CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers

Alexandros Koliousis[†], Pijika Watcharapichat[♯], Matthias Weidlich[‡],
Luo Mai[†], Paolo Costa[♯], Peter Pietzuch[†]

[†]Imperial College London [‡]Humboldt-Universität zu Berlin [♯]Microsoft Research

{a.koliousis, luo.mai, prp}@imperial.ac.uk, {pijika.watcharapichat, paolo.costa}@microsoft.com,
matthias.weidlich@hu-berlin.de

ABSTRACT

Deep learning models are trained on servers with many GPUs, and training must scale with the number of GPUs. Systems such as TensorFlow and Caffe2 train models with parallel synchronous stochastic gradient descent: they process a batch of training data at a time, partitioned across GPUs, and average the resulting partial gradients to obtain an updated global model. To fully utilise all GPUs, systems must increase the batch size, which hinders statistical efficiency. Users tune hyper-parameters such as the learning rate to compensate for this, which is complex and model-specific.

We describe CROSSBOW, a new single-server multi-GPU system for training deep learning models that enables users to freely choose their preferred batch size—however small—while scaling to multiple GPUs. CROSSBOW uses many parallel model replicas and avoids reduced statistical efficiency through a new synchronous training method. We introduce SMA, a synchronous variant of model averaging in which replicas *independently* explore the solution space with gradient descent, but adjust their search *synchronously* based on the trajectory of a globally-consistent average model. CROSSBOW achieves high hardware efficiency with small batch sizes by potentially training multiple model replicas per GPU, automatically tuning the number of replicas to maximise throughput. Our experiments show that CROSSBOW improves the training time of deep learning models on an 8-GPU server by 1.3–4× compared to TensorFlow.

PVLDB Reference Format:
A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB*, 12(11): 1399–1413, 2019.
DOI: https://doi.org/10.14778/3342263.3342276

1. INTRODUCTION

"If batch size could be made arbitrarily large [...], then training is amenable to standard weak scaling approaches. However, if the training [...] is restricted to small batch sizes, then we will need to find other algorithmic and architectural approaches to their acceleration."
– J. Dean, D. Patterson and C. Young [14], March 2018

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342276

Deep learning has revolutionised many application fields, including computer vision [35, 18], speech recognition [19, 68] and natural language processing [31]. The training of deep learning models is expensive: it takes roughly half a month to reproduce the state-of-the-art accuracy for the ImageNet challenge on a single NVIDIA Titan X GPU [16]. To reduce training time, systems exploit data parallelism across many GPUs to speed up training [38, 20, 13]. Consequently multi-GPU servers have become widely available: a new 10-GPU server with NVIDIA Pascal GPUs costs less than $40,000 [48], and public cloud providers offer GPU server instances with up to 16 GPUs [2].

Users expect training time to go down with the number of GPUs in a server. Scaling the training process is challenging though because it requires a system to fully utilise the parallelism of all GPUs without introducing bottlenecks. Existing systems, including TensorFlow [1], MXNet [9], CNTK [59], and Caffe2 [28], use parallel *synchronous stochastic gradient descent* (S-SGD) [34] for training: input batches of training data are partitioned across GPUs. Each GPU then updates its local model replica before a synchronisation operation calculates a new global model for the training of the next input batch.

To utilise many GPUs effectively, S-SGD must therefore use a large batch size. The batch size typically grows linearly with (1) the number of GPUs, and (2) the performance of each GPU. In practice, batch sizes of 64,000 are now not uncommon [27]. With large batch sizes though, *statistical efficiency* [71] of the training process reduces [33, 41]. As the per-GPU model replicas synchronise less frequently in relation to the processed training data, the converge rate decreases, which in turn increases the time-to-accuracy until the trained model reaches a target accuracy. Users try to compensate for this reduction in statistical efficiency by increasing the learning rate [16], or adjusting the batch size adaptively [62]. These techniques, however, require model-specific tuning and do not fundamentally solve the problem but eventually fail for very large batch sizes [14, 27, 16]. Given these implications of large batches, users prefer to use small batches when possible [41].

The goal of our work is to explore how to design a deep learning system that effectively trains with small batch sizes, i.e. between 2 and 32 [41], while still scaling to many GPUs. The starting point for our design is that, on each GPU, we simply train a model replica with a small batch size. This introduces two challenges, which we address in the paper: (i) how to synchronise this potentially large number of model replicas without adversely affecting statistical efficiency; and (ii) how to ensure that the hardware resources of each GPU are fully utilised, thus achieving high hardware efficiency?

We describe the design and implementation of CROSSBOW, a new single-server multi-GPU deep learning system that decreases time-to-accuracy when increasing the number of GPUs, irrespective of the batch size. The design of CROSSBOW makes the following new contributions:

(1) Synchronous model averaging (SMA). CROSSBOW uses SMA, a new synchronisation approach that synchronises model replicas in a scalable fashion with a low reduction in statistical efficiency. In SMA, multiple parallel *learners* each train their own model replica independently. Learners access a global average model to coordinate: they adjust their trajectories with an update proportional to their divergence from the average model. The average model thus reaches better minima faster than individual learners. All replicas, including the average model, are updated after each learner processes a single batch, and all accesses to the average model are strongly consistent.

(2) Auto-tuning the number of learners. With a small batch size, a single learner may not fully utilise the resources of a GPU. CROSSBOW therefore places multiple concurrent learners on the same GPU. The number of learners per GPU is tuned automatically. During training, CROSSBOW increases the number of learner until there is no increase in training throughput, i.e. the maximum hardware efficiency has been reached. It then uses the number of learners that resulted in peak throughput.

(3) Concurrent task engine. CROSSBOW has a task scheduler that dynamically schedules learners that process the next batch on the first available GPU. The scheduler issues learning and synchronisation tasks concurrently in order to prevent the synchronisation performed by SMA from becoming a bottleneck. CROSSBOW achieves this by breaking the global synchronisation barrier of SMA into a hierarchical tree: each learner synchronises using a local copy of the average model that resides on its GPU; and local models synchronise across GPUs. The local and global synchronisation operations have optimised implementations according to their communication scopes (e.g. using all-reduce [60]). They also overlap with the forward and backwards error propagation of learners.

In our experimental evaluation, we show that, when training ResNet-50 with 2 model replicas per GPU and a batch size of 16, CROSSBOW reaches a given target accuracy 1.5× faster than TensorFlow. Training with multiple model replicas per GPU reduces time-to-accuracy by 1.9× for ResNet-32, by 4.2× for VGG-16, and by 2.7× for LeNet, respectively. SMA improves statistical efficiency with multiple model replicas by up to 1.6×; while the task engine of CROSSBOW improves hardware efficiency by up to 1.8×.

The rest of the paper is organised as follows: §2 discusses the challenges when scaling the training by increasing the batch size; §3 introduces CROSSBOW's synchronous model averaging approach with independent learners; §4 describes the design and implementation of the CROSSBOW task engine; §5 presents out experimental results; §6 surveys related work; and §7 concludes.

2. SCALING DEEP LEARNING

Deep learning models, e.g. multi-layer convolutional neural networks [36], have been shown to achieve high accuracy for many image or speech classification problems [18, 3]. Since increasing the amount of training data and the number of model parameters improves their accuracy [18, 13], deep learning models require training approaches that exploit the parallelism of modern hardware.

2.1 Mini-batch gradient descent

Supervised training of deep learning models uses labelled samples, split into training and test data. A model gradually "learns" to predict the labels of training data by adjusting its model parameters based on the error. It usually takes several passes (or *epochs*) over the training data to minimise the prediction error. The test data is used to measure the model accuracy on previously unseen data. The most important metric is *test accuracy*, which measures the ability of the model to make predictions when deployed "in the wild".

More formally, let w be a vector of model parameters (*weights*), and $\ell_{\mathsf{x}}(\mathsf{w})$ be a loss function that, given w, measures the difference between the predicted label of a sample (x, y) and the ground truth y. The training problem is to find a w^* that minimises the average loss over all training data. In today's systems, this is achieved using *stochastic gradient descent* (SGD) [56, 4, 5], an iterative training algorithm that adjusts w based on a few samples at a time:

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \gamma_n \nabla \ell_{\mathsf{B}_n}(\mathsf{w}_n) \tag{1}$$

where γ_n is the learning rate in the n-th iteration of the algorithm, B_n is a mini-batch of b training samples, and $\nabla\ell$ is the gradient of the loss function, averaged over the batch samples:

$$\nabla \ell_{\mathsf{B}_n}(\mathsf{w}_n) = \frac{1}{b} \sum_{\mathsf{x} \in \mathsf{B}_n} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \tag{2}$$

It is common to augment Eq. (1) with *momentum*, a technique known to accelerate the convergence of deep learning models [63]. Using momentum, the training process favours gradients that descent in directions known to improve accuracy from previous iterations. The iterative training algorithm with momentum becomes:

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \gamma_n \nabla \ell_{\mathsf{B}_n}(\mathsf{w}_n) + \mu(\mathsf{w}_n - \mathsf{w}_{n-1}) \tag{3}$$

where μ is the momentum parameter and $\mathsf{w}_n - \mathsf{w}_{n-1}$ denotes the algorithm's previous search direction.

Gradient back-propagation [57] is used to compute the model gradients when weights are spread across multiple layers. This is done in two steps: (i) an input batch propagates *forward* through the layers to compute the predicted label. This is compared with the ground-truth label associated with each sample in the batch, measuring the error; and (ii) the error propagates *backwards* from layer to layer in reverse order. The error is used to compute the gradient for the weights in each layer, and the weights can then be updated incrementally by applying Eq. (3).

When training a deep learning model, the goal is to reduce the time to reach a target level of test accuracy (*time-to-accuracy*). Two factors affect the time-to-accuracy: (i) the number of iterations that a training algorithm such as SGD requires to find a solution with a given test accuracy (*statistical efficiency*); and (ii) the execution time of each iteration (*hardware efficiency*).

2.2 Training with GPUs

GPU architectures are well suited for increasing the hardware efficiency of the training process. A GPU is a many-core processor that is designed for high processing throughput. It features thousands of cores, which are simple floating-point arithmetic units. Arranged in groups, cores form tens of streaming multi-processors (SMs). Multiple threads can execute the same instruction per cycle on different data, such as a training sample or a weight.

