Taming Hyper-parameters in Deep Learning Systems

Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, Peter Pietzuch
Imperial College London

Abstract

Deep learning (DL) systems expose many tuning parameters ("hyper-parameters") that affect the performance and accuracy of trained models. Increasingly users struggle to configure hyper-parameters, and a substantial portion of time is spent tuning them empirically. We argue that future DL systems should be designed to help manage hyper-parameters. We describe how a distributed DL system can (i) remove the impact of hyper-parameters on both performance and accuracy, thus making it easier to decide on a good setting, and (ii) support more powerful dynamic policies for adapting hyper-parameters, which take monitored training metrics into account. We report results from prototype implementations that show the practicality of DL system designs that are hyper-parameter-friendly.

1. Introduction

As deep learning (DL) models are used across many application domains such as speech and image classification [22, 6], users face the challenge of tuning hyper-parameters during the training of DL models to achieve high accuracy and good training performance. Today's distributed DL systems such as TensorFlow [1], PyTorch [44], and MXNet [11] expose a wide range of hyper-parameters, including the training batch size [46], the learning rate [46], momentum [40], floating point precision [21] and so on. Evidence from practitioners suggests that users spend a tremendous amount of resources tuning these hyper-parameters [35, 24, 20], mostly empirically, when training complex state-of-the-art models such as ResNet [22] and BERT [15]. Such manual hyper-parameter tuning leads to the highest reported accuracies and fastest training times in DL competitions such as ImageNet [14] and SQuAD [45].

The plethora of proposed approach for automated hyper-parameter tuning [50, 18, 25] shows the desire to remove the burden of hyper-parameter tuning from users, but current approaches either exhaustively search the space of possible settings at a high time and resource cost [49, 35, 17], or offer only brittle tuning strategies based on model-specific heuristics [20, 24, 54].

We observe that there are several reasons why current DL systems make it cumbersome for users to tune hyper-parameters: (1) they expose hyper-parameters that affect both training accuracy and performance, often introducing a tension between the two goals. Users now need to strike a balance between achieving the highest accuracy and training the model in the shortest amount of time, facing the challenge that there is no single good setting [26]; (2) DL systems do not support expressive strategies for tuning hyper-parameters. Typically hyper-parameters must be configured offline as part of a driver program, e.g. by defining a static training schedule that adapts parameters such as the learning rate or batch size based on the epoch number in the training process [22, 49]. This makes tuning policies brittle [20] and unable to react to the progress of the training process [35].

To tame hyper-parameters, we believe that future scalable DL systems must be designed with hyper-parameters friendliness in mind. Similar to self-driving database systems [39], which avoid exposing configuration parameters, DL systems should achieve two goals:

(1) DL systems should **remove performance-focused hyper-parameters** as much as possible. Users should be able to freely adjust hyper-parameters to achieve the highest model accuracy without having to worry about reducing training performance. A DL system should ensure the best possible resource utilisation and parallelisation of computation independently of hyper-parameter settings.

(2) DL systems should provide richer support for **policies that dynamically adapt hyper-parameters** during the training process. They should continuously monitor metrics, such as gradient noise scale [35], gradient variance [53] and second-order gradient [33], and allow users to express dynamic policies that react to monitored metrics.

We demonstrate the above two goals can by achieved by two proof-of-concept implementations, CROSSBOW and KUNGFU. CROSSBOW [28] removes the performance effect of a critical hyper-parameter, namely the batch size, enabling

ISBN [to be supplied]. . . $15.00

a user to train a DL model with a small batch size while still achieving high performance through good scalability; KUNGFU embeds monitoring functionality in a distributed DL system, and exposes the monitored metrics to support expressive policies for advanced hyper-parameter adaptation.

2. The challenge of hyper-parameters

Next we describe the use of hyper-parameters in DL systems and the associated challenges.

2.1 Hyper-parameters in deep learning systems

When using DL models, increasing the amount of training data and the size of the model improves their accuracy [22, 12]. For training, DL systems therefore exploit the parallelism of modern hardware accelerators such as GPUs. Computation is typically expressed as a *dataflow* graph [1], which consists of individual operators that can be scaled out.

