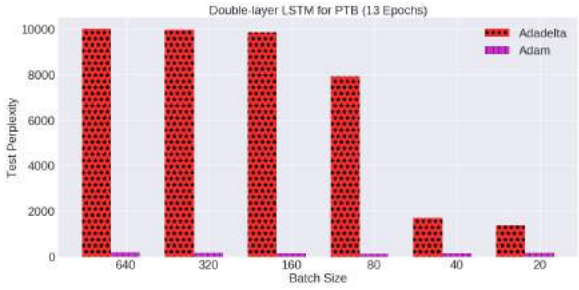


Figure 12: The LARS correction ratio based on RMSprop solver. It gives the fast learner a even larger learning rate.

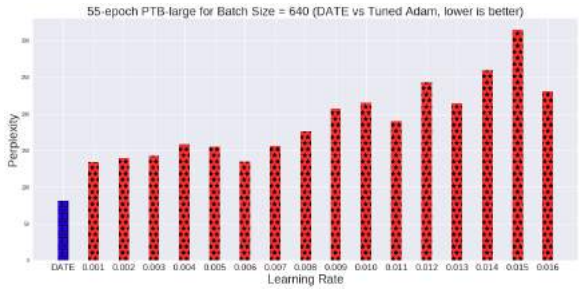


13.1

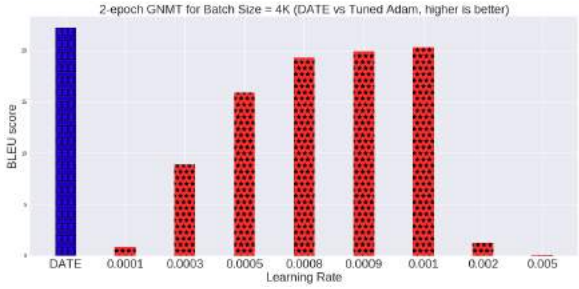


13.2

Figure 13: By just using the default hyper parameters, we find Adam works much better than Adadelta for MNIST and PTB datasets. Perplexity and Error Rate: lower is better.

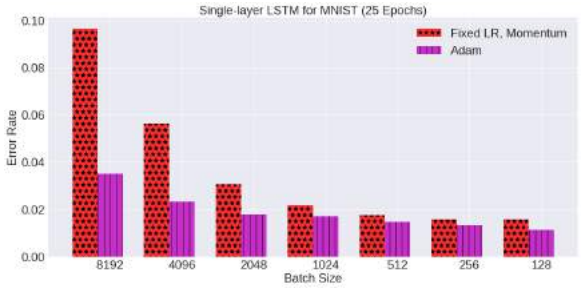


14.1

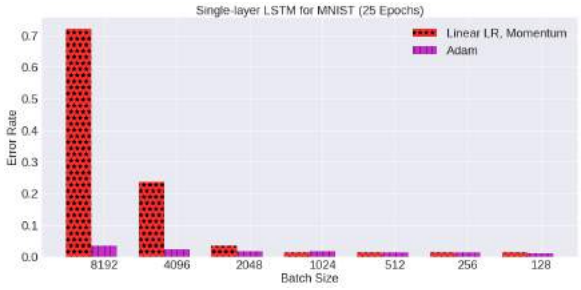


14.2

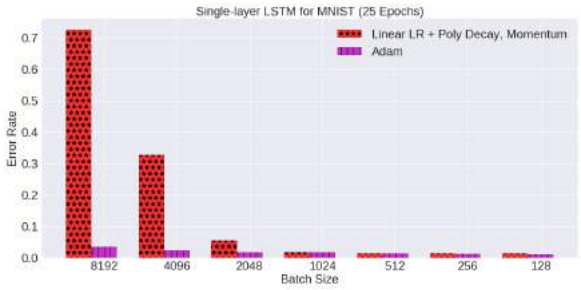
Figure 14: For Perplexity, lower is better. For BLEU score, higher is better. From these figures, we can observe that DATE performs better than the tuned Adam solver for PTB-large and GNMT applications.