When training a deep learning model on a GPU, a batch of training data B (or, during the backwards phase, its error) is transformed via a series of matrix or vector floating-point operations (e.g. $\mathsf{B} \times \mathsf{w}$) as it propagates from one layer to the next. GPUs can perform more floating-point operations per weight read than a CPU, thus achieving more model updates per second [32].

Programs for the GPU are *kernels*, which can be executed in a blocking or non-blocking fashion. Complex multi-layer deep learning models may comprise of hundreds of kernels. Kernels are executed in-order as part of a GPU *stream*, which is a queue of device work. A GPU can have more than one stream, which allows kernels to execute *concurrently*. Modern GPUs support *events*,

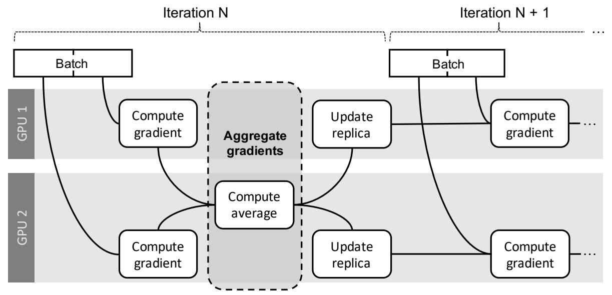


Figure 1: Parallel S-SGD with two GPUs (Each GPU reads half a batch and computes a gradient based on its local model replica. Training does not proceed to the next batch until all replicas have been updated with the aggregate value of the computed gradients.)

which are a publish/subscribe mechanism to synchronise across different streams, without having to stall the entire GPU pipeline.

Copying input data from CPU to GPU memory over the PCIe bus is typically assisted by a *copy engine* on the GPU, which runs independently from the GPU's compute engine that schedules kernels. Systems therefore hide the latency of communication by overlapping communication with computation tasks (e.g. using NVIDIA's Collective Communication Library (NCCL) [46]).

A server can have multiple GPUs, and GPU-to-GPU data transfers use the PCIe bus or exploit a fast direct interconnect such as NVIDIA's NVLink bridge [47]. The GPUs in a server are interconnected in a topology with varying bandwidth: e.g. in a two-socket multi-GPU server, the GPUs may form a binary tree in which each GPU pair is connected to a PCI switch, and two pairs are connected with a PCI host bridge, attached to a CPU socket.

2.3 Parallel synchronous gradient descent

Current parallel training approaches distribute the gradient computation across multiple GPUs, but differ in how they synchronise the gradients. The prevailing training algorithm is parallel *synchronous SGD* (S-SGD) [34]. It requires all GPUs to have a consistent view of the n-th version of the model before the $(n+1)$-th iteration starts: (i) at each iteration, S-SGD partitions a batch equally across GPUs; (ii) each GPU computes a *partial* gradient from a batch partition and the latest model version; (iii) GPUs then coordinate to merge partial gradients into an *aggregate* gradient (according to Eq. (2)); and (iv) the aggregate gradient is used to update the models (according to Eq. (3)) before the next iteration.

Figure 1 shows the execution of S-SGD on a two-GPU server. Each GPU has a local model replica in its memory, which is used to compute the partial gradients. The GPUs coordinate so that the same aggregate gradient is applied to all local replicas ensuring consistency: a GPU collects partial gradients, averages them, and then disseminates the result.

S-SGD can be combined with a parameter server (PS) architecture [38] that synchronises GPUs against a centralised entity. The gradients can also be partitioned and sent to multiple PSs for aggregation and updating of a sharded model.

To address stragglers during gradient computation or synchronisation, researchers have proposed an *asynchronous* variant of SGD (A-SGD) [7]. In A-SGD, a GPU progresses to the next iteration immediately after its partial gradient was added to the aggregate gradient, and uses the value accumulated thus far to update its model replica. This leads to *stale* gradients and hard-to-understand asynchrony, making it difficult to train complex neural network models such as ResNet effectively. In contrast, S-SGD has better convergence properties, which is why it has become the de-facto standard for the training of deep neural networks [16, 27]. We therefore also focus on synchronous training.

2.4 Challenges in scaling training

"Training with large mini-batches is bad for your health. More importantly, it's bad for your test error. Friends don't let friends use mini-batches larger than 32."
–Y. LeCun (`@ylecun`), April 2018

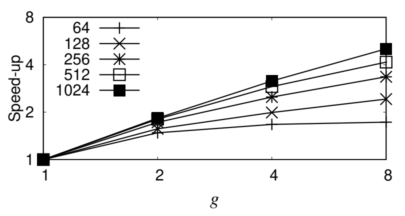
The batch size is a critical parameter when training with parallel S-SGD: if the batch is too small, the GPU is not fully utilised because the communication overhead to move data to and from the GPU dominates. In parallel training, the (aggregate) batch size must therefore increase *linearly* with the number of GPUs, resulting in a constant batch size per GPU; otherwise, the overall throughput scales poorly. This effect can be seen in the plot of hardware efficiency in Figure 2a. It shows the relative throughput speed-up when training a model with TensorFlow as we increase the number of GPUs. If the aggregate batch size remains constant (e.g. 64), the throughput does not increase linearly because the batch size per GPU decreases (e.g. with 8 GPUs the batch per GPU is just 8). In contrast, if we increase the aggregate batch size (e.g. to 512 or 1,024 for 8 GPUs), thus maintaining a constant batch size per GPU, we observe a linear speed-up. The same phenomenon holds when considering the time per epoch. Figure 2b shows that large batch sizes reduce the time for a complete pass over the training data.

While the above shows that large batch sizes are ideal to ensure high *hardware efficiency*, they exhibit poor *statistical efficiency* [41], which is expressed as the number of epochs required to converge to a given accuracy (ETA). This is shown in the plot of statistical efficiency in Figure 2c: as the batch size increases, TensorFlow requires more epochs to converge. The reasons are twofold: (1) with large and redundant training datasets (as it is often the case), small batches ensure faster training because only few batches are sufficient to capture the dimensionality of the problem space and converge quickly to good solutions [37, 5]; (2) a small batch size leads to "noisier" gradient updates, which widen the exploration of the loss landscape, making it more likely to find better solutions with a higher test accuracy [22, 21, 33, 26]

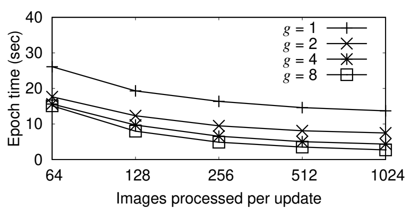
This trade-off between hardware and statistical efficiency is particularly detrimental in parallel training. While increasing the batch size increases the throughput linearly with the number of GPUs, beyond a certain threshold (e.g. 256 in Figure 2c), the number of epochs increases *super-linearly*, thus preventing a linear reduction of training time. This effect remains when using a parameter server (PS) architecture to implement S-SGD: with a sharded model, large batch sizes are also necessary for high hardware efficiency.

A typical solution to mitigate this issue and compensate for the loss of statistical efficiency with larger batch sizes is hyper-parameter tuning, e.g. dynamically adjusting the batch size as well as other hyper-parameters, such as the learning rate and the momentum, during the training process. In particular, it has been observed that, as long as the ratio between the learning rate and the batch size remains constant, training may be improved by varying the batch size [26, 62, 16, 22]. This only holds when the learning rate remains relatively small though [16].

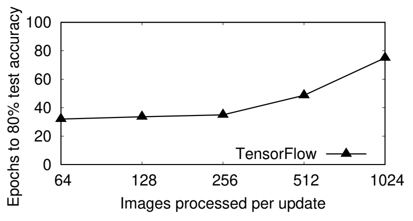
While hyper-parameter tuning can achieve quasi-linear scaling of the training time for large networks such as ResNet-50 on up to 1,024 GPUs [16, 62, 27], it requires a time-consuming model-specific methodology, which is often beyond the reach of non-experts and cannot be applied easily to new models or hardware architectures. In some cases, even with hyper-parameter tuning, it is hard to scale training time on multiple GPUs: a recent study from



(a) Hardware efficiency



(b) Time per epoch



(c) Statistical efficiency

Figure 2: Trade-off between hardware and statistical efficiency (The figures shows (a) the speed-up with an increasing number of GPUs when varying the batch size; (b) the corresponding epoch time; and (c) the number of epochs to reach a target accuracy of 80% when varying the batch size. The model trained is ResNet-32 with TensorFlow.)

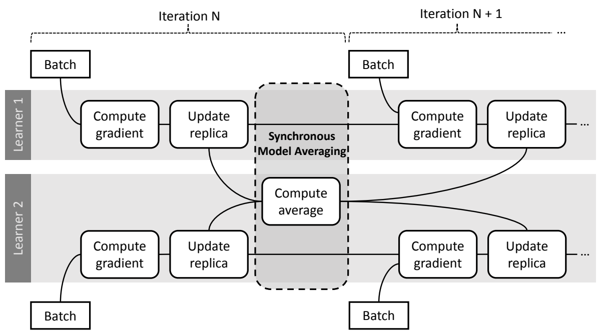


Figure 3: Parallel training with two learners (Each learner independently trains a model replica. The replica is updated based on the locally computed gradients as well as corrections derived through model averaging.)

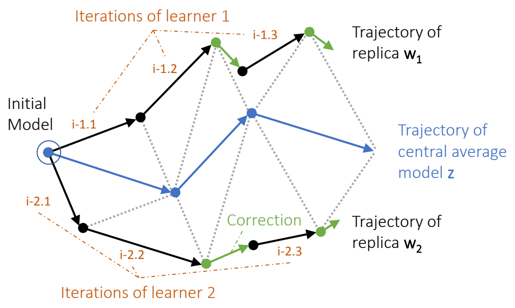


Figure 4: Intuition behind SMA (Two replicas w_1 and w_2 are trained by two learners. Their model updates are incorporated in a central average model z. Based on the latter, corrections for w_1 and w_2 are derived and applied after each iteration.)

Google Brain [61] shows that convolutional networks exhibit only limited scaling with batch sizes larger than 64 and, for recurrent neural networks, e.g. long short-term memory (LSTM), the threshold seems to be even lower (16) [61].

