Expedite Neural Network Training via Software Techniques

Fangzhe Chang
Nokia Bell Labs, USA
fangzhe.chang@nokia-bell-labs.com

Dong Liu
Nokia Bell Labs, USA
d.liu@nokia-bell-labs.com

Thomas Williams
Nokia Bell Labs, USA
thomas.williams@nokia-bell-labs.com

Thomas Woo
Nokia Bell Labs, USA
thomas.woo@nokia-bell-labs.com

ABSTRACT

Training of Deep Neural Networks (DNN) can take a long time and correspondingly incurs a fair amount of cost, especially as the DNN models grow in complexity and training datasets become ever larger. The de facto approach to reducing training time is to split the data minibatches across multiple hardware accelerators (GPUs and TPUs) and train the model in a distributed fashion. Recent advances have allowed the use of hundreds to thousands of accelerators in training to radically reduce training time for some models. This practical paper studies software techniques for expediting neural network training without requiring a large number of expensive hardware accelerators. These techniques include: (1) *local accumulation training* to decrease the number of synchronization points; (2) *heterogeneous distributed training* where faster workers do not have to wait an unnecessarily long time for the slower workers at synchronization points; (3) *online cooling until early stop* where batch size and learning rate are adjusted to gradually cool down training until an early stop point. These techniques can be applied in isolation or combined together. Our experiments show that integrated application of these techniques can reduce the overall training time by 30%.

CCS CONCEPTS

• **Computing methodologies** → **Distributed artificial intelligence**; **Machine learning**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Machine Learning, Neural Network Training

1 INTRODUCTION

Deep Neural Network (DNN) models can learn from (i.e., be trained with) expert knowledge, usually in the form of labeled data. Since they have the potential to adapt to implicit data patterns and changing conditions in real-world problems, DNN models are becoming increasingly popular, spreading from the traditional areas of vision and natural language processing to a wider range including voice recognition and interaction, machine translation, self-driving cars, and even conventional software functions such as indexing [14]. As the DNN models grow in complexity and datasets become ever larger [2, 9], training can take a very long time and correspondingly incur a fair amount of cost.

Long training time has become the key hurdle for productivity of deep learning development. Just like instantaneous compilation has transformed traditional program development, we envision that rapid training of DNN models can substantially transform deep learning development and its adoption. At this point, the use of DNN and in general machine learning has rapidly moved beyond early exploration and reached the application phase for many organizations. In this phase, it is important to start developing a systematic or a cookbook approach to expedite DNN training, thus making DNN development accessible to early adopters.

Recent advances have allowed the use of hundreds to thousands of accelerators in training to radically reduce training time for some models. For instance, the ResNet-50 model was originally trained over the ImageNet1K [21] dataset on a single machine (with 8 P100 GPUs) for 29 hours [8], which was reduced to an hour [7] on 32 machines (each with 8 P100 GPUs, a total of 256 P100 GPUs) connected by a 50Gb Ethernet, and to half an hour [23] on half a TPU pod (256 tensorcores), and most recently to just a few minutes [12] on 256 machines (each with 8 P40 GPUs, a total of 2,048 P40 GPUs) over a 100Gb Ethernet. Noticeably, most of the gains are made due to progress in utilizing a large number of hardware resources.

Despite these results, there are still significant practical hurdles.

(1) In reality, access to a large number of homogeneous GPUs is expensive and may not always be available. Other than a few web-scale companies, most organizations only have a handful of multi-GPU servers. Many heroes experiments conducted on training time reduction used large dedicated GPU farms, the results of which may not apply directly on more modest on-premise clusters or cloud-based clusters.
(2) GPUs are undergoing rapid innovation. For instance, Nvidia has released three major GPU architectures (Pascal, Volta, Turing) in just over two years. As an enterprise acquires GPU resources over multiple purchase cycles, their environment is bound to have a mixture of newer and older GPUs and machines. All training results reported above assume the use of homogeneous GPUs. In real life, it is critical to be able to use any GPU mix for training.
(3) There have been many papers on different techniques in optimizing training, covering different areas: hyperparameter search, freezing, etc. Most are focused on exploring a single technique. There is a lack of study of how different techniques can be used in combination effectively. More specifically, rather than individual techniques, the key is to understand an overall training schedule. More specifically, what techniques to apply at what time as training progresses. It is more useful to develop a cookbook approach integrating state-of-the-art techniques to create an effective training schedule, especially one that can monitor training progress and dynamically adjust training hyperparameters.

This practical paper studies software techniques to expedite neural networks training without requiring a large number of homogeneous GPUs. Our objective is to significantly reduce training time while still maintaining the quality (e.g., accuracy, bleu, etc.) of the training. It has been reported [11] that around 75% of jobs reached within 0.1% of their best model with just 40% of training epochs. This points out the likelihood to reduce a large portion of training time through controlling training schedules. In particular, we focus on the following software techniques and demonstrate how they help reduce training time.

Local accumulation training where gradients from multiple batches are accumulated before sending for synchronization. Local accumulation training essentially forms large batch sizes beyond what can fit in the memory of an individual GPU/TPU. Consequently it reduces the number of synchronization points for each epoch. It also statistically reduces idle time caused by the variance in compute time on participating devices due to different sample sizes (e.g., lengths of sentences).