The DL system trains models using labelled samples, split into training and test data. A model gradually "learns" to predict the labels by adjusting its *model weights* based on the error. It 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. A key metric is *test accuracy*, which measures the ability of the model to make predictions "in the wild".

A DL system iteratively refines model weights until it achieves a desired test accuracy. Let w be a vector of the 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. A DL system tries to find w^* that minimises the average loss usually through *mini-batch stochastic gradient descent* (SGD) [46, 9, 10]. More formally,

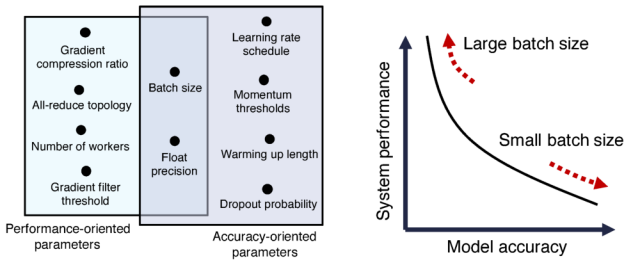
$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{b} \sum_{\mathsf{x} \in \mathsf{B}_n} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \qquad (1)$$

where γ_n is the learning rate in the n-th iteration of the algorithm, and B_n is a batch of b training samples, and $\nabla \ell$ is the gradient of the loss function, averaged over the batch samples. The learning rate γ_n and the batch size $|\mathsf{B}_n|$ are examples of *hyper-parameters*, as they affect the overall training process and must be set by the user.

To scale out the computation across multiple processors or accelerators, DL systems often exploit data parallelism. In *parallel synchronous SGD* (S-SGD), K parallel workers share model replicas and compute gradients for distinct partitions of training data locally. Local gradients are averaged to correct the shared model:

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{Kb} \sum_{\mathsf{j} < \mathsf{K}} \sum_{\mathsf{x} \in \mathsf{B}_{n,j}} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \qquad (2)$$

As shown in Figure 1a, hyper-parameters can be grouped into two classes: (1) *performance-oriented* hyper-parameters, such as the number of parallel workers and the communication topology, which critically affect the system's processing throughput; and (2) *accuracy-oriented* hyper-parameters, such as the learning rate and momentum, which impact the optimality of local minimas and thus the overall accuracy of the trained model.



(a) Hyper-parameters that affect performance and accuracy

(b) Trade-off between batch sizes

Figure 1: Hyper-parameters and their challenges

In practice, it is necessary to apply a wide range of tuning strategies to hyper-parameters. To reduce the time to reach a desired test accuracy (*time-to-accuracy*), users must carefully manage the performance-oriented hyper-parameters. To overcome communication bottlenecks, for example, a user can (i) tune a gradient filter threshold to skip small enough gradients [31], (ii) set a compression ratio for gradients to save bandwidth [32], (iii) choose different all-reduce topologies for gradients [24], and (iv) adjust the float precision to reduce gradient sizes [24].

For accuracy-oriented hyper-parameters, users change (i) the schedule of batch sizes to improve convergence [49], (ii) the schedule of learning rates to improve model accuracy [8], (iii) the length of a warming-up phase to avoid collapsing convergence [20], (iv) the dropout probability to overcome over-fitting [7], and (v) the momentum to improve the robustness of optimisers [27, 56].

2.2 Managing hyper-parameters is hard

When managing the large universe of hyper-parameters, users often find it challenging for two reasons:

(1) Resolving trade-offs within hyper-parameters. There are hyper-parameters that affect both system performance and model accuracy. Figure 1a shows two of the most prominent ones: batch size and the floating-point precision of weights (and their gradients). These hyper-parameters introduce a trade-off when users want to achieve both high system throughput and high test accuracy.