A general approach for scaling S-SGD on multiple GPUs therefore remains an open challenge due to the conflicting impact of large batch sizes on hardware and statistical efficiency. In the next section, we show how CROSSBOW addresses this problem by leveraging a new synchronisation approach among fully-utilised GPUs, thus achieving high GPU throughput without sacrificing converge speed.

3. SYNCHRONOUS MODEL AVERAGING WITH LEARNERS

Our approach relies on the concept of a *learner*, which independently trains a model replica for a given input batch (§3.1). Having many independent learners requires careful synchronisation in order to achieve high statistical efficiency. We introduce a new algorithm, named *synchronous model averaging* (SMA), that consolidates the model updates computed by many learners (§3.2). After that, we discuss how to train multiple learners per GPU (§3.3) and how to determine the number of learners to use (§3.4).

3.1 Independent learners

Parallel S-SGD imposes tight synchronisation when processing partitioned batches. The gradients computed based on *all* model replicas are aggregated, and the obtained result is incorporated by all replicas. After each iteration, before the computation of gradients for the next batch begins, all replicas are therefore the same.

Our idea is to introduce more diversity into the learning process based on the notion of a *learner*. A learner is an entity that trains a single model replica *independently* with a given batch size. The rationale for this abstraction is that it decouples the batch size from the degree of parallelism in the learning process. Each learner can process a batch of small size to achieve high statistical efficiency, while the number of learners offers a way to achieve high hardware efficiency. Figure 3 shows two learners, each being assigned a different complete batch. A learner computes a gradient and immediately updates its replica based on the gradient. It then continues with the gradient computation for the next batch. To prevent learners from diverging, each learner also applies a *correction* to its model, which is incorporated synchronously as part of the next update of the replica. As we explain in the next section, corrections penalise local replicas when deviating from the consensus among all replicas.

In contrast with parallel S-SGD, model replicas with learners evolve independently because they are not reset to a single global model after each batch. Unlike asynchronous learning approaches [55, 45, 71], each replica is corrected in each iteration to maintain the convergence of the learning process. Learners enable us to achieve both high statistical efficiency and hardware efficiency, avoiding the trade-off between then faced by existing approaches.

3.2 SMA algorithm

To synchronise the local models of learners, we propose *synchronous model averaging* (SMA), a new algorithm based on model averaging [51, 50, 58]. SMA consolidates the model updates of learners by maintaining a *central average model*. We illustrate the idea behind SMA in Figure 4. Starting with the initial model, two learners train replicas, w_1 and w_2, with distinct batches. Once the learners have computed the gradients and updated their local replicas, the updates are applied to a central average model.

A challenge for SMA is that, since learners are independent, they explore different local minima during training and therefore would diverge over time. As a solution, SMA uses the average model to compute *corrections* for each learner, thus ensuring that they follow

```
input  : $\mathsf{w}_0$, an initial model;
         $\mathsf{w}_1 \dots \mathsf{w}_k$, $k$ model replicas trained by $k$ learners;
         $\mathbb{B}$, a set of batches;
         $\gamma$, a learning rate parameter;
         $\mu$, a momentum parameter;
output : $\mathsf{z}$, the trained model.
// Initialise central average model and its previous version
1  $\mathsf{z} \leftarrow \mathsf{w}_0$ ;
2  $\mathsf{z}_{prev} \leftarrow \emptyset$ ;
3  while target accuracy not reached $\wedge\ |\mathbb{B}| \geq k$ do
       // $i$-th iteration of the learning algorithm
4      $c_1, \dots, c_k \leftarrow \emptyset, \dots, \emptyset$ ;
5      for $j \in \{1, \dots, k\}$ do
           // $j$-th learner in the $i$-th algorithm iteration
6          $B_j \leftarrow select(\mathbb{B})$ ;              // Select batch for learner $j$
7          $\mathbb{B} \leftarrow \mathbb{B} \setminus \{B_j\}$ ;               // Remove the batch
8          $g_j \leftarrow \gamma \nabla \ell_{B_j}(\mathsf{w}_j)$ ;         // Gradient for replica $j$
9          $c_j \leftarrow \alpha(\mathsf{w}_j - \mathsf{z})$ ;            // Correction for replica $j$
10         $\mathsf{w}_j \leftarrow \mathsf{w}_j - g_j - c_j$ ;            // Update replica $j$
       // Update central average model
11     $\mathsf{z}' \leftarrow \mathsf{z}$ ;
12     $\mathsf{z} \leftarrow \mathsf{z} + \sum_{j=1}^{k} c_j + \mu(\mathsf{z} - \mathsf{z}_{prev})$ ;
13     $\mathsf{z}_{prev} \leftarrow \mathsf{z}'$ ;
```

Algorithm 1: Synchronous model averaging (SMA)

the trajectory of the central average model (see Figure 4). Intuitively, a correction represents a penalty for disagreeing with the consensus in the previous iteration, and pulls the model replica of a learner toward the average value over all replicas [6]. Eventually all replicas will agree on the optimality of the central average model.

When the variance of all gradients is low, prior approaches that use corrections either postpone them [73] or compute them using an inconsistent view of the central average model [40]. In contrast, SMA applies corrections *synchronously* at every iteration. This avoids introducing any error in the stochastic process when corrections are applied and therefore accelerates convergence.

A desirable property of model averaging is that the asymptotic variance of the central average model decays faster (at an optimal rate [50]) than the variance of individual replicas, improving convergence over S-SGD. While model averaging reduces variance, it is known that it is ineffective at forgetting the initial weights [25, 15].

To discard initial weights faster, SMA introduces *momentum* (see §2.1). Unlike prior work that only applies momentum to individual model replicas [73, 39], SMA incorporates the directions of model weights into the updates to the central average model: updates in directions of persistent descent are kept; those in other directions are cancelled or diminished. SMA uses Polyak's momentum method [49] because, compared to Nesterov's accelerated gradient [44], it fits model averaging better: the update to the central average model is computed by all learners based on current positions and not estimated ones [63].

Existing solutions either refrain from using averaging initially, starting at later iterations [25] or vary the contribution of each correction over time [73]. Both require model-specific parameter tuning [50], which SMA's use of momentum avoids.

We formalise the SMA algorithm in Alg. 1. It takes as input a model, initialised as w_0, a set of k model replicas $\mathsf{w}_1 \dots \mathsf{w}_k$ that are managed by k learners, a set of batches $\mathbb{B}$, along with two hyper-parameters: the learning rate γ and the momentum μ. Upon termination, the algorithm returns the trained model.

First SMA initialises the central average model z and a reference

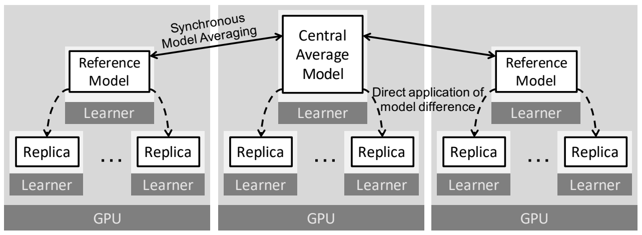


Figure 5: Synchronising multiple learners per GPU (SMA is applied for one reference model per GPU; further local learners incorporate differences of their replicas and the reference model.)

to a previous version of it, z_{prev} (lines 1–2). It defines an iterative learning process (lines 3–13) that terminates when the target accuracy has been reached by the central average model z or there are insufficient batches available.

In each iteration, a learner j proceeds as follows: (i) it selects a batch B_j (line 6) and, using the batch and its replica w_j, the learner computes a gradient g_j (as in Eq. 1) under the given learning rate (line 8); (ii) it computes a correction c_j as the difference between the replica w_j and the central average model z where $\alpha \approx 1/k$ is a constant (line 9); and (iii) the model replica w_j is then updated by applying the gradient g_j and the correction c_j (line 10).

The iteration completes with an update to the central average model z (line 12). This update is twofold: (i) it includes the corrections derived for all the k model replicas assigned to the independent learners, which represent the current differences of the replicas with respect to z; and (ii) a second component exploits the configured momentum and the previous version of the central average model z_{prev} to accelerate convergence by maintaining the direction of gradients [19]. In the i-th iteration of SMA, z_{prev} is the model at the beginning of the $(i-1)$-th iteration.

Similar to most parallel training approaches [34, 73], SMA benefits from an online adaptation of hyper-parameters. Based on the accuracy observed after each iteration, the learning rate (parameter γ in Alg. 1) may be reduced step-wise to overcome oscillation of the trained model and improve accuracy [37]. Such adaptation can be realised by updating γ directly in each iteration in SMA. For example, when training ResNet-50, it is common to reduce the learning rate twice, at the 30th and 60th epochs [18], which are chosen empirically based on the accuracy observed after each iteration.

With SMA, however, oscillatory behaviour is observed on the central average model, whereas a change in the learning rate affects the training of each model separately. Since SMA does not reset all models in each iteration, the accuracy may not necessarily improve each time the learning rate changes. When detecting such a situation, we therefore restart SMA: Alg. 1 is executed again with the latest version of the central average model z as the new initial model w_0.

Overall, SMA offers the benefits of small-batch training through model averaging that improves test accuracy at scale. Its use of synchronous corrections makes training insensitive to different non-convex models because frequent and consistent corrections reduce gradient variance faster than asynchronous ones when the number of learners increases. Its use of momentum on the central average model accelerates convergence with many learners, compensating for the adverse effect of lower variance at the start of training.

3.3 Training multiple learners per GPU

When selecting a small batch size for training to achieve high statistical efficiency, training a single model replica may not saturate all GPU resources. Learners decouple the processing of a batch from the available hardware, permitting the execution of multiple

```
input : $\tau$, a throughput threshold parameter;
1 $l_1, \ldots, l_m \leftarrow 1, \ldots, 1$ ;        // Number of learners for each of $m$ GPUs
2 $t'_1, \ldots, t'_m \leftarrow 0, \ldots, 0$ ; // Throughput observed earlier for the $m$ GPUs
3 while SMA executes do
4     for $g \in \{1, \ldots, m\}$ do
          // Observe learning throughput of $g$-th GPU
5         $t \leftarrow$ get-current-throughput$(g)$;
          // Adapt number of learners for $g$-th GPU
6         if $t - t'_g > \tau$ then $l_g \leftarrow l_g + 1$ ;
7         else if $t < t'_g \wedge l_g > 1$ then $l_g \leftarrow l_g - 1$ ;
8         $t'_g \leftarrow t$ ;
```

Algorithm 2: Selecting the number of learners per GPU

learners per GPU. With a small batch size (e.g. 2), dozens of learners can run concurrently on a multi-GPU server.