Heterogeneous distributed training where faster workers do not have to wait unnecessarily long time for the slower workers at the synchronization points, using proportional batching and local accumulations.

Monitoring and cooling until early stop where the batch sizes and learning rates are adjusted according to monitored metrics (e.g., training accuracy and test accuracy). It gradually cools down the magnitude of updates, until the updates no longer help improve the metric as expected.

Our techniques are specifically designed to address the practical issues raised earlier about DNN training, and we study them in combination in a realistic training environment.

The key contributions of this paper are as follows.

(1) We experiment with using local accumulation as a method to reduce training time by allowing large batch sizes without being limited by the memory capacity of the GPUs.
(2) We use proportional batching and local accumulation for maximizing the training throughput when training on a heterogeneous set of GPUs and machines.
(3) We generalize the plateau detection algorithm over multiple quality metrics for online adjustment of batch size as well as learning rate, and for early stop decision.
(4) We study the performance of these techniques experimentally across multiple models. We demonstrated effectiveness and compatibility of these techniques when used together. Our experiences show that integrated use of the above techniques can reduce the overall training time by 30%.

The paper is organized as follows. Section 2 discusses existing work on expediting DNN training. Section 3 describes in detail the software techniques we investigate in this paper. The experimental results are reported in Section 4. Section 5 summarizes the paper.

2 RELATED WORK

It is intriguing to reduce the training time without sacrificing the quality of the trained model. Many efforts have aimed at speeding up DNN training. The most widely adopted approach is to divide the training up among many hardware accelerators (GPUs or TPUs) that can process the training using data parallel synchronous SGD. To be able to exploit a large number of GPUs, one needs a large batch size. However, the generalization accuracy of the resulting model has suffered when a large batch was used in training [13]. Recent advances have demonstrated a number of techniques such as warm-up [7] and LARS [26] to overcome this limitation, though some of these techniques require extra support from the underlying framework or modification to the original application. Combinations of those techniques have allowed models like ResNet-50 to be trained with batch sizes 8K [7], 16K [23], 32K [1, 5, 26], and 64K [12], using hundreds to thousands of GPUs. In comparison, this paper focuses on software techniques (e.g., local accumulation training) for expediting training even without large number of hardware resources.

In general the amount of computation needed for training using SGD is fixed. Although there are known methods for reducing the training computation by early stop [19]. It has been observed [11] that in practice around 75% of trainings only need 40% of epochs to get a model within 0.1% of their best. Early stopping will greatly save the amount of computation for such jobs. The 'Early Stopping, but When?" paper [19] proposed three classes of stopping criteria. More complex algorithms were also introduced, e.g., in [6, 15], which will incur significant overhead if used online.

When spreading training among multiple machines, heterogeneity in hardware capabilities may cause idle time at the synchronization point. Use of local accumulation to reduce the variance in workload between different workers has been independently proposed in [18]. In addition, we also investigate using proportional batch sizes for bridging the gaps among distributed workers.

Freezing [3, 4, 20] have been shown to reduce training time and prevent overfitting when the layers are frozen at the right time. As reported in [20], neural networks broadly converge from the bottom up. One can successively freeze lower layers during training, avoiding all computation needed for deriving gradients of and for updating the weights of the frozen layers. On the other side, freezing a layer prevents adjusting its parameters to more suitable values, potentially leading to models of degraded quality. We experimented extensively with this technique and found out the time saving often came with a non-negligible degradation on training quality. We realized that the subtle changes to the bottom layers even at a late stage could be significant to some model.

We have not investigated all the software techniques that can potentially reduce training time. Some of them are well accepted practices, including 1) use of half precision or mixed precision [17] training, which allows more parallel computation inside GPUs and expands the number of samples that fit into GPU memory; 2) use of efficient communication primitives during updates, such as ALL_REDUCE and its various implementations, compression or quantization of gradients; 3) optimizing data pipeline to overlap data fetching and computation, etc.

3 SOFTWARE TECHNIQUES FOR REDUCING TRAINING TIME

We focuses on the following software techniques for reducing training time: local accumulation, proportional batch for heterogeneous training, and online cooling schedule until early stop.

3.1 Local Accumulation Training

When training with Stochastic Gradient Decent (SGD), at each iteration a subset (i.e., mini-batch) of samples are processed, the corresponding average gradients are scaled by a learning rate, and the resulting updates are applied on model parameters. In implementation, the subset is further split among participating GPUs. After each GPU finishes processing its share, the gradients from all workers are gathered together, essentially forming a synchronization point. In *local accumulation* training, a worker accumulates the gradients for a number (say k) of iterations before synchronizing with other participants. As a result, the mini-batch size is conceptually increased by k fold and the number of synchronization points reduced by a factor of k in an epoch. Fewer synchronization means not only shorter waiting time for each other, but also smaller communication overhead for transferring gradients and parameter values. The training time will be accordingly reduced.

Local accumulation can be viewed as a software technique that allows increasing the effective batch size beyond the hardware limits of the number of available GPUs and capacity of GPU memory. Accordingly it suffers from the same drawbacks as large batch training: the reduction in generalization accuracy in the trained model [7, 13]. Similar techniques, such as gradual warm-up can be attempted to alleviate the problem.