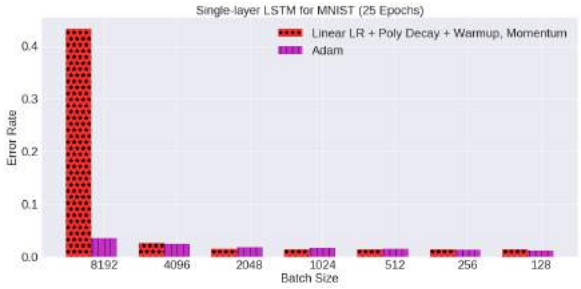
15.1



15.2

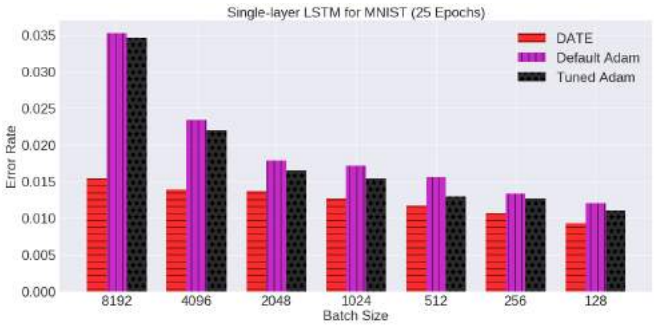


15.3

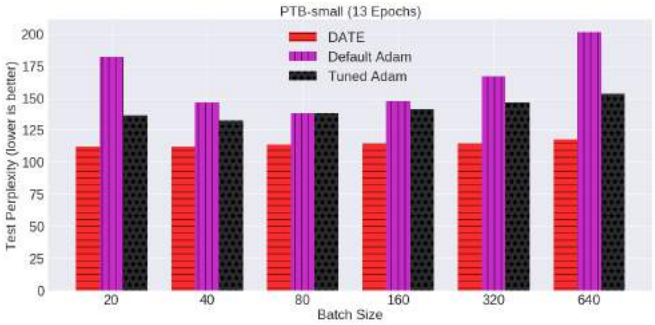


15.4

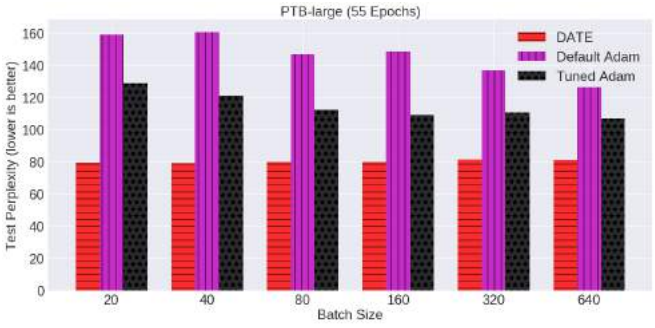
Figure 15: Adam can beat existing tuning techniques. We tune the learning rate for batch size = 128 and refer to it as η_0. Let us also refer to batch size as B. In Figure 15.1, all the tuning versions use η_0. In Figure 15.2, all the tuning versions use the linear scaling scheme (i.e. $\eta_0 \times B/128$). In Figure 15.3, all the tuning versions use the linear scaling scheme (i.e. $\eta_0 \times B/128$) and poly decay with power = 2. In Figure 15.4, all the tuning versions use the linear scaling scheme (i.e. $\eta_0 \times B/128$), poly decay with power = 2, and 5-epoch warmup.