Given a potentially large number of learners, we take advantage of the fact that some learners reside on the same GPU and access shared memory, which is at least an order of magnitude faster than PCIe for inter-GPU communication. Rather than aggregating all differences in a single step, we therefore separate synchronisation on the intra-GPU and inter-GPU level.

We organise synchronisation as illustrated in Figure 5. While hierarchical synchronisation has been put forward in many contexts, CROSSBOW exploits such a scheme for intra-GPU as well as inter-GPU synchronisation of model replicas. To synchronise the learners executing on a single GPU, one learner is chosen to manage a *reference model*. Each learner then computes the difference between its model replica and the local reference model. This difference is then applied to the respective replica. At the global level, the SMA algorithm is executed. It uses one of the local reference models as the central average model (z in Alg. 1); all other reference models (one per GPU) are the replicas incorporated into the model averaging process (w_j in Alg. 1).

3.4 Choosing the number of learners

The number of learners per GPU or, put differently, the number of model replicas to be trained in parallel, is an important parameter. It must be chosen carefully for a given batch size: when training too few replicas, a GPU is under-utilised, wasting resources; when training too many, the execution of otherwise independent learners is partially sequentialised on a GPU, leading to a slow-down.

We propose to tune the number of learners per GPU based on the training throughput at runtime. Unlike existing approaches to auto-tuning that target hyper-parameters such as the learning rate and the momentum, this enables adaptive control of the number of model replicas per GPU. By observing the number of processed batches per second, we detect under- and over-utilisation of a GPU. As described in Alg. 2, we initially use a single learner per GPU (line 1). For each GPU, we then consider the learning throughput (lines 4–8): if a significant increase in throughput is observed, i.e. the increase is a above a predefined tolerance threshold τ, a new learner is added to the respective GPU (line 6). Upon observing a decrease in throughput, we reduce the number of learners again (line 7).

Changing the number of learners is also beneficial in terms of statistical efficiency. Having initially few learners reduces the noise of stochastic gradients, fostering convergence of the reference models and thus the central average model. Eventually, though, this hampers the optimality of convergence as a smaller part of the loss space is explored.[1] By increasing the parallelism gradually, we avoid this issue. Intuitively, a small initial number of learners allows the central average model to reach the neighbourhood of the solution quickly, which is then comprehensively explored with increased training parallelism.

[1]Similar observations have been made regarding dynamic changes of the batch size [62] and learning rates [16].

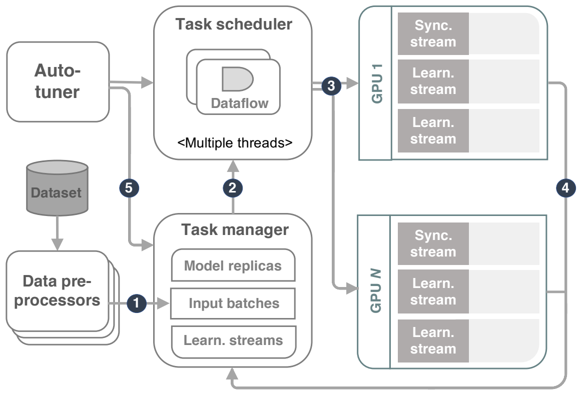


Figure 6: CROSSBOW design

4. CROSSBOW SYSTEM DESIGN

To support the training of deep learning models using SMA, the design of CROSSBOW has several unique features:

(1) Since we train multiple learners per GPU, CROSSBOW must share GPUs efficiently. CROSSBOW executes learners concurrently on a GPU by scheduling each to run on a separate GPU stream.

(2) We decide on the number of learners per GPU at runtime. The design of CROSSBOW must support changing the number of learners per GPU dynamically based on the available GPU resources.

(3) SMA synchronises all learners when they access the central average model. The design of CROSSBOW must implement this global synchronisation operation efficiently, and exploit concurrency during synchronisation to avoid bottlenecks.

Next we introduce the main component of CROSSBOW's design in §4.1. Based on the above requirements, we then describe hows tasks execute (§4.2) and are scheduled (§4.3), and how the number of learners is tuned dynamically (§4.4). We finish with an explanation of memory management in §4.5.

4.1 System overview

Figure 6 shows the main components of CROSSBOW:

(1) The **data pre-processors** read the training dataset into memory and arrange samples into batches, possibly after some transformations such as image decoding and cropping.

(2) The **task manager** controls the pools of model replicas, input batches and learner streams. It handles task completion events originating from the GPUs.

(3) The **task scheduler** assigns learning tasks to GPUs based on the available resources. It also triggers synchronisation operations at the end of each iteration.

(4) The **auto-tuner** monitors the training throughput and creates new learners on a GPU on-demand.

Figure 6 shows how CROSSBOW executes an iteration of SMA: the data pre-processors populate the input batch pool with pointers to data, one complete batch at a time (step ❶). The task scheduler checks if a model replica and a co-located learner stream are available (step ❷) and then schedules a learning task (step ❸). Upon completion, the task manager handles the event and returns the learner stream and the model replica to the pool (step ❹). It also

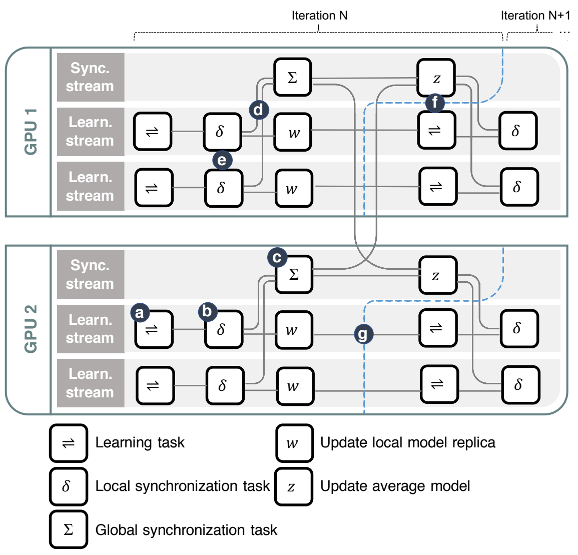


Figure 7: Dataflow graph on 2 GPUs with 2 learners each (The figure shows how tasks are scheduled on streams during two successive iterations of SMA, separated by a dashed line; solid lines are data dependencies between tasks.)

frees up a slot in the input batch pool, to be populated by one of the data pre-processors (step ❶). The auto-tuner monitors the rate at which learning tasks complete and interrupts the training by adding a new learner (step ❺).

4.2 Task execution

CROSSBOW trains a deep learning model by executing a dataflow graph, as shown in Figure 7. The dataflow graph consists of a set of *learning tasks* interleaved with *synchronisation tasks*. CROSSBOW represents the processing layers of a deep learning model as a graph of operators (e.g. a convolution or a matrix multiplication). A *learning task* encapsules multiple operators (see ⓐ). It takes as input a batch of training samples and a model replica and outputs a gradient. The task scheduler executes a learning task on any of the *learner streams* available on the GPU on which the replica used by that learning task resides.

Synchronisation tasks can be local or global: (i) a *local synchronisation task* computes the difference between a model replica and the central average model (see ⓑ). It uses the difference, together with the gradient of the corresponding learning task, to update the model replica. There is one local synchronisation tasks per replica on the same learner stream as the corresponding learning task; (ii) a *global synchronisation task* aggregates all local differences to update the central average model (see ⓒ). CROSSBOW allocates one average model replica per GPU, and uses a separate stream, the *synchronisation stream*, to execute global synchronisation tasks.

Placing a global execution barrier between the synchronisation and all preceding learning tasks would be expensive: it would block the task scheduler and delay the GPUs already waiting for tasks to complete. Instead, the task scheduler overlaps the synchronisation tasks from one iteration with the learning tasks of the next.

When overlapping tasks, the task scheduler considers *data dependencies* between tasks (see ⓓ). The horizontal direction in the figure represents time, and the vertical direction a spatial GPU partitioning based on streams. A line connecting two tasks denotes that the task on the right cannot start until the one on the left has completed.

Within an iteration, all local synchronisation tasks can execute concurrently (see ⓔ). They only require read-only access to the central average model in order to compute the difference from each model replica. The local synchronisation tasks, however, depend on the average model being updated consistently on each GPU by the global synchronisation tasks of the previous iteration.

Global synchronisation tasks can also execute concurrently within an iteration (see §3.3). The intra-GPU operations of a global synchronisation task execute as soon as their dependency to a local synchronisation task is satisfied (see ⓓ). The inter-GPU operations are implemented as a collective *all-reduce* primitive [60]. All-reduce creates a ring topology in which each GPU exchanges data partitions with its peers. A GPU reduces the partition data that it receives by combining it with its own, and eventually every GPU holds the final aggregated data. As a result, all-reduce evenly distributes the computation of the update for the average model among the GPUs.

The global synchronisation tasks in one iteration can execute concurrently with the learning tasks in the next (see ⓕ). Once the local synchronisation tasks have completed, each replica is updated. Now the task scheduler can issue the next learning task to the learner stream without waiting for other tasks to complete (see ⓖ).

4.3 Task scheduling

In each iteration of SMA, the task scheduler schedules one learning task for each model replica in the pool, followed by synchronisation tasks. As the task manager returns newly-updated model replicas to the pool, the task scheduler schedules further learning tasks and associates the next batch with a model replica on a first-come, first-served basis. Compared to round-robin scheduling, as used by PyTorch [52] or TensorFlow [1], this improves hardware efficiency because the task scheduler does not wait for a given replica to become available. After an assignment, the task scheduler hands the learning task over to one of its worker threads, in particular, one that runs on the same socket as the task's designated GPU. The worker thread issues the task to one of the GPU's streams as a sequence of kernels. All kernel calls to the GPU are non-blocking, and the thread returns immediately to schedule the next task.