When the batch size is increased via local accumulation, learning rate may require a corresponding change. Different rules have been suggested. A linear increase of the learning rate with respect to batch size was proposed in [16] “to keep the mean SGD weight update per training example constant”. A square-root rule was instead suggested in [10] to “make sure that the covariance matrix stays the same for all mini-batch sizes”. Nevertheless, it has been observed [22] that neither the linear rule nor the square-root rule may be appropriate and suggested to search for a proper learning rate for each different batch size instead.

Figure 1 and 2 illustrate the practical difficulty of relating varying of batch sizes to that of learning rates, using the training ResNet-20 v2 on CIFAR 100 as example. Both figures contain a common data point, corresponding to the hyperparameter setting of batch size 1024 and learning rate 1. Either batch size in Figure 1 or learning rate in Figure 2 is doubled (or halved) for consecutive data points. As one can see, these two curves do not exactly match in shape. In particular, changing learning rate is more sensitive to causing divergence than changing batch sizes. Doubling batch size is not always equivalent to halving learning rate as proposed by [23].

Figure 3 normalizes the above two curves with respect to the linear scaling rule (see the solid ‘BS LINEAR’ and ‘LR LINEAR’ curves) and the square-root scaling rule (see the dotted ‘BS SQRT’ and ‘LR SQRT’ curves). As one can see, there are ranges for each rule to be applicable. The linear rule applies when the two solid curves overlap well (for the per-sample learning rate between 0.00025 and 0.002). The square-root rule works well when the dotted curves (i.e., the lr/sqrt(bs) ratio) are between 0.016 and 0.3. This means that the linear rule and square-root rule may only work for parts of the hyper-parameter space, even when the hyper-parameters don’t change during the training. In practice, learning rates often cool down during trainings, making it harder to choose a proper learning rate schedule. We look into this issue in Section 3.3.

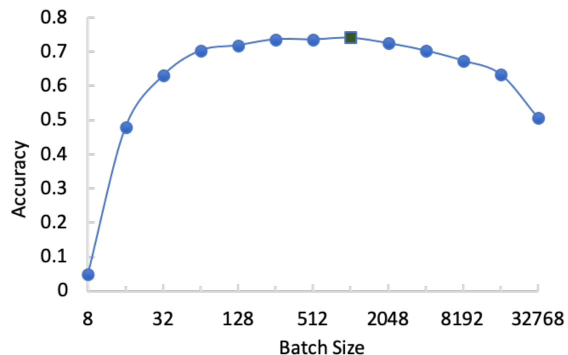

Figure 1: Double batch size when training on CIFAR 100

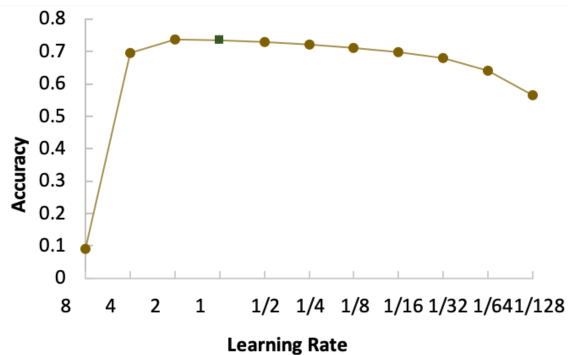

Figure 2: Halve learning rate when training on CIFAR 100

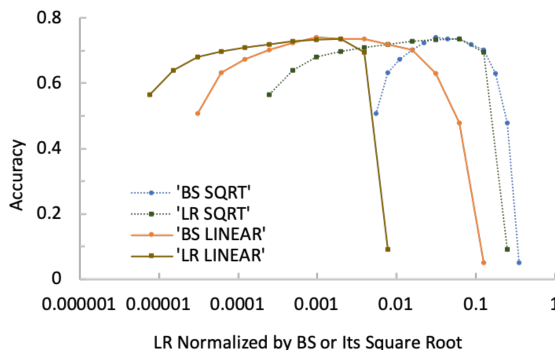

Figure 3: Linear rule and square-root rule are suitable for specific regions

3.2 Heterogeneous Distributed Synchronous Training

When training a model using data parallel synchronous SGD, all participating workers synchronize after processing a minibatch of samples to ensure gradients are collected and applied to parameters before obtaining the new weights and proceeding with the next iteration. Current practices of distributed synchronous training use similar types of machines and GPUs, e.g., as in [7, 23]. In such settings, all workers get an equal share of the minibatch and complete the computation around the same time at each iteration. Therefore,

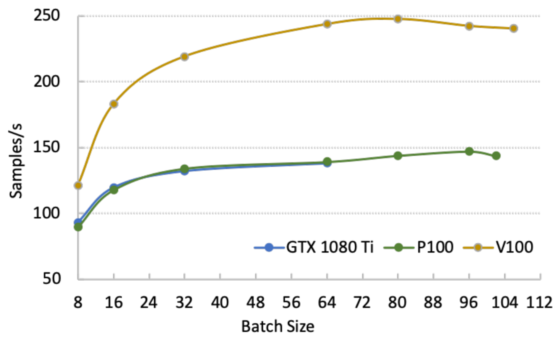

Figure 4: When training a TensorFlow Inception model on ImageNet, heterogeneous GPUs (GTX 1080 Ti, Tesla P100, and Tesla V100) differ in throughput as batch size varies.

a worker tends to wait only very little time for other workers to be ready for synchronization. However, as newer generations of GPUs are introduced to market while the old ones are still in use, it becomes ever more likely that the GPUs available in a cluster are heterogeneous. They differ in the number of cores, tensorcores, amount of memory, the presence of NVLink, etc. As shown in Figure 4, three types of GPUs can accommodate a minibatch of different sizes and yield different throughputs for a same model.