Figure 1b shows this trade-off for batch size. In parallel training, increasing the batch size K-fold, increases system throughput linearly with the number of processors K because the work per processor remains constant. Beyond a certain threshold though, increasing the batch size adversely affects model accuracy: the number of epochs required to

converge to a given accuracy increases super-linearly because there are fewer model updates, and not enough variance to explore the solution space [26, 34]. A similar trade-off exists when selecting different floating-point precisions. A user must manually find the best compromise in a given training scenario.

(2) Adaptive tuning of hyper-parameters. Effective hyper-parameter tuning typically requires adaptive schedules that change over time: (i) a DL model has a complex non-convex space [26]. As training progresses, the model must adjust hyper-parameters to fit itself into the loss space, e.g. using a descending learning rate to capture sharp minima [23], thus improving accuracy; and (ii) from a performance point-of-view, a DL system must adapt to maximise the utilisation of expensive GPU resources (e.g. the latest generation of a GPU server costs $34 per hour in commodity clouds [4]). As cloud providers offer less expensive *preemptive* GPU resources (e.g. at a 70% discount compared to reserved instances [5]), GPU resources may be reclaimed during training with short notice (usually tens of seconds), and the training schedule must react to this.

Existing DL systems allow users to adopt pre-defined schedules for hyper-parameters [1, 44, 48] hard-coded in a driver program. This has two shortcomings: (i) a schedule that has been shown effective for one DL model often performs poorly for others [35]. Hyper-parameter schedules rely on assumptions about the properties and training conditions of particular models [3, 2]. These assumptions rarely hold across models (e.g. the training of a ResNet family is fundamentally different from those for generative adversarial networks [19]); and (ii) a hyper-parameter schedule concluded from a small-scale experiment may not be effective at large scale. Training at different scales often exhibit different convergence behaviours as the batch size [20] and data pre-processing can both vary [36].

3. Hyper-parameter-friendly DL systems

We now describe designs for DL systems that help address the above challenges related to hyper-parameters.

3.1 Avoiding performance-oriented hyper-parameters

Our idea for removing performance-oriented hyper-parameters is based on the observation that new DL system designs should leverage new GPU hardware features for concurrency combined with alternative approaches for data parallelism. We illustrate this by explaining how a DL system can relax the tension that the choice of batch size introduces between system performance and model accuracy.

The trade-off with batch size. As existing DL system with parallel S-SGD scale to more GPUs, users must choose larger batch size. Since a batch is sharded across all GPUs, this keeps the amount of work per GPU constant. Large batch sizes, however, come at the cost of lower statistical efficiency [26, 34], which leads to slower convergence of the model to a high accuracy. Users try to compensate for this by tuning hyper-parameters such as increasing the learning rate [20], or adjusting the batch size [49]. These techniques are model-specific though and eventually fail to be effective for very large batch sizes [13, 24, 20]. Given these issues, users prefer to use small batches when possible [30, 34].

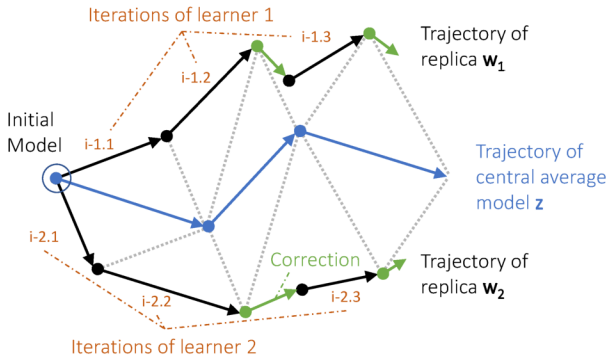
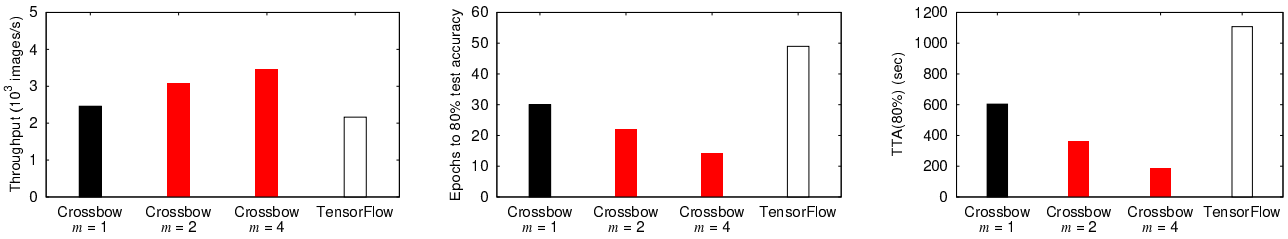