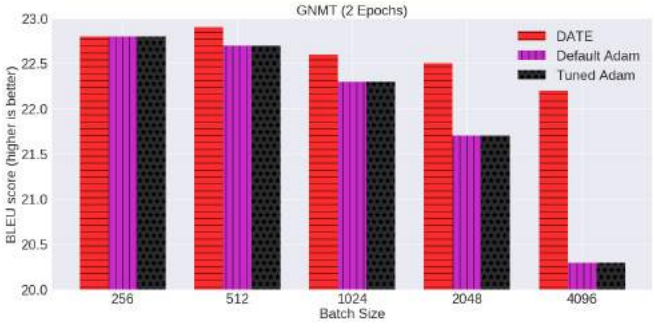
16.1



16.2

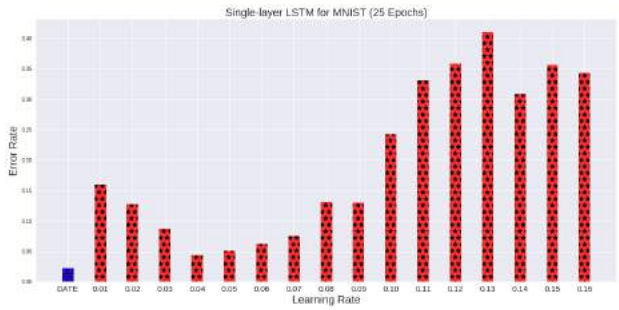


16.3

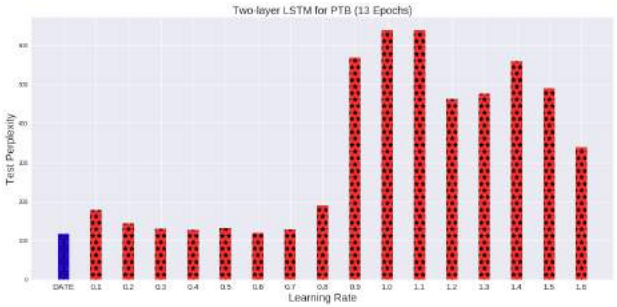


16.4

Figure 16: For the perplexity, lower is better. For BLEU score, higher is better. DATE performs much better than Adam solver for PTB and GNMT applications (running the same number of epochs). Even we comprehensively tuned the learning rate of Adam, it still can not beat DATE.

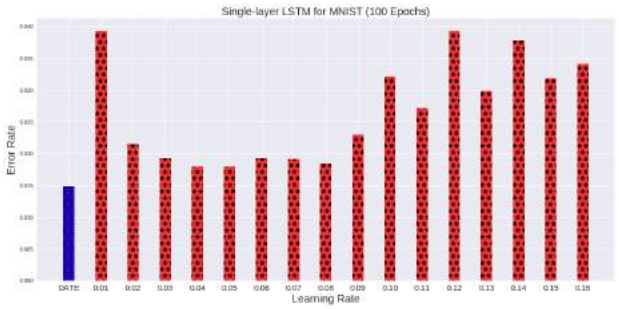


17.1

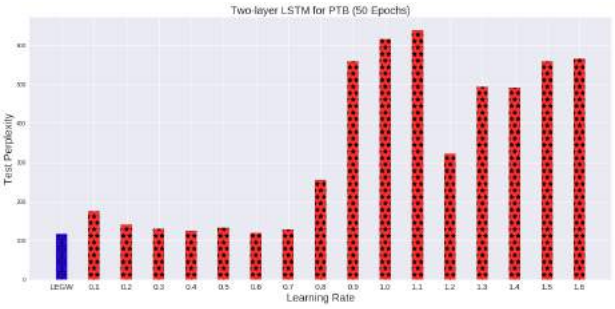


17.2

Figure 17: The data in this figure is collected from 8K batch size. Even when we comprehensively tune the learning rate of the baseline, it still is not able to beat DATE. For other hyper-parameters, DATE uses the same setting with the baseline.



18.1



18.2

Figure 18: The data in this figure is collected from 640 batch size. Even we comprehensively tune the initial learning rate of the baseline, it still is not able to beat DATE. For other hyper parameters, DATE uses the same setting with the baseline. Furthermore, we run the training long enough to make sure all of them are converged. DATE is still better.