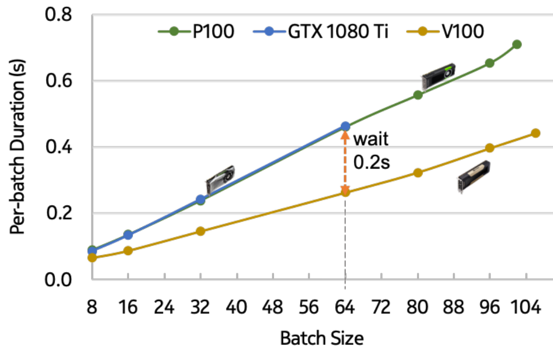

Figure 5: When using equal batch sizes in the distributed synchronous training of MxNet Inception model, V100 GPUs finishes processing the batches sooner (e.g., by 0.2s for batch size 64) than P100 and 1080 Ti GPUs.

Existing frameworks such as TensorFlow and MxNet are agnostic to the different GPU capabilities and allocate equal share of a minibatch to them when the heterogeneous GPUs are used in a distributed synchronous training. As a result, at the end of each iteration a much faster GPU must wait for the slower ones to finish their computation. The faster ones cannot proceed before the synchronization completes. As shown in Figure 5, a V100 GPU can process a batch of 64 images in 0.2 second less time than a P100 GPU on average. This consequently leads to the more expensive faster GPUs yielding the same throughput as the old slower GPUs, as shown in Figure 6.

In *heterogeneous distributed synchronous training*, each participating worker is given the proportional amount of work so that on average they complete their computation at around the same time. More specifically, the proportional amount of work can be in two forms below or their combination.

Proportional Batch Each worker (or GPU) uses a micro (i.e., per worker/GPU) batch size proportional to its processing capability.

Proportional Local Accumulation Each worker locally accumulates the gradients a different number of times proportional to its processing power before each update (See Table 7 for an example).

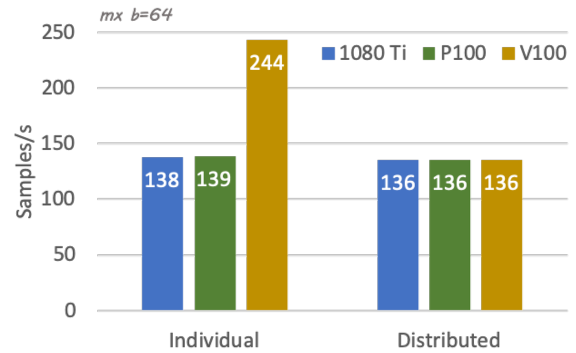

Figure 6: Train MxNet Inception model. Waiting for slower GPUs causes the faster GPU to deliver the same throughput as the slower ones in the distributed training (right).

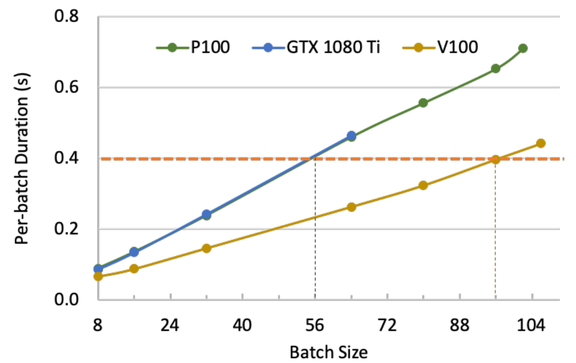

Figure 7: Use batch sizes proportional to the capability of GPUs. More specifically, batch size 96 on a V100 and batch size 56 on a P100 and on a 1080 Ti will make all the workers to finish around the same time per iteration when training the Inception model on ImageNet.

Figure 7 illustrates an approach for deciding the per-GPU batch size in heterogeneous training: A V100 can be allocated 96 samples whereas a P100 or 1080 Ti GPU can be given 56 samples in each iteration so that all of them complete processing in about 0.4 second. The proportional batching approach does not require that all the samples (and their corresponding DNN graph) fit into GPU device memory. When a share of minibatch is too big for the GPU memory, it can be further broken down using multi-pass as in Section 3.1.

As a result, a faster worker not only can load more samples into GPU memory at each iteration, but also can run more iterations before every synchronization so that it does not have to wait for a slower worker.

When the proportional batch approach requires a too large a micro batch size to fit in the memory of its GPUs, one can use the proportional local accumulation or a combination of both to derive a proper setting that minimizes the waiting time at the synchronization points due to GPU heterogeneity.