A challenge for the task scheduler is to ensure that tasks that can run concurrently are executed concurrently by a GPU. As there are multiple streaming multi-processors (SMs) per GPU with no shared resources among them, a GPU can execute multiple tasks concurrently by assigning them to different sets of SMs. This favours our approach of training with small batch sizes because the kernels of a learning task usually require few SMs. As a solution, the task scheduler assigns concurrent tasks to different GPU streams, which enables the GPU to make efficient internal scheduling decisions: tasks submitted to the same stream execute in issue order; tasks on different streams may execute concurrently. A worker thread issues the task's kernels to its assigned streams, along with any event generators or handlers.

The task scheduler uses GPU events to preserve data dependencies between submitted tasks. If there is a dependency between tasks τ_1 and τ_2, the scheduler submits an *event generator* after τ_1 completes and an *event handler* before τ_2 begins. When the event generator on τ_1's stream executes, it signals τ_2's handler that all preceding tasks on that stream, τ_1 included, have completed; when the event handler on τ_2's stream executes, all subsequent tasks on that stream block until it receives a signal from τ_1's event generator. Analogous to the task scheduler, the task manager also uses multiple threads to handle task completion events in parallel, returning model replicas return to the pool in a timely manner.

A challenge when scheduling multiple learning tasks on the GPU is that their computational load varies both within and across operators. As a result, the GPU scheduler achieves varying degrees of concurrency across learning tasks, and training throughout does

Table 1: Deep learning benchmarks and datasets used

Model	Dataset	Input size (MB)	# Ops	Model size (MB)
LeNet	MNIST	179	24	4.2
ResNet-32	CIFAR-10	703	267	1.8
VGG-16	CIFAR-100	703	121	57.3
ResNet-50	ILSVRC 2012	1,073,375	384	97.5
ResNet-101	ILSVRC 2012	1,073,375	758	169.9

not scale linearly with the number of learners: some operators may run in parallel while others may block until sufficient resources become available. Since CROSSBOW cannot control how the GPU scheduler schedules operator kernels, it uses multiple threads to submit kernels. This enables the GPU scheduler to consider multiple kernels together and thus achieve better packing with lower variance.

4.4 Tuning learners

The *auto-tuner* changes the number of learners per GPU at runtime without prior knowledge of the trained model (e.g. the number and computational complexity of each model layer) or the training environment (e.g. the number and capability of each GPU). To this end, it implements the adaptation procedure introduced in §3.4 and formalised in Alg. 2.

The auto-tuner measures the training throughput by considering the rate at which learning tasks complete, as recorded by the task manager. As defined in Alg. 2, the number of learners per GPU is then increased or decreased based on the observed change in training throughput. Note that, on a server with homogeneous GPUs, the auto-tuner may measure only the throughput of a single GPU to adapt the number of learners for all GPUs.

The auto-tuner initiates the creation of a new learner after the learning and synchronisation tasks of the current iteration have been scheduled. Adding a learner to a GPU requires allocating a new model replica and a corresponding learner stream. The auto-tuner places temporarily a global execution barrier between two successive iterations (step ❻ in Figure 7), avoiding overlap with other tasks. The new model replica is initialised with the latest value of the average model. The auto-tuner also locks the resources pools, preventing access by the task scheduler or manager, while they are being resized.

Even for large models, such as ResNet-50, auto-tuning completes within milliseconds. The main overhead comes from the memory allocation and the initialisation of the model weights. Since model weights and their gradients are kept in contiguous memory, a single allocation call suffices.

4.5 Memory management

Data pre-processors transfer the input data from CPU to GPU memory using direct memory access (DMA). They write the pre-processed training samples to a page-aligned, page-locked circular buffer whose memory range is registered to the GPU's address space. This memory can be read directly by the GPU with higher bandwidth than unregistered, pageable memory. The buffer size must accommodate at least one input batch per learner (i.e. enough for a complete iteration of SMA). CROSSBOW uses double buffering to create a pipeline between data pre-processors and the task scheduler. When the pre-processors stall the pipeline because it takes more time to prepare the data on the CPU than to process it on a GPU, some or all of the input data transformations are scheduled on the GPUs as a preamble to each learning task.

Deep learning models require more memory to store the output of their dataflow operators than the model itself. For example, the ResNet-50 model is 97.5 MB in size but consumes 7.5 GB of memory to store the outputs from 384 operators. The memory requirement scales linearly with the batch size, as well as the number of learners. CROSSBOW must therefore reduce the memory footprint of each learner when training multiple of them per GPU.

CROSSBOW, similar to TensorFlow [1], MxNet [9] and SuperNeurons [66], devises an *offline memory plan* to reuse the output buffers of operators using reference counters. During initialisation, CROSSBOW traverses the operators of a learning task. When visiting an operator, it considers preceding operators and reuses an output buffer if the reference count is zero; otherwise it assumes that a new output buffer must be created. To account for data dependencies, it then decrements the reference counter of the operator's inputs and increments the counters of the operator's output. Such an offline plan reduces the memory footprint of a learner by up to 50% because outputs are mostly reused during the backwards phase.

The above approach cannot be used in CROSSBOW directly. Replicating the offline plan for each learner would lead to over-allocation of memory, when executing multiple learners per GPU. CROSSBOW exploits that, in practice, not all instances of the same operator would execute concurrently. This enables the sharing of some of the output buffers among learners on the same GPU using an *online memory plan*. For each operator, the task scheduler maintains a pool of output buffer pointers to GPU memory. Pools are shared by all learners on the same GPU. At runtime, when the task scheduler considers an operator for execution, it reuses the first available buffer in the output buffer pool; if none are available, it allocates a new one. The task scheduler increments the reference counter of the operator's output according to its data dependencies and issues the kernel on a stream. When the operator completes, the task manager decrements the reference counter of the operator's input and output buffers. A buffer with zero references returns to the pool, and it can be reused by other learning tasks. When neither a complete model replica nor a complete learning task fits into GPU memory, CROSSBOW could exploit related work on handling large models [54, 12] or large operator outputs [24].

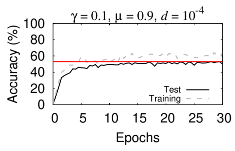
5. EVALUATION

In this section, we evaluate the performance of our CROSSBOW prototype when training on a multi-GPU server. We begin by comparing its behaviour against TensorFlow [1] using five deep learning macro-benchmarks, as we vary the number of GPUs and the number of learners per GPU (§5.2). Next, we assess the impact of CROSSBOW's core features through a set of micro-benchmarks: we explore the benefits of executing multiple learners per GPU in terms of statistical and hardware efficiency (§5.3); and we measure the ability of the auto-tuning mechanism to identify the best number of learners per GPU (§5.4) and the impact of the model averaging used in SMA (§5.5). Finally, we quantify the efficiency of the synchronisation task implementation (§5.6).

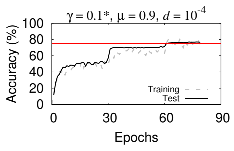
5.1 Experimental set-up

Experiments are conducted on a server with two Intel Xeon E5-2650 v3 2.3 GHz CPUs (20 CPU cores in total) and 256 GB of RAM. The server has 8 NVIDIA GeForce GTX Titan X (Pascal) GPUs, each with 3,072 cores and 12 GB of RAM, connected via PCIe 3.0 ($\times$16). It runs Linux kernel 4.4 with the NVIDIA driver 367.57 and CUDA 8.0 with the cuDNN 6.0 library. We also use another server with 8 NVIDIA Tesla V100 (Volta) GPUs, each with 5,120 cores, 640 Tensor cores, and 16 GB of RAM, connected in a mesh topology with NVLink.

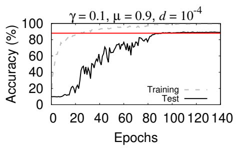
As a baseline, we use TensorFlow version 1.4 (1.13 on the Volta GPUs). When deployed on more than one GPU, TensorFlow uses



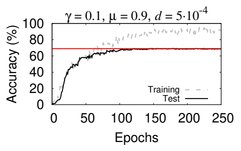
(a) ResNet-50



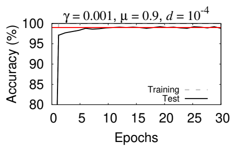
(b) ResNet-101



(c) ResNet-32

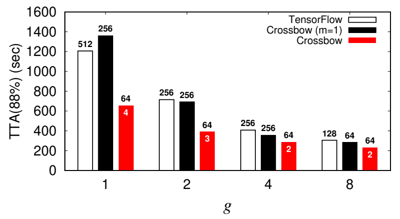


(d) VGG

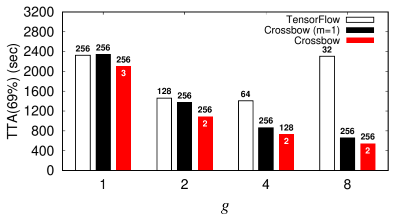


(e) LeNet

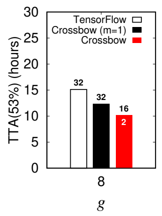
Figure 8: TensorFlow's convergence over epochs (Test and training accuracy over epochs for a given learning rate γ, momentum μ, and weight decay d. In TensorFlow, ResNet-101 uses the linear scaling rule [16]. The red lines show our test accuracy targets.)



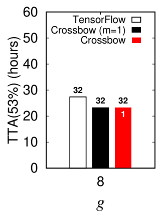
(a) ResNet-32



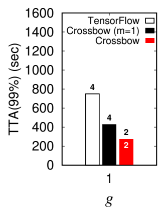
(b) VGG



(c) ResNet-50



(d) ResNet-101



(e) LeNet

Figure 9: Time-to-accuracy for five deep-learning models (Numbers on top of the bars report the batch size per GPU that achieved that accuracy; numbers inside the CROSSBOW bars report the best number of model replicas per GPU.)

parallel synchronous SGD (§2.3): GPUs have an identical replica of the model and, after each iteration, they exchange the computed gradients using NCCL's all-reduce. Each GPU is managed by a dedicated thread on the CPU, and TensorFlow allocates one stream for computation and one for data transfers. In Crossbow, SMA is used (§3.2), and GPUs exchange model differences (rather than gradients) through NCCL. Each GPU then updates its reference model copy with the accumulated difference in a consistent manner.