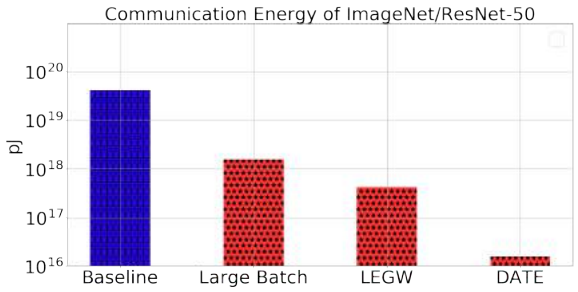


Figure 19: Let us use Er to denote the network communication energy cost in pJ. In our experiments, the baseline uses a batch size of 256. Large-batch approach increases the batch size to 32K and reduces the communication energy cost from Er to $\frac{\sqrt{(32768/256)}}{(32768/256)}Er$. The baseline conducts the hyper-parameter tuning 100 times. LEGW reduces the the communication energy cost from Er to $Er/100$ by enabling auto-tuning. DATE combines them together and reduces the communication energy cost from Er to $\frac{\sqrt{(32768/256)}}{(100\times32768/256)}Er$.

REFERENCES

[1] 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).

[2] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.

[3] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2016. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838* (2016).

[4] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[5] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. 2017. Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291* (2017).

[6] James Demmel. 2013. Communication-Avoiding Algorithms for Linear Algebra and Beyond.. In *IPDPS*. 585.

[7] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. *arXiv preprint arXiv:1712.02029* (2017).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[10] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

[11] AA Goldstein. 1977. Optimization of Lipschitz continuous functions. *Mathematical Programming* 13, 1 (1977), 14–22.

[12] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[13] 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).

[14] 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.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *arXiv preprint arXiv:1705.08741* (2017).

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[18] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[19] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

[20] 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).

[21] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–12.

[22] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[23] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[27] Mu Li. 2017. *Scaling Distributed Machine Learning with System and Algorithm Co-design*. Ph.D. Dissertation. Intel.

[28] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 661–670.

[29] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2016. Feature pyramid networks for object detection. *arXiv preprint arXiv:1612.03144* (2016).

[30] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. 2017. Neural Machine Translation (seq2seq) Tutorial. *https://github.com/tensorflow/nmt* (2017).

[31] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics* 19, 2 (1993), 313–330.

[32] James Martens and Roger Grosse. 2015. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*. 2408–2417.

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

[34] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. 2016. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In *Advances in Neural Information Processing Systems*. 3882–3890.

[35] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2018. Second-order Optimization Method for Large Mini-batch: Training ResNet-50 on ImageNet in 35 Epochs. *arXiv preprint arXiv:1811.12019* (2018).

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

[37] Herbert Robbins and Sutton Monro. 1985. A stochastic approximation method. In *Herbert Robbins Selected Papers*. Springer, 102–109.

[38] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*. 10435–10444.

[39] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489* (2017).

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

[41] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[42] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. 2018. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626* (2018).

[43] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012), 26–31.

[44] Yiren Wang and Fei Tian. 2016. Recurrent residual learning for sequence classification. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 938–943.

[45] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[47] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image Classification at Supercomputer Scale. *arXiv preprint arXiv:1811.06992* (2018).

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

[49] Yang You, Zhao Zhang, C Hsieh, James Demmel, and Kurt Keutzer. 2017. ImageNet training in minutes. *CoRR, abs/1709.05011* (2017).

[50] Fisher Yu and Vladlen Koltun. 2015. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122* (2015).

[51] Matthew D Zeiler. 2012. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We wrote our source codes based on a private version of Google's TensorFlow framework. For all the applications, we changed the optimizer engine of TensorFlow to our own implementation. For each application, we build a python project to implement the neural networks. The kernel is implemented in C/C++. We have three platforms: (1) a cloud server with eight V100 GPUs. (2) a cloud server with eight TPU-v2 chips. and (3) a cluster with 512 TPU-v2 chips (TPU Pod).

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`Not published; Available upon request.`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: TPU-v2

Operating systems and versions: Debian GNU/Linux 9.7 (stretch)

Compilers and versions: N/A

Applications and versions: ImageNet/ResNet-50; PTB; MNIST/LSTM

Libraries and versions: private TensorFlow

Key algorithms: SGD; LARS; Adam Optimizer

Input datasets and versions: ImageNet 2012; MNIST; PTB

Paper Modifications: For all the applications, we changed the optimizer engine of TensorFlow to our own implementation. We implement the DATE optimizer. For each application, we build a python project to implement the neural networks.