3.3 Monitoring and Cooling Until Early Stop

In order to reduce the training time while still maintaining quality of the trained model, it is obviously important when to stop the training—after the model has converged additional training will be a waste. Stopping the training at the right moment not only saves training time but also prevents overfitting. From the perspective of simulated annealing, a training session typically consists of an initial optional warm-up phase and a later cool-down phase. In the cool-down stage the magnitude of updates is reduced to ensure progress and convergence. Therefore, deciding when to stop and when to cool down properly is critical to shortening training time.

Cooling can be achieved either by decaying learning rate explicitly or automatically via an optimizer, or by increasing batch size [23]. The advantage of increasing batch size is that it decreases the number of synchronizations during the training, although the extent of its cooling effect is capped by the size of the dataset and very large batch sizes could lead to degraded model quality.

Deciding when to cool down and when to stop can be challenging. The on-plateau approach (e.g., ReduceLROnPlateau in Keras) monitors the progress of a training session and reduces the learning rate once a monitored quantity plateaus. We generalize the on-plateau algorithm to monitor both training and evaluation metrics (e.g., training accuracy and validation accuracy), support multiple actions, and allow flexible control over the plateau conditions. Intuitively, it compares expectation and actual progress made over a time period. When not enough progress is observed, the training should be further cooled down or stop.

More specifically, the TracEO algorithm makes cooling and early stop decisions based on a train metric and a validation metric. It could be used with only one metric or expanded to support more metrics. In practice, a variety of metrics can be metered for progress, including loss, training metrics (e.g., accuracy, perplexity), validation metrics, and test metrics. These metrics change over time, though often not monotonically. Loss and training metrics tend to be smooth, whereas test metric and validation metric can be jumpy. Therefore, the algorithm monitors the progress of the training metric and range of change for the validation metric, and compares them with given expectation. The expectation often decays over time since typically progress and oscillation reduce as training gradually converges. Multiple actions can be triggered when lack of progress in training metric is noticed or when the validation metric is flat. That is,

Increase batch size up to a given upper bound, e.g., 10% of the dataset [23].

Decrease learning rate until a given lower bound.

Stop when the most recent cooling action above fails to bring non-negligible changes, or the target metric is reached, or a number of epochs (steps) is reached.

The challenges for this kind of algorithms lies in their sensitivity to controlling parameters such as size of time window and amount of oscillation to be considered as flat. The TracEO algorithm takes several measures: (1) It further smooths out the training metric using a running average. (2) It takes the average progress of all epochs within the window instead of the maximum improvement usually used in the on-plateau approaches. (3) It calculates flatness of the validation metric elastically. The time window can be stretched longer as long as the average, top, and bottom values have not changed significantly. As a result, decision points will be less sensitive to the exact controlling parameters of the algorithm.

4 EXPERIMENTS

Currently the underlying frameworks such as TensorFlow and MxNet do not yet support local accumulation or heterogeneous batch transparently. We modified the training applications individually to support both features and the online control algorithm where necessary. For local accumulation, gradients need be accumulated and reset locally on each GPU device every given number of steps. MxNet provides partial support of it—accumulating gradients in the 'add' mode. For heterogeneous batch, we need to ensure that all participating workers go through equal number of synchronization points in every epoch despite processing different number of samples. In particular we modified how training samples were partitioned and shuffled among the workers in the underlying framework. However, we believe such functions are generic across neural network models and applications and can be conveniently added into the frameworks in the future.

The experiments in this section used the ResNet model [8] and the Inception model [24] for image classification and the Transformer model [25] for machine translation. The models were taken from TensorFlow benchmark code and from examples included in the MxNet and its gluon-nlp package. We adopted other settings built into the packaged examples and did not change the source code to apply techniques suggested in literature [1, 12, 26] that could improve the quality of the resulting models. When a checkpoint of the model was written at the end of each epoch, the time to write the checkpoint was included in the duration measurements.

4.1 Local Accumulation Training

We studied the effect of local accumulation on speeding up training using the ResNet-50 model and the Transformer model on the TensorFlow and the MxNet frameworks, on single machines and on two machines across network.

We first used local accumulation to train the TensorFlow ResNet-50 model on ImageNet1K [21]. The experiments were run on a server with 8 NVIDIA Tesla V100 GPUs with 16GB of memory each connected by NVLink. We used ALL_REDUCE rather than a parameter server and set the hyperparameters following [7]. For instance, we gave each GPU a batch size 32 as per [7] even though the V100 GPU can process more samples concurrently. Table 1 shows the results as local accumulation varied from 1 to 128 and the corresponding batch size increased from 256 to 32K. In general