We compare performance using TensorFlow's suite of benchmarks [64] shown in Table 1. We select a mix of models, representing different sizes and shapes of networks. This includes both small (LeNet), large (ResNet-50) and very large networks (ResNet-101) as well as deep and low-dimension networks (ResNet-32) and shallow and high-dimension ones (VGG). To enable a fair comparison, we configure both systems with the same data augmentation, model variable initialisation and hyper-parameter settings. Following common practices, the learning rate in ResNet-32 is multiplied by 0.1 at epochs 80 and 120 [18]; the learning rate in VGG is halved every 20 epochs [65].

In our experiments, we vary two main parameters: the batch size per learner (b) and the number of model replicas (m). Our main metric is the time-to-accuracy TTA (x), defined as the time taken for the median test accuracy of the last 5 epochs to be equal or above a given threshold x. For each of the four deep learning models, we choose different thresholds based on the highest accuracy reached by TensorFlow in our set-up. According to the results in Figure 8, we select the following thresholds: 99% (LeNet), 88% (ResNet-32), 69% (VGG-16), 53% (ResNet-50) and 74.9% (ResNet-101). Higher accuracies can be achieved by leveraging dynamic hyper-parameter tuning. As discussed in §2.4, these techniques, however, are model- and architecture-specific and lack generality. In contrast, the goal of our evaluation is to compare the different approaches underlying CROSSBOW and TensorFlow under uniform settings.

5.2 Scalability

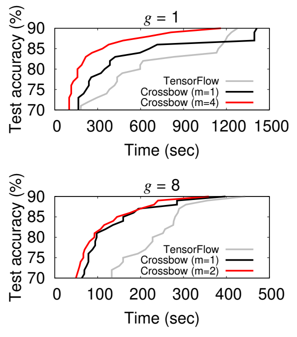
We begin by comparing the performance of CROSSBOW and TensorFlow when scaling the number of GPUs. Figure 9 shows the performance of CROSSBOW and TensorFlow for the five deep learning models. For ResNet-32 and VGG (Figures 9a and 9b), we change the number of GPUs from 1 to 8; for ResNet-50 (Figure 9c) and ResNet-101 (Figure 9d), we show the results for 8 GPUs only due to the long time required to train with few GPUs (e.g. with 1 GPU, TensorFlow takes more than 5 days for training ResNet-50); for LeNet (Figure 9e), we only run the experiment with 1 GPU because, given the small model size, training on multiple GPUs leads to a higher training time due to the synchronisation overhead.

First we consider the performance of CROSSBOW when using only one learner per GPU, i.e. m=1 (black bar in Figure 9). In this configuration, for ResNet-32 and VGG, CROSSBOW achieves a performance comparable or slightly worse than TensorFlow when training on a small number of GPUs (1 or 2). The reason is that both ResNet and VGG are relatively compute-intensive models. With few learners, the synchronisation overhead is limited and, hence, the benefits of SMA (§3) and the fast task scheduling (§4.3) are less relevant. As we increase the number of GPUs to 4 or 8, the number of learners and the amount of synchronisation among them increases proportionally. The results now highlight the higher performance of SMA compared to TensorFlow's S-SGD scheme, with up to a 72% reduction in TTA for VGG with 8 GPUs (and 7% for ResNet-32, respectively). A similar improvement shows for ResNet-50 in Figure 9c and ResNet-101 in Figure 9d: CROSSBOW achieves an 18% reduction in TTA in both networks (8 GPUs).

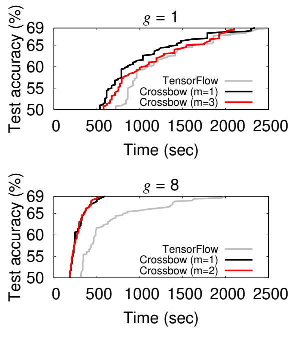
Notably, when training VGG, TensorFlow scales poorly with the number of GPUs. Increasing from 2 to 4 GPUs does not noticeably reduce TTA and with 8 GPUs the TTA almost doubles. The reasons are twofold: (i) our batch size exploration shows that the best TTA is achieved with an (aggregate) batch size of 256, which means that, with 8 GPUs, the batch size per GPU is only 32, leading to under-utilisation; and (ii) with more GPUs, the communication and synchronisation costs increase, degrading performance.

CROSSBOW offers a benefit even when training on a single GPU (one learner in total) if the model is not compute-intensive. For LeNet, each learning task takes less than 1 ms (compared to ∼220 ms for ResNet-50) and, hence, the scheduling overhead becomes critical. By leveraging its efficient task scheduler, CROSSBOW yields a significant TTA reduction (43%) compared to TensorFlow with one learner (see Figure 9e).

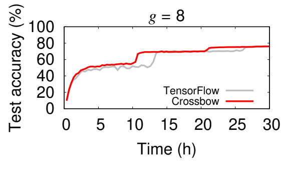
Thus far, we have focused on a CROSSBOW configuration with m=1. A key advantage of CROSSBOW is its ability to increase



(a) ResNet-32



(b) VGG



(c) ResNet-101

Figure 10: Convergence over time

hardware efficiency without affecting statistical efficiency by adding more learners per GPU ($m>1$). In this case, CROSSBOW significantly improves performance even with few GPUs. For ResNet-32, CROSSBOW with $m=4$ achieves a 46% TTA reduction with 1 GPU and a 24% TTA reduction with $m=2$ and 8 GPUs; for VGG, the reduction is 10% for 1 GPU and 77% for 8 GPUs. Similar benefits with $m=2$ also show for ResNet-50 (33% TTA reduction, corresponding to 5 hours) and LeNet (63% TTA reduction). As explained in §3, $m>1$ increases the GPU throughput without necessitating a larger batch size that would affect statistical efficiency. For ResNet-50, CROSSBOW with $m=2$ uses a small batch size of $b=16$ compared to an aggregate batch size of $32\times8=256$ for TensorFlow (64 and 1,024 for ResNet-32, respectively).

While we choose to use the highest accuracy reached by TensorFlow as the threshold x in the TTA (x) metric, similar improvements hold for other accuracy thresholds. In Figure 10, we plot TTA over time for ResNet-32 and VGG with 1 and 8 GPUs and for ResNet-101 with 8 GPUs only due to the high computation time. CROSSBOW achieves high accuracy within a few minutes: with 8 GPUs, it takes 92 seconds to exceed a 80% accuracy for ResNet-32 compared to 252 seconds for TensorFlow—a 63% TTA reduction. Similarly, CROSSBOW achieves a 74% TTA reduction for VGG. This indicates that SMA converges quickly to a region containing good minima. Due to its large size, for ResNet-101, it is not possible to fit more than one learner per GPU. As shown in Figure 10c, even in this case, CROSSBOW outperforms TensorFlow due to its more efficient synchronisation algorithm and optimised implementation, especially until epoch 30, where gradient variance amongst learners is high.

5.3 Statistical and hardware efficiency

CROSSBOW with $m>1$ outperforms TensorFlow because it can increase hardware efficiency without negatively impacting statistical efficiency. As discussed in §2.4, Tensorflow must increase the batch size to improve hardware efficiency but this comes at the cost of reduced statistical efficiency. In contrast, CROSSBOW uses the number of learners m as an additional control parameter to increase hardware efficiency without having to resort to larger batch sizes.

We show this in Figure 11 by plotting the GPU utilisation for TensorFlow and CROSSBOW as we increase the batch size. With

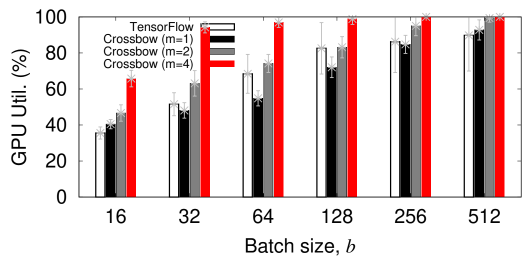


Figure 11: GPU utilisation for TensorFlow and CROSSBOW for various batch sizes (This experiment uses ResNet-32.)

a batch size of 64, TensorFlow is unable to fully saturate the GPU (utilisation is 68%), and it must resort to a batch size of at least 512 to achieve nearly full utilisation. As we showed in Figure 2c, this comes at a cost of reduced statistical efficiency, resulting in a larger number of epochs for convergence. By increasing the number of learners per GPU, CROSSBOW fully utilises the GPUs even for small batch sizes: for $b=64$ with $m=4$ learners, CROSSBOW utilises 97% of the GPU (74% for $m=2$, respectively).

In Figures 12 and 13, we show how hardware and statistical efficiency, and the resulting TTA, are affected by m when using 1 and 8 GPUs, respectively. We only report the results for ResNet-32 ($b=64$) but similar trends hold for the other models. Compared to the experiments in Figure 9a, we lower the target accuracy for TTA to 80% as otherwise the results would be skewed by the change in the learning rate at epoch 80 (see §5.1).

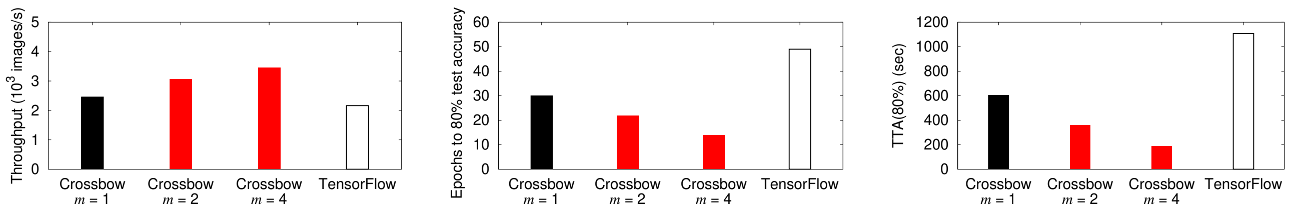
When training with 1 GPU, using $m=4$ increases the throughput by a factor of $1.4\times$ compared to the case with a single learner (Figure 12a) because the multiple learners fully utilise a GPU. Interestingly, this improves statistical efficiency as well—the number of epochs required to converge drop from 30 ($m=1$) to 14 ($m=4$) (see Figure 12b). This is because multiple model replicas can explore a larger portion of the space in parallel while the average model can reduce the variance among them, thus requiring fewer epochs to find good minima. As a result of the higher hardware *and* statistical efficiencies, the TTA is also reduced by $3.2\times$ (Figure 12c).