Figure 2: Optimisation trajectories with independent model replicas under model averaging (Two model replicas are trained independently with fixed batch sizes and corrected by an average model per iteration.)

Decoupling performance and batch size. We want to design a DL system that decouples data processing throughput from the batch size of the learning algorithm. This means that we must exploit the full parallelism of all GPUs independently of the chosen batch size. Besides data parallelism, modern GPUs can also execute multiple gradient computations independently to increase processing throughput via the use of separate execution queues. GPUs also have advanced primitives (e.g. NVLink [38] and NCCL [37]) for inter-GPU synchronisation.

Instead of increasing the batch size, a DL system can therefore train multiple model replicas with a small batch size on one GPU and efficiently consolidate their training state. A replica can process a data batch independently, compute a gradient, immediately update itself, and continue with the gradient computation for the next batch. Training many replicas on GPUs allows a DL system to increase data processing throughput while producing independent model updates throughout training, which avoids having to increase the batch size linearly with more GPUs .

A challenge that this introduces is that a large number of indenpendent model replicas is harder to synchronise efficiently. Instead of using S-SGD, which uses a global model to *replace* local models on GPUs per iteration, we exploit *model averaging* techniques [42, 41, 47], which *correct* diverged model replicas with a central average model. Figure 2 shows how model replicas interact with the average model. Starting with the same initial model, two learners train model replicas w_1 and w_2 using SGD with distinct batches. When the learners have computed the gradients and updated their local replicas, their updates are applied to a



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

Figure 3: Training performance of Crossbow and TensorFlow (Crossbow uses different numbers of replicas m.)

central average model. The average model is used to compute *corrections* for each replica. This ensures that they follow the trajectory of the central average model, while still maintaining their independence.

Crossbow implementation. To explore the feasibility of efficient small batch training with independent model replicas and model averaging, we have a prototype implementation of a multi-GPU DL system called CROSSBOW [28]. When the user sets the bach size hyper-parameter to a small value, CROSSBOW automatically computes the best number of parallel replicas to fully utilise all GPU resources. It then schedules the replicas to be trained using the given batch size. To maximise the number of independent replicas, CROSSBOW optimises replicas for data locality and object reuse. It also prevents synchronisation bottlenecks: the central average model is replicated across GPUs, enabling the system to efficiently coordinate a large number of replicas.

Experimental results. We evaluate if CROSSBOW can indeed offer the merits of small batch sizes while increasing hardware utilisation. We use CROSSBOW to train the ResNet-32 model [22] with a batch size of 64 (found to be the best after exploration) on a NVIDIA Titan X GPU, and compare to TensorFlow [1].

We show the results in Figure 3. With 4 model replicas (Figure 3a), CROSSBOW increases the throughput by a factor of 1.4× compared to one replica. By adding replicas instead of increasing the batch size, we also observe an improvement in statistical efficiency, as shown in Figure 3b: the number of epochs required to converge reduces from 30 to 14. The reason is that the independent replicas can synchronise efficiently using model averaging. Figure 3c shows that the combined improvements in hardware and statistical efficiency reduce the overall training time (i.e. time to an 80% accuracy) by 3.2×.

3.2 Supporting dynamic hyper-parameter policies

Our idea for supporting dynamic policies for hyper-parameter tuning is to embed monitoring and control functionality as part of a distributed DL system. We demonstrate the benefit of dynamic hyper-parameter policies through a use case in which a DL system observes the gradient noise and uses it to adapt the batch size during the training process.