Algorithm 1 TracEO Training Algorithm

```
Input: bs, lr, bs_max, lr_max, warm, obsv, epsilon, dir, cool, smooth, stretch, tgt
Training loop:
1:  progress = λ a,b: max(b - a, 0) if dir==‘up’ else max(a - b, 0)
2:  procedure FLAT(e, vals, obsv, epsilon, stretch)   ▷ flatness of vals in window obsv, extensible up to a stretch factor
3:      mn, av, mx = min, avg, max of {vals[i] | i ∈ (e - obsv, e]}   ▷ min, average, max of val metrics over the obsv window
4:      if mx - mn < epsilon then
5:          return True   ▷ height of bounding box is negligible
6:      extend = (mx - mn) / epsilon; w = extend × obsv   ▷ consider the extended window
7:      if extend ≤ stretch and w ≤ e then
8:          mn_e, av_e, mx_e = min, avg, max of {vals[i] | i ∈ (e - w, e - obsv]}   ▷ min, average, max over the extended window
9:          return mx_e ≤ mx + epsilon and mn - epsilon ≤ mn_e and av - epsilon ≤ av_e ≤ av + epsilon
10:             ▷ flat if top/bottom/mean lines have not changed much over extended period despite recent oscillation
11:     return False
12: for e in 0..epochs do
13:     train for 1 epoch with (bs, lr), yielding training metric trn and validation metric val
14:     vals[e] = val; trns[e] = smoothed(trn, smooth)   ▷ exponentially smooth out trn with the factor smooth
15:     if e < warm and lr < lr_max then
16:         warm up lr til lr_max
17:         continue
18:     ng = avg {progress(trns[e - obsv], t) | e - obsv < t ≤ e} < epsilon or FLAT(e, vals, obsv, epsilon, stretch)
19:     if (target tgt is reached) or (ng and sincelast) then
20:         break   ▷ stop if target is reached or no good results has been observed since the last cooling
21:     if ng then
22:         cool down by increasing batch size bs (up to bs_max) and decaying learning rate lr by a combined factor of cool
23:         sincelast = True
24:         optionally decay epsilon and skip execution of lines 19–24 for a few (e.g., obsv/2) epochs after each cool-down
25:     else
26:         sincelast = False
```

larger local accumulation reduced training time, but with a cost on the accuracy of the modes. The top-1 validation accuracy of increasingly larger batches gradually become lower.

Table 1: Train TensorFlow ResNet-50 with local accumulation (LA)

BATCH (LA)	ACCURACY(%)	TIME(H)	SPEEDUP(%)
256 (1)	75.29	18.33	
512 (2)	75.38	17.90	2.33
1K (4)	75.31	17.33	5.44
2K (8)	74.84	17.73	3.24
4K (16)	74.83	16.04	12.47
8K (32)	74.61	16.35	10.79
16K (64)	72.50	15.57	15.02
32K (128)	68.11	15.59	14.93

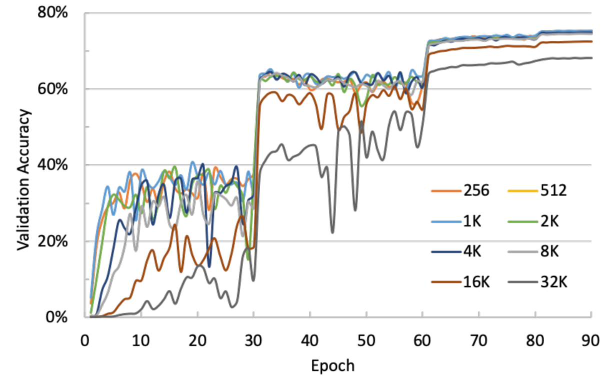

Figure 8: Accuracy per epoch for ResNet-50 (v1) on ImageNet1K.

Figure 8 depicts per-epoch accuracy in details. A batch size 4K (i.e., local accumulation 16) can reduce the training time by about 10% with a sacrifice of less than 0.5% accuracy. The training updates (i.e., synchronizes) 5004 times in each epoch for batch size 256, which is reduced to 312 times for batch size 4K. The decrease in synchronizations per epoch reduces the training time. The synchronization overhead is low when training on the GPUs of a single host and using ALL REDUCE over NVLink high speed interconnect. Therefore, this result should be considered the lowest level speed-up that could be achieved. Involving more GPUs on multiple hosts in the training will magnify the network delays associated with the synchronization and result in larger speed-ups.

We next experimented with varying batch sizes as per [23]: Increasing batch size from 8K to 16K as a way to cool down the

training after 30 epochs. In this experiment, we had to restart the TensorFlow worker and reload the checkpoint to effect the batch size change. The time to restart the worker (about 130 seconds) was included in the duration calculations in Table 2. It shows that large accumulations and large batch sizes can be used as a method of cooling down and at the same time reducing training time. However, one need to weigh the time saving against the loss of quality to obtain a suitable local accumulation number for the training.

Table 2: Train ResNet-50 with varying batch sizes.

BATCH	ACCURACY(%)	TIME(H)	SPEEDUP(%)
8K–16K	74.64	16.25	11.19
8K–32K	74.12	14.95	18.41

We also investigated the effect of local accumulation using a second Transformer model. Table 3 shows the training of the TensorFlow Transformer (big) model on eight V100 GPUs on a single machine over the WMT17 English to German news commentary dataset. It used the Lazy Adam Optimizer (β_1 0.9, β_2 0.997). The learning rate was linearly warmed up in 64000 steps until reaching the factor 4.0, and then reduced by 1/sqrt(steps) at every step. The original implementation stores variables in the IndexedSlices data structure, where each individual variable are updated separately on synchronization. In the process of implementing local accumulation, we copied it into a normal tensor. As a result, the baseline training (i.e., local accumulation 1) was reduced from 26.38 hours to 12.36 hours. Further 9% reduction was achieved using accumulation 8 without much degradation of bleu score. The degradation could be alleviated by adjusting the peak learning rate factor. For instance, we a got better bleu score for the case of local accumulation 2 when using a learning rate factor 2 instead of 4.