In contrast, the behaviour with 8 GPUs is somewhat different. While $m=2$ yields higher throughput ($1.3\times$), increasing the number of learners to $m=4$ does not further improve the throughput (Figure 13a). The reason is that, with 8 GPUs and 4 learners per GPU, the overall number of learners is 32, which introduces a significant amount of synchronisation overhead. In terms of statistical efficiency (Figure 13b), increasing the number of learners to $m=2$ does not significantly affect the number of epochs to converge but increasing it further leads to reduced statistical efficiency—with 32 learners in total, there is not enough stochastic noise in the training process, which makes it harder for the average model to escape bad minima. In this case, $m=2$ represents the best trade-off because it allows for higher hardware efficiency without noticeably worsening statistical efficiency. Indeed, this configuration reduces training time by $1.3\times$ (Figure 13c).

These results show that increasing the number of learners per GPU is beneficial to reduce the training time. However, identifying the correct number of learners is crucial to achieving the best performance, as we show next.

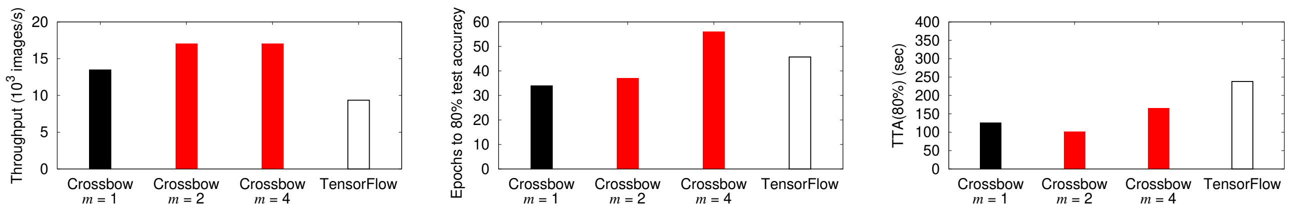
5.4 Selecting the number of learners

To select the number of learners per GPU m, CROSSBOW progressively increases m until the throughput (expressed as images processed per second) stops improving (see §3.4). To validate this approach, Figure 14 shows the TTA and throughput achieved for



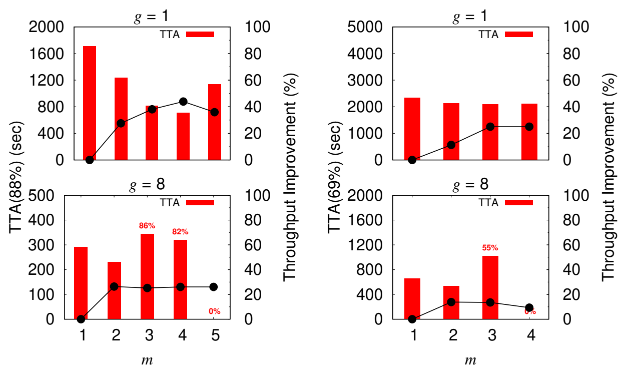
(a) Hardware efficiency **(b) Statistical efficiency** **(c) Time to accuracy**

Figure 12: Trade-off between hardware and statistical efficiency with 1 GPU (This experiment uses ResNet-32 and b=64.)



(a) Hardware efficiency **(b) Statistical efficiency** **(c) Time to accuracy**

Figure 13: Trade-off between hardware and statistical efficiency with 8 GPUs (This experiment uses ResNet-32 and b=64.)



(a) ResNet-32 (b=64) **(b) VGG (b=256)**

Figure 14: Varying the number of models (This experiment uses the same batch sizes as Figure 9. The best number of models (i.e. minimising TTA) is the one that saturates training throughput.)

increasing values of m when training ResNet-32 and VGG with 1 and 8 GPUs, respectively.

For ResNet-32, we observe that, with 1 GPU, the throughput grows until m=4 and then decreases; with 8 GPU, the throughput improves until m=2 and then remains relatively constant for m>2. As predicted by our auto-tuning technique, the lowest TTA is achieved with m=4 for 1 GPU and m=2 for 8 GPUs, respectively. Similarly, for VGG, the throughput plateaus at m=3 for 1 GPU and m=2 for 8 GPUs, respectively, which correspond to the lowest values of TTA. This demonstrates the ability of CROSSBOW's auto-tuning technique to identify quickly the correct number of learners in order to minimise TTA.

5.5 Synchronisation model

Now we compare the performance of SMA when training ResNet-32 against existing synchronisation algorithms, namely Hogwild! (Hogwild) [55] and Elastic Averaging SGD (EA-SGD) [73].

Figure 15 shows that SMA outperforms Hogwild by 1.8× with 1 GPU and m=4 and by 1.5× with 8 GPUs and m=2 (16 learners). This is because, in Hogwild, learners do not synchronise when they update their GPU-resident reference model, and as a result they compute gradients using a stale model. This effect is less pronounced with more GPUs because the synchronisation overheads limit staleness. SMA also significantly reduces the TTA compared to EA-SGD (by 1.1× with 1 GPU and by 1.4× with 8 GPUs, respectively). The gap increases with the number of GPUs because the more learners are used, the lower the (asymptotic) variance of the average model becomes, making it hard to escape from local minima. Therefore, without including momentum, the average model converges more slowly. In addition, SMA restarts the averaging process when the learning rate changes to avoid oscillating behaviour (see §3.2), which also contributes to a lower TTA.

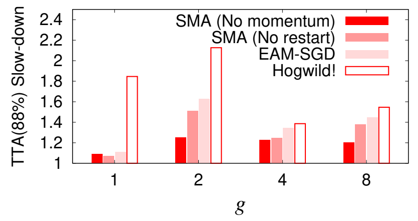


Figure 15: Comparison of SMA with EAM-SGD and Hogwild! in CROSSBOW (This experiment uses ResNet-32.)

To assess the individual impact of each of these two factors, we include in the figure two variants of SMA: one that does not use momentum on the central average model, SMA(No momentum), and one that does not restart the synchronisation algorithm, SMA(No restart). The results show that both optimisations are needed, with momentum providing the greater benefit.

CROSSBOW synchronises the different replicas with the average model in each iteration. The authors of EA-SGD propose to synchronise every τ>1 iterations to reduce the communication overhead. We study the impact of this optimisation on CROSSBOW in Figure 17. While less frequent synchronisation (τ>1) increases the overall throughput (up to 31% for τ=4 compared to τ=1), it negatively affects convergence speed, resulting in a higher TTA (53% longer with τ=4 compared to τ=1). Therefore, we always use τ=1.

5.6 Synchronisation efficiency

In the above experiments, we focused on the *algorithmic* benefits of the SMA synchronisation approach. Now, instead, we want to

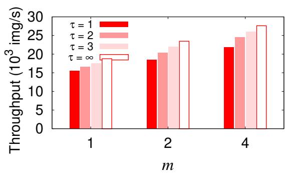


Figure 16: Effect of synchronisation frequency on H/W efficiency in CROSSBOW (This experiment uses ResNet-32 and g=8.)

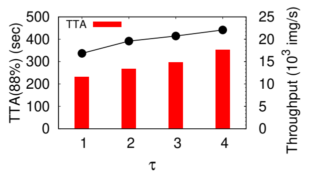


Figure 17: Effect of synchronisation frequency on time-to-accuracy in CROSSBOW (This experiment uses ResNet-32, g=8 and m=2.)

understand the performance of our synchronisation *implementation*. Similar to Figure 17, we conduct an experiment in which we measure the throughput achieved by CROSSBOW for increasing values of τ, including also the case with *no* synchronisation at all. Since we are only interested in the performance aspect, we do not report the TTA (we have already shown that τ=1 yields the shortest TTA). The goal of this experiment is rather to observe the increase in throughput as we reduce the synchronisation frequency. The rationale is that, if synchronisation incurred a high cost, the throughput would drastically increase as we reduce the amount of synchronisation.

Contrary to this expectation, the results in Figure 16 show that throughput is only marginally higher, growing from 15,500 images/s with τ=1 to 18,500 images/s with no synchronisation at all (20%) with m=1 (27% for m=4, respectively). This indicates that CROSSBOW's synchronisation implementation is well-optimised and introduces only a modest overhead.

6. RELATED WORK

Training with multiple GPUs. Machine learning systems use data parallelism, model parallelism, or a mix or both (see DistBelief [13] and, more recently, FlexFlow [29]) to decrease training time. TensorFlow [1], PyTorch [52], MXNet [9], CNTK [59] and Caffe2 [16] exploit data parallelism by default and use S-SGD as their de-facto training algorithm. S-SGD, however, couples the batch size and the number of GPUs. To compensate for the loss of statistical efficiency incurred by large batch sizes, users tune other hyper-parameters using techniques such as auto-tuning [70, 27], scaling [34] and warming up [16, 34] the learning rate, auto-tuning the batch size [62], auto-tuning the momentum [72] and others. The effectiveness of these tuning techniques is problem-specific [14], and users invest substantial time to find a scalable set-up [27]. CROSSBOW explores a different direction by decoupling the batch size and the number of GPUs. It provides a design for a task engine that can fully utilise a multi-GPU server even when the batch size is small.

Increasing GPU utilisation. There are proposals to improve the hardware utilisation of machine learning systems using cooperative scheduling. ModelBatch [43] and NVIDIA's Multi-Process Service train multiple deep learning models on a GPU in a cooperative manner, but the problem of model synchronisation remains unresolved. Litz [53] explores the scheduling of training and synchronisation tasks on CPUs, but its lack of GPU support makes it ineffective for deep learning. Ray [42] trains deep learning models using cooperative GPU tasks but only shares GPUs using time-sharing. In contrast, CROSSBOW provides efficient concurrent execution of learning and synchronisation tasks on GPUs, which is the key to achieve high hardware efficiency when training deep learning models with small batch sizes.