Dynamic hyper-parameter policies. To achieve high training accuracy, a DL system must adapt hyper-parameters over time to fit the search for minima in a complex loss space [49, 35, 16]. This could be done by monitoring metrics that reflect the current training progress [33, 35, 53], and then tuning different hyper-parameters [8, 55].

As an example, we consider the case in which a user wants to adapt the batch size based on the *gradient noise scale* (GNS) [35]. GNS is a statistical measure for the signal-to-noise ratio of gradients, and expresses the variation in the data as seen by the model. Intuitively, when the noise is small, using large batches of training data is counterproductive; conversely, when GNS is large, the model learns better with more data per batch.

Dynamic hyper-parameter policies require the scalable monitoring of training metrics such as GNS. Existing DL systems resort to external monitoring functionality as provided by TensorBoard [51] or Prometheus [43]. Since such monitoring tools collect logs offline, the metrics become only available after training, forcing users to derive static hyper-parameter policies from them.

```
1  import training as tr
2  import monitoring as mon
3  import communication as comm
4  import control as ctrl
5
6  def adapt_batch_size(ctrl, noise):
7    g_batch_size = tr.exp_decay(noise, 0.01)
8    ctrl.global_batch_size.sync(g_batch_size)
9
10 def build_driver_program(sample, loss):
11   grads = tr.resnet(sample, loss).auto_diff()
12   avg_grads = comm.all_reduce(grads)
13   optimiser = tr.optimiser(avg_grads)
14   noise = mon.noise(grads, avg_grads)
15   avg_noise = comm.all_reduce(noise)
16   c = ctrl.control(optimiser, avg_noise)
17   c.hook(adapt_batch_size, avg_noise)
```

Listing 1: A dynamic hyper-parameter policy

Expressive policies. Adaptive hyper-parameter tuning requires a high-level abstraction to define dynamic policies. Such policies may be expressed as part of a driver program, and require ideally minimal efforts for implementing the monitoring and control programs. In addition, the abstraction must support computation over monitored metrics and

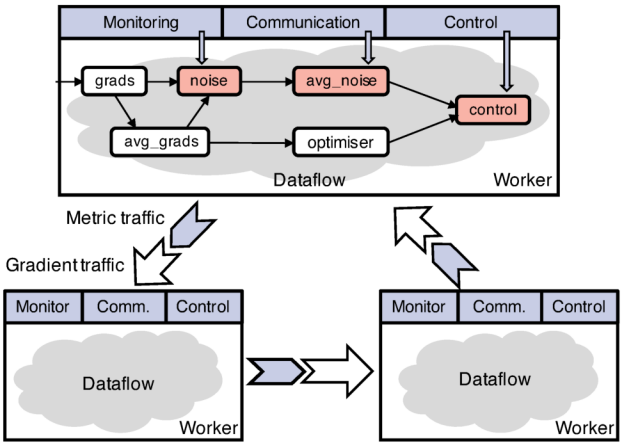


Figure 4: Dynamic hyper-parameter management with three workers (Each replicated training dataflow is augmented with monitoring and control operators. The monitored metrics are piggybacked as part of the communication for distributed gradient synchronisation.)

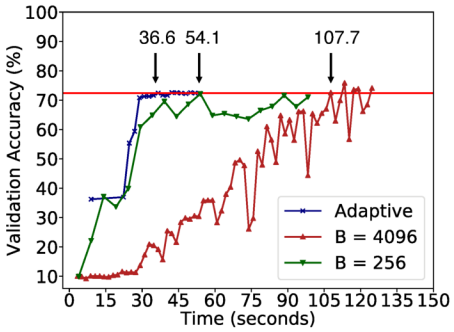


Figure 5: Time-to-accuracy of adaptive and static batch size policies in KUNGFU (B denotes batch size.)

synchronisation of the changes to hyper-parameters in a distributed environment.