Table 3: Train TensorFlow Transformer Model on WMT17 (en2de) dataset.

BATCH (LA)	TIME(H)	BLEU(%)	SPEEDUP(%)
24K	26.38	29.32	ORIGINAL
24K (1)	12.36	29.15	BASELINE
48K (2)	11.49	28.87	7.04
96K (4)	11.27	28.69	8.82
192K (8)	11.14	28.12	9.87
384K (16)	11.10	23.99	10.19
768 (32)	11.59	24.67	6.23

In addition to TensorFlow, we further experimented with using local accumulation on a MxNet model—the Transformer model taken from its gluon-nlp package. The training was over the WMT2014BPE dataset on 8 Volta GPUs and used the hyperparameters as suggested by its documentation. Instead of fixing the accumulation at 16, we doubled it from 1 to 64. We also extended the training from the original 30 epochs to 60 epochs to make sure the best bleu value was obtained. Accordingly the batch size grew bigger linearly with local accumulation, from 21K (i.e., $8 \times 2700 \times 1$) to 1350K (i.e., $8 \times 2700 \times 64$). The results were summarized in Table 4. When accumulation number increased from 1 to 2, training time was surprisingly reduced from 141 hours to 56 hours, mainly due to time saving in the validation and test phase (i.e., not related to local accumulation). The reduction in the training phase (due to local accumulation) was about 11%. Another 10% reduction could be achieved for 4, 8, and 16 local accumulations. After that the training time stayed flat or somewhat incremented. Since we did not fine tune learning rate for the different batch sizes, the hand-picked learning rate for 16 accumulations was too large for the local accumulation 1 scenario. As a result, it only achieved a test bleu of 1.13% (diverged). The bleu score jumped to be more than 25% with 2 passes, reached peaked 26.45% at 8 passes, and gradually degraded after that. This means that a local accumulation in the middle range (e.g., 8 or 16) should be selected with a purpose of both expediting the training and obtaining better models.

Table 4: Train MxNet Transformer Model on wmt2014bpe dataset

PER-GPU BATCH	TIME(H)	BLEU(%)
2700×1	141.93	1.13
2700×2	56.31	25.62
2700×4	50.16	26.21
2700×8	48.58	26.45
2700×16	48.19	26.34
2700×32	49.82	26.29
2700×64	55.20	25.34

Training speed-up by local accumulation should be more visible in a multi-machine distributed environment.

Table 5 shows training TensorFlow ResNet-50 model on two Tesla V100 machines connected by 100G Ethernet and using four GPUs on each machine. The training time reduced as the batch size increased. On the other hand, the accuracy level also decreased noticeably for batch sizes larger than 8k. A sweet spot seemed to be 8 accumulations (i.e., 2k batch size) which enjoyed almost 50% speedup at the cost of a slight reduction on accuracy. Comparing usage of local accumulation on a single machine, it is clear that local accumulation will yield more benefits in a distributed environment where communication and synchronization costs are higher.

Table 5: Train ResNet-50 with local accumulation on two machines.

BATCH (LA)	ACCURACY(%)	TIME(H)	SPEEDUP(%)
256 (1)	75.35	45.40	
512 (2)	75.35	33.93	25.3
1K (4)	75.33	30.30	33.3
2K (8)	75.07	22.91	49.5
4K (16)	74.49	20.36	55.2
8K (32)	73.44	19.48	57.1
16K (64)	72.66	18.12	60.1
32K (128)	69.10	18.27	59.8

Similarly Table 6 shows training of the TensorFlow Transformer (big) model on two machines connected via 100Gbps Ethernet, using four V100 GPUs on each machine. Training time was significantly reduced when the number of local accumulations increased from 1 to 32. One can see a more than 70% saving on time when using at least 8 accumulations, however, with a degradation of bleu scores. This degradation was related to sticking with the same learning rate factor 4. A new search for the hyper-parameters seemed necessary for these larger batches. For instance, the linear scaling approach gave us a bleu score 29.45 for 2 accumulations (i.e., 48k batch size) and score 29.32 for 4 accumulations (i.e., 96k batch size), but a score 0.29 (i.e., diverged) for 16 accumulations. It is still an open question what a best peak learning rate should be adopted when batch sizes are scaled up using local accumulations.

Table 6: Train TensorFlow Transformer with local accumulation on two machines.