Asynchronous training. Prior work also attempts to improve hardware utilisation using asynchronous SGD [7], often at the expense of statistical efficiency [8]. Hogwild! [55] and Dogwild! [45] do not specify data dependencies between learning and synchronisation tasks: all workers access a central model concurrently, leading to higher training throughput. To compensate for the loss in statistical efficiency, DimmWitted [71] coordinates parallel replicas in a CPU-oriented NUMA architecture. Each NUMA node has its own model replica shared by its cores. Within a node, cores update a shared replica asynchronously. In contrast to these efforts, CROSSBOW follows a synchronous training approach and therefore does not compromise statistical efficiency with stale updates.

Model averaging was originally proposed as an asynchronous method to distribute training. Polyak-Ruppert's *averaged SGD* [51, 50, 58] first demonstrated that an average model can asymptotically converge faster to a solution than the individual model replicas used to compute it. In practice, it is difficult to find this asymptotic region [69], especially with models that have complex loss spaces [10]. To improve the statistical efficiency of model averaging, recent studies [6, 39] propose to use the average model to correct the trajectory of model replicas, but the effectiveness of this approach was shown only for non-deep-learning problems.

In deep learning, *elastic averaging SGD* (EA-SGD) [73] uses the average model to correct model replicas occasionally, keeping the communication cost low. *Asynchronous decentralised SGD* (AD-SGD) [40] further reduces server communication traffic by requiring each replica to perform model averaging with only one worker per iteration. Compared to these techniques, SMA is a synchronous algorithm that shares and maintains a consistent view of the central average model across all learners in each iteration. SMA further improves the statistical efficiency of model averaging by adopting momentum (see §3.2) to correct the average model. Upon changes to the hyper-parameters during training, SMA also restarts the averaging process to preserve statistical efficiency.

Distributed training. When scaling the training of deep learning models in distributed clusters, a parameter server (PS) [38] design is the de-facto approach. In contrast to CROSSBOW, which improves the training performance with small batch sizes on a single multi-GPU server, PS-based systems address the challenges of using a cluster for distributed learning, including the handling of elastic and heterogeneous resources [23, 30], the mitigation of stragglers [13, 11, 17], the acceleration of synchronisation using hybrid hardware [12], and the avoidance of resource fragmentation using collective communication [67, 60, 27]. Similar to prior model averaging systems [73], CROSSBOW could adopt a PS design to manage its average model in a distributed deployment. We view the distribution of CROSSBOW as future work.

7. CONCLUSIONS

CROSSBOW improves hardware efficiency when training with the preferred batch size, however small, with a low loss of statistical efficiency. It trains multiple model replicas on the same GPU, tuning their number automatically as to maximise training throughput. Despite training many more model replicas compared to existing approaches, CROSSBOW avoids reduced statistical efficiency using SMA. The latter is a new training algorithm in which replicas *independently* explore the solution space with gradient descent, but adjust their search *synchronously* based on the trajectory of a globally-consistent central average model. Our experimental results show that CROSSBOW shortens the time-to-accuracy during training by up to 4× compared to TensorFlow. CROSSBOW is available at: `https://github.com/lsds/Crossbow`.

Ackwnoledgements. This material is based upon work partially supported by Google Cloud and Huawei.

8. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Amazon EC2 Instance Types, 2017. `https://aws.amazon.com/ec2/instance-types/`.

[3] S. Ö. Arik, M. Chrzanowski, A. Coates, G. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, J. Raiman, S. Sengupta, and M. Shoeybi. Deep Voice: Real-time Neural Text-to-Speech. arXiv:1702.07825 [cs.CL], Feb. 2017.

[4] L. Bottou. On-line Learning and Stochastic Approximations. In D. Saad, editor, *On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press, New York, NY, USA, 1998.

[5] L. Bottou, F. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2):223–311, 2018.

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, Jan. 2011.

[7] S. Chaturapruek, J. C. Duchi, and C. Ré. Asynchronous Stochastic Convex Optimization: The Noise Is in the Noise and SGD Don't Care. In *28th International Conference on Neural Information Processing Systems (NIPS)*, 2015.

[8] J. Chen, R. Monga, S. Bengio, and R. Józefowicz. Revisiting Distributed Synchronous SGD. arXiv:1604.00981 [cs.LG], Apr. 2016.

[9] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC], Dec. 2015.

[10] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The Loss Surfaces of Multilayer Networks. In *18th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015.

[11] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (ATC)*, 2014.

[12] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *11th European Conference on Computer Systems (EuroSys)*, 2016.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large Scale Distributed Deep Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, 2012.

[14] J. Dean, D. Patterson, and C. Young. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro*, 38(2):21–29, Mar. 2018.

[15] A. Defossez and F. Bach. Averaged Least-Mean-Squares: Bias-Variance Trade-offs and Optimal Sampling Distributions. In *18th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015.

[16] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677 [cs.CV], June 2017.

[17] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV], Dec. 2015.

[19] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov. 2012.

[20] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *26th International Conference on Neural Information Processing Systems (NIPS)*, 2013.

[21] S. Hochreiter and J. Schmidhuber. Flat Minima. *Neural Computation*, 9(1):1–42, Jan. 1997.

[22] E. Hoffer, I. Hubara, and D. Soudry. Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks. In *30th International Conference on Neural Information Processing Systems (NIPS)*, 2017.

[23] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *PVLDB*, 11(5):566–579, 2018.

[24] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[25] P. Jain, S. M. Kakade, R. Kidambi, P. Netrapalli, and A. Sidford. Parallelizing Stochastic Gradient Descent for Least Squares Regression: Mini-batching, Averaging, and Model Misspecification. *Journal of Machine Learning Research*, 18(223):1–42, 2018.

[26] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. J. Storkey. Three Factors Influencing Minima in SGD. arXiv:1711.04623 [cs.LG], Nov. 2017.

[27] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. arXiv:1807.11205 [cs.LG], July 2018.

[28] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *22nd ACM International Conference on Multimedia (MM)*, 2014.

[29] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *35th International Conference on Machine Learning (ICML)*, 2018.

[30] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-Aware Distributed Parameter Servers. In *2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.

[31] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation.

Transactions of the Association for Computational Linguistics, 5:339–351, 2017.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[33] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv:1609.04836 [cs.LG], Sept. 2016.

[34] A. Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. arXiv:1404.5997 [cs.NE], Apr. 2014.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*, 2012.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998.

[37] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[38] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient Mini-batch Training for Stochastic Optimization. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014.

[40] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *35th International Conference on Machine Learning (ICML)*, 2018.

[41] D. Masters and C. Luschi. Revisiting Small Batch Training for Deep Neural Networks. arXiv:1804.07612 [cs.LG], Apr. 2018.

[42] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[43] D. Narayanan, K. Santhanam, and M. Zaharia. Accelerating Model Search with Model Batching. In *1st Conference on Systems and Machine Learning (SysML)*, SysML '18, 2018.

[44] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27:372–376, 1983.

[45] C. Noel and S. Osindero. Dogwild! – Distributed Hogwild for CPU and GPU. Distributed Machine Learning and Matrix Computations NIPS 2014 Workshop, 2014.

[46] NVIDIA Collective Communications Library (NCCL), 2018. `https://developer.nvidia.com/nccl`.

[47] NVLink Fabric Multi-GPU Processing, 2018. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[48] Octoputer 4U 10-GPU Server with Single Root Complex for GPU-Direct, 2018. `https://www.microway.com/product/octoputer-4u-10-gpu-server-single-root-complex/`.

[49] B. Polyak. Some Methods of Speeding up the Convergence of Iteration Methods. *USSR Computational Mathematics and Mathematical Physics*, 4:1–17, Dec. 1964.

[50] B. Polyak. New Stochastic Approximation Type Procedures. *Avtomatica i Telemekhanika*, 7(7):98–107, Jan. 1990.

[51] B. Polyak and A. Juditsky. Acceleration of Stochastic Approximation by Averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

[52] PyTorch, 2018. `https://pytorch.org`.

[53] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *2018 USENIX Annual Technical Conference (ATC)*, 2018.

[54] C. Qin, M. Torres, and F. Rusu. Scalable Asynchronous Gradient Descent Optimization for Out-of-core Models. *PVLDB*, 10(10):986–997, 2017.

[55] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *24th International Conference on Neural Information Processing Systems (NIPS)*, 2011.

[56] H. Robbins and S. Monro. A Stochastic Approximation Method. *Ann. Math. Statist.*, 22(3):400–407, Sept. 1951.

[57] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. In D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[58] D. Ruppert. Efficient Estimators from a Slowly Convergent Robbins-Monro Process. Technical Report 781, School of Operations Research and Industrial Enginnering, Cornell University, Ithaka, New York 14853-7501, Feb. 1988.

[59] F. Seide and A. Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[60] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in Tensor Flow. arXiv:1802.05799 [cs.LG], Feb. 2018.

[61] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. arXiv:1811.03600 [cs.LG], Nov. 2018.

[62] S. L. Smith, P. Kindermans, and Q. V. Le. Don't Decay the Learning Rate, Increase the Batch Size. arXiv:1711.00489 [cs.LG], Nov. 2017.

[63] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *30th International Conference on Machine Learning (ICML)*, 2013.

[64] TensorFlow Benchmarks, 2018. `https://github.com/tensorflow/benchmarks`.

[65] VGG16 models for CIFAR-10 and CIFAR-100 using Keras, 2018. `https://github.com/geifmany/cifar-vgg`.

[66] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[67] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[68] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer,

A. Stolcke, D. Yu, and G. Zweig. The Microsoft 2016 Conversational Speech Recognition System. arXiv:1609.03528 [cs.CL], Jan. 2017.

[69] W. Xu. Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent. arXiv:1107.2490 [cs.LG], Dec. 2011.

[70] Y. You, I. Gitman, and B. Ginsburg. Large Batch Training of Convolutional Networks. arXiv:1708.03888 [cs.CV], Sept. 2017.

[71] C. Zhang and C. Ré. DimmWitted: A Study of Main-memory Statistical Analytics. *PVLDB*, 7(12):1283–1294, 2014.

[72] J. Zhang and I. Mitliagkas. YellowFin and the Art of Momentum Tuning. arXiv:1706.03471 [stat.ML], June 2017.

[73] S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with Elastic Averaging SGD. In *28th International Conference on Neural Information Processing Systems (NIPS)*, 2015.