Based on these requirements, we propose the abstraction shown in Listing 1: (i) a driver program contains monitoring operators that directly attach to the training operators (line 13 attaches the gradient and average-gradient operators to a monitoring operator that computes the GNS); (ii) it includes collective communication operators that help compute global metrics (line 14 computes the average noise scale using an all-reduce operator); and (iii) it has control operators that can continuously evaluate monitoring metrics and synchronise modification to hyper-parameters (line 15-16 register an adaptation function called every iteration of training, and line 7-8 transform the noise scale to a new global batch size which is automatically synchronised on distributed driver programs).

Embedding policies in dataflows. We want to design a DL system that can efficiently realise the above abstraction. This can be achieved by transforming the above driver program into a dataflow that is replicated by the workers.

Figure 4 uses the GNS use case to show the design. Each worker has a DL library that performs training using dataflows. It creates a training dataflow augmented with monitoring, communication and control operators, as defined in Listing 1. These operators are asynchronously executed and reuse the result produced by the training operators (i.e. grads and avg_grads), thus avoiding the interruption of training and the copying of data. In addition, the communication operators can use a networking layer that piggybacks the communication of monitoring data with synchronised gradients. This not only improves networking performance, but also reuses scale-out mechanisms [24] in both training and monitoring. Finally, the replicated control operators can be treated as an execution barrier. This provides clear semantics to users regarding *when* and *where* the changes to hyper-parameters take place in a distributed setting.

KungFu implementation. We have implemented the above design for dynamic hyper-parameter tuning as part of a distributed training library called KUNGFU for TensorFlow [1]. The computation of GNS and the associated batch size controller are implemented as TensorFlow operators. To modify batch sizes dynamically, KUNGFU fixes the batch size per dataflow, guaranteeing high GPU utilisation, and implements a decentralised runtime that allows TensorFlow nodes to join and leave with low overhead. The KUNGFU runtime also features an efficient collective communication layer so that the additional distributed measurement of metrics does not lead to bottlenecks.

Experimental results. We implement the sample policy in Listing 1 using KUNGFU. We evaluate the effectiveness of this policy when training the ResNet-32 model [22] with the CIFAR-10 dataset [29], as implemented in the TensorFlow benchmarks project [52]. The experiment runs on a 20-CPU-core server with 4 NVIDIA Titan X GPUs. We compare this dynamic policy with two static policies that use small and large batch sizes (256 and 4096, respectively). We fix the learning rate to 0.1 and compare their time to reach the shared maximum test accuracy (in our setting, 72%).

Figure 5 shows the time-to-accuracy results. The small-batch policy benefits from a high statistical efficiency and thus converges 50% faster than the large-batch policy, even though its GPU utilisation is lower. The adaptive policy starts with a small batch size of 128 and gradually reaches 4096 after 4 epochs, helping it obtain the merits of both small and large-batch training. Hence, it achieves the best time-to-accuracy, which is 32% faster than the static small-batch policy. This also demonstrates the low performance overheads of the KUNGFU approach.

We also evaluate the robustness of convergence for these three policies, and let them train for 9 extra epochs after reaching the target accuracy. The adaptive policy converges

to a more stable accuracy compared to the others. This implies that the use of online monitoring metrics helps select hyper-parameters that can best fit in the loss space, thus improving the quality of minima.

4. Conclusions

Today's DL systems face substantial challenges related to the configuration of hyper-parameters. In this paper, we have described how the designs of future DL systems can help users tame hyper-parameters. We have showed that it is possible to rethink designs (i) to remove the impact of critical hyper-parameter, such as the batch size, on both performance and accuracy; and (ii) to support the efficient monitoring of training metrics, thus enabling a range of dynamic policies for hyper-parameters to be implemented efficiently.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (2016), pp. 265–283.

[2] ALLEN-ZHU, Z., AND LI, Y. Can SGD learn recurrent neural networks with provable generalization? *CoRR abs/1902.01028* (2019).

[3] ALLEN-ZHU, Z., LI, Y., AND LIANG, Y. Learning and generalization in overparameterized neural networks, going beyond two layers. *CoRR abs/1811.04918* (2018).