BATCH (LA)	TIME(H)	BLEU(%)	SPEEDUP(%)
24K (1)	45.22	29.37	
48K (2)	30.62	28.96	32.29
96K (4)	18.75	28.56	58.54
192K (8)	13.35	27.85	70.48
384K (16)	11.27	26.58	75.08
768K (32)	10.00	24.65	77.89

4.2 Heterogeneous Distributed Training

We first used the Inception model implemented on TensorFlow and MxNet to study the benefit of heterogeneous distributed training. Figure 9 shows the throughput of training TensorFlow Inception model over the ImageNet1K dataset on three heterogeneous GPUs: 1080 Ti, P100, and V100. Even though they could deliver a training throughput of 131, 142, 220 images per second respectively when training a separate model individually, they achieved a throughput of 128 when three of them working together training a shared model in the synchronous data parallel mode with each responsible for 1/3 of 288 images in a minibatch. By adjusting the allocation proportionally to their capabilities, we can allocate 76 samples of the minibatch to 1080 Ti and P100 each and 136 samples to the V100. As a result, a total throughput of 421 images per second was observed, yielding a 9.6% improvement.

Figure 10 shows the scenario to train the MxNet Inception model on two machines: one with four P100 GPUs and another with four V100 GPUs. The two machines were capable of processing 563 and 960 images per second respectively. However, directly using them under MxNet only gave a total throughput of 1112 due to equal split of the minibatch among the eight GPUs. By giving V100 a bigger share (100 samples per GPU) and P100 a smaller share (60 samples per GPU), a higher throughput 1450 images/s was achieved, yielding a 30.4% improvement.

4.3 Combining Local Accumulation and Heterogeneous Distributed Training

Some training environments will naturally call for adopting multiple software techniques together. For instance, when training a DNN model on a cluster of machines consisting of different number or different types of GPUs, it is suitable to use both local accumulation and heterogeneous training.

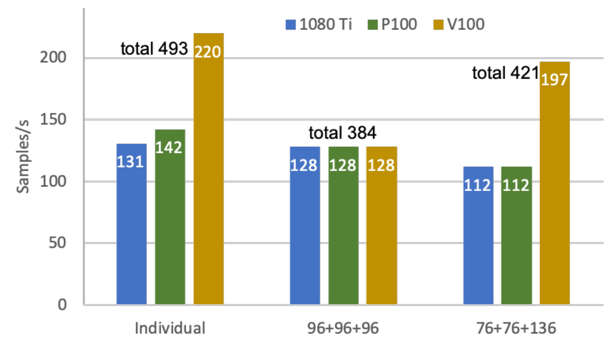

Figure 9: Train TensorFlow Inception model over ImageNet1K using three heterogeneous GPUs: V100, P100, 1080Ti. Left: three independent trainings. Center: Distributed synchronous training. Right: Distributed heterogeneous training.

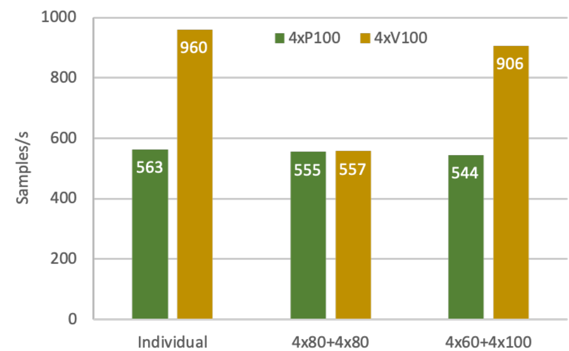

Figure 10: Train MxNet Inception model on two machines, one with four V100 GPUs and another with four P100. Left: two independent trainings. Center: Distributed synchronous SGD training when dividing the minibatch equally. Right: Distributed training with proportional batching.

Table 7: Local accumulation (LA) and heterogeneous training of ResNet-50 on two V100 and two 1080 Ti GPUs

V100 BATCH (LA)	1080 TI BATCH (LA)	TIME(H)
4K (64)	4K (64)	52.58
5K (80)	3K (48)	43.65
8K (128)		59.91

When two machines are available, equipped with two V100 GPUs and two 1080 Ti GPUs respectively, one has to decide how to best utilize them to train his model: either only using the faster machine or using both machines. Table 7 shows the results of combining heterogeneous batch allocation and local accumulation, training ResNet-50 using two machines homogeneously, using two machines heterogeneously, and only using the faster machine. In the first case,

both machines use 64 local accumulations and split a 8K minibatch evenly, leading to a training time of 52.58 hours. In the second case, by using both machines heterogeneously and giving more work to the V100 machine, the training could be shortened to 43.65 hours, a reduction of 17%. In the third case, when only the fast V100 machine was used, the training took 59.91 hours—the longest time. It indicates that a weaker machine could still contributed to the training positively and that local accumulation can be used to bridge the gap between different types of GPUs.

Experiment on the Transformer model shows similar results. Table 8 summarizes training of TensorFlow Transformer model using different local accumulations on a machine with two V100 GPUs and another machine with two 1080 Ti GPUs, connected via 100Gbps Ethernet. Due to memory limitation of 1080 Ti, per GPU batch size 2K was used. In the baseline setting, both GPUs used 12 local accumulations, resulting in a training time 14.67 hours. When adjusted to 14 accumulations on V100 and 10 accumulations on 1080 Ti, the training time was shortened to 12.65 hours—about 14% reduction. Obviously determining the suitable local accumulations for different types of GPUs is important, since different choices would yield different level of time reduction.

Table 8: Local accumulation (LA) and heterogeneous training using TensorFlow Transformer Model

LA(V