Output from scripts that gathers execution environment information.

```
Distributor ID: Debian
Description:    Debian GNU/Linux 9.7 (stretch)
Release:        9.7
Codename:       stretch
+ uname -a
Linux infer 4.9.0-8-amd64 #1 SMP Debian 4.9.130-2
↪  (2018-10-27) x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    2
Core(s) per socket:    1
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) CPU @ 2.00GHz
Stepping:              3
CPU MHz:               2000.182
BogoMIPS:              4000.36
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              56320K
NUMA node0 CPU(s):     0,1
Flags:                 fpu vme de pse tsc msr pae mce
↪  cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪  mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp
↪  lm constant_tsc rep_good nopl xtopology
↪  nonstop_tsc pni pclmulqdq ssse3 fma cx16 pcid
↪  sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx
↪  f16c rdrand hyp
ervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd
↪  ibrs ibpb stibp kaiser fsgsbase tsc_adjust bmi1
↪  hle avx2 smep bmi2 erms invpcid rtm mpx avx512f
↪  avx512dq rdseed adx smap clflushopt clwb avx512cd
↪  avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves
↪  arat arch_capabilities
+ cat /proc/meminfo
MemTotal:        7663544 kB
MemFree:         1217236 kB
MemAvailable:    5879748 kB
Buffers:          131488 kB
Cached:          4745548 kB
SwapCached:            0 kB
Active:          5618980 kB
Inactive:         598316 kB
Active(anon):    1340468 kB
Inactive(anon):    75552 kB
Active(file):    4278512 kB
Inactive(file):   522764 kB
```

```
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                60 kB
Writeback:             0 kB
AnonPages:       1340328 kB
Mapped:           178644 kB
Shmem:             75764 kB
Slab:             179968 kB
SReclaimable:     161292 kB
SUnreclaim:        18676 kB
KernelStack:        1824 kB
PageTables:         6216 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     3831772 kB
Committed_AS:    1804792 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      360436 kB
DirectMap2M:     7503872 kB
DirectMap1G:           0 kB
+ env
MAIL=/var/mail/xxx
USER=xxx
SSH_CLIENT=35.230.107.188 34836 22
SHLVL=1
HOME=/home/xxx
OLDPWD=/home/xxx
SSH_TTY=/dev/pts/0
LOGNAME=xxx
_=/usr/bin/nohup
XDG_SESSION_ID=659
TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/
↪  usr/games
XDG_RUNTIME_DIR=/run/user/1003
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
↪  1;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;0
↪  1:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=3
↪  4;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*
↪  .arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*
↪  .lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:
↪  *.txz=01;3
1:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=
↪  01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;
↪  31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;
↪  31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;
↪  31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;3
↪  1:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;3
↪  1:*.alz=01
;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:
↪  *.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:
↪  *.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;
↪  35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;
↪  35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01
↪  ;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=0
↪  1;35:*.pcx
=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01
↪  ;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=0
↪  1;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=0
↪  1;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01
↪  ;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=0
↪  1;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;
↪  35:*.xwd=0
1;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35
↪  :*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36
↪  :*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;3
↪  6:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36
↪  :*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;3
↪  6:*.xspf=00;36:
SHELL=/bin/bash
PWD=/home/xxx/Author-Kit
SSH_CONNECTION=35.230.107.188 34836 10.164.0.39 22
PYTHONPATH=:/usr/share/models/
+ inxi -F -c0
./collect_environment.sh: 8:
↪  ./collect_environment.sh: inxi: not found
+ lsblk -a
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0  250G  0 disk
└─sda1   8:1    0  250G  0 part /
```

ARTIFACT EVALUATION

Verification and validation studies: For machine learning classification, we usually use 70% of the dataset as the training data, 20% of the dataset as the test data, and 10% of the dataset as the validation data. In this way, we are confident that our results are solid and do not come from over-fitting.

Accuracy and precision of timings: For all the results including timings and accuracy, we run the same experiments six times and use the average.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: We use the same setting with the baseline. All the parameters are presented in our paper. We believe it should be easy to reproduce.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. For all the results including timings and accuracy, we run the same experiments six times and use the average.