[4] AMAZON. Amazon EC2 P3 Instance Product Details. `https://aws.amazon.com/ec2/instance-types/p3/`, 2019. Online; accessed: 2019-05-17.

[5] AMAZON. Amazon Spot Instance Prices. `https://aws.amazon.com/ec2/spot/pricing/`, 2019. Online; accessed: 2019-05-17.

[6] ARIK, S. Ö., CHRZANOWSKI, M., COATES, A., DIAMOS, G., GIBIANSKY, A., KANG, Y., LI, X., MILLER, J., RAIMAN, J., SENGUPTA, S., AND SHOEYBI, M. Deep voice: Real-time neural text-to-speech. *CoRR abs/1702.07825* (2017).

[7] BA, L. J., AND FREY, B. Adaptive dropout for training deep neural networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (USA, 2013), NIPS'13, Curran Associates Inc., pp. 3084–3092.

[8] BAYDIN, A. G., CORNISH, R., MARTÍNEZ-RUBIO, D., SCHMIDT, M., AND WOOD, F. D. Online learning rate adaptation with hypergradient descent. *CoRR abs/1703.04782* (2017).

[9] BOTTOU, L. On-line learning and stochastic approximations. In *On-line Learning in Neural Networks*, D. Saad, Ed. 1998.

[10] BOTTOU, L., CURTIS, F., AND NOCEDAL, J. Optimization methods for large-scale machine learning. *SIAM Review 60*, 2 (2018), 223–311.

[11] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR abs/1512.01274* (2015).

[12] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., AURELIO RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., AND NG, A. Y. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[13] DEAN, J., PATTERSON, D., AND YOUNG, C. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro 38*, 2 (Mar 2018), 21–29.

[14] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[15] DEVLIN, J., CHANG, M., LEE, K., AND TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR abs/1810.04805* (2018).

[16] DUCHI, J., HAZAN, E., AND SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, Jul (2011), 2121–2159.

[17] ELSKEN, T., METZEN, J. H., AND HUTTER, F. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377* (2018).

[18] FEURER, M., KLEIN, A., EGGENSPERGER, K., SPRINGENBERG, J. T., BLUM, M., AND HUTTER, F. Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2015), NIPS'15, MIT Press, pp. 2755–2763.

[19] GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[20] GOYAL, P., DOLLÁR, P., GIRSHICK, R. B., NOORDHUIS, P., WESOLOWSKI, L., KYROLA, A., TULLOCH, A., JIA, Y., AND HE, K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR abs/1706.02677* (2017).

[21] GUPTA, S., AGRAWAL, A., GOPALAKRISHNAN, K., AND NARAYANAN, P. Deep learning with limited numerical precision. In *International Conference on Machine Learning* (2015), pp. 1737–1746.

[22] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[23] HOFFER, E., HUBARA, I., AND SOUDRY, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett,

Eds. Curran Associates, Inc., 2017, pp. 1731–1741.

[24] JIA, X., SONG, S., HE, W., WANG, Y., RONG, H., ZHOU, F., XIE, L., GUO, Z., YANG, Y., YU, L., CHEN, T., HU, G., SHI, S., AND CHU, X. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *CoRR abs/1807.11205* (2018).

[25] JIN, H., SONG, Q., AND HU, X. Efficient neural architecture search with network morphism. *CoRR abs/1806.10282* (2018).

[26] KESKAR, N. S., MUDIGERE, D., NOCEDAL, J., SMELYANSKIY, M., AND TANG, P. T. P. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR abs/1609.04836* (2016).

[27] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[28] KOLIOUSIS, A., WATCHARAPICHAT, P., WEIDLICH, M., MAI, L., COSTA, P., AND PIETZUCH, P. R. CROSSBOW: scaling deep learning with small batch sizes on multi-gpu servers. *CoRR abs/1901.02244* (2019).

[29] KRIZHEVSKY, A. Convolutional deep belief networks on cifar-10, 2010.

[30] LECUN, Y. A., BOTTOU, L., ORR, G. B., AND MÜLLER, K.-R. *Efficient BackProp*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 9–48.

[31] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (2014), pp. 583–598.

[32] LIN, Y., HAN, S., MAO, H., WANG, Y., AND DALLY, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *CoRR abs/1712.01887* (2017).

[33] MARTENS, J., AND GROSSE, R. B. Optimizing neural networks with kronecker-factored approximate curvature. *CoRR abs/1503.05671* (2015).

[34] MASTERS, D., AND LUSCHI, C. Revisiting small batch training for deep neural networks. *CoRR abs/1804.07612* (2018).

[35] MCCANDLISH, S., KAPLAN, J., AMODEI, D., AND TEAM, O. D. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162* (2018).

[36] MENG, Q., CHEN, W., WANG, Y., MA, Z.-M., AND LIU, T.-Y. Convergence analysis of distributed stochastic gradient descent with shuffling. *arXiv preprint arXiv:1709.10432* (2017).

[37] NVIDIA COLLECTIVE COMMUNICATIONS LIBRARY (NCCL), 2018. `https://developer.nvidia.com/nccl`.

[38] NVLINK FABRIC, 2018. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[39] PAVLO, A., ANGULO, G., ARULRAJ, J., LIN, H., LIN, J., MA, L., MENON, P., MOWRY, T. C., PERRON, M., QUAH, I., ET AL. Self-driving database management systems. In *CIDR* (2017), vol. 4, p. 1.

[40] POLYAK, B. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics 4* (12 1964), 1–17.

[41] POLYAK, B. New stochastic approximation type procedures. *Avtomatica i Telemekhanika 7*, 7 (01 1990), 98–107.

[42] POLYAK, B., AND JUDITSKY, A. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization 30*, 4 (1992), 838–855.

[43] PROMETHEUS. The Prometheus monitoring system and time series database. `https://github.com/prometheus/prometheus`, 2019. Online; accessed: 2019-05-18.

[44] PYTORCH, 2018. `https://pytorch.org`.

[45] RAJPURKAR, P., ZHANG, J., LOPYREV, K., AND LIANG, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[46] ROBBINS, H., AND MONRO, S. A stochastic approximation method. *Ann. Math. Statist. 22*, 3 (09 1951), 400–407.

[47] RUPPERT, D. Efficient estimators from a slowly convergent Robbins-Monro process. Tech. Rep. 781, School of Operations Research and Industrial Enginnering, Cornell University, Ithaka, New York 14853-7501, February 1988.

[48] SEIDE, F., AND AGARWAL, A. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016* (2016), B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds., ACM, p. 2135.

[49] SMITH, S. L., KINDERMANS, P., AND LE, Q. V. Don't decay the learning rate, increase the batch size. *CoRR abs/1711.00489* (2017).

[50] SNOEK, J., LAROCHELLE, H., AND ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (USA, 2012), NIPS'12, Curran Associates Inc., pp. 2951–2959.

[51] TENSORFLOW. TensorFlow's Visualization Toolkit. `https://github.com/tensorflow/tensorboard`, 2019. Online; accessed: 2019-05-18.

[52] TENSORFLOW BENCHMARKS, 2019. `https://github.com/tensorflow/benchmarks`.

[53] TSUZUKU, Y., IMACHI, H., AND AKIBA, T. Variance-based gradient compression for efficient distributed deep learning. *CoRR abs/1802.06058* (2018).

[54] YOU, Y., LI, J., HSEU, J., SONG, X., DEMMEL, J., AND HSIEH, C. Reducing BERT pre-training time from 3 days to 76 minutes. *CoRR abs/1904.00962* (2019).

[55] ZHANG, J., AND MITLIAGKAS, I. Scaling SGD batch size to 32K for ImageNet training. *CoRR abs/1708.03888* (2017).

[56] ZHANG, J., AND MITLIAGKAS, I. YellowFin and the art of momentum tuning. *CoRR abs/1706.03471* (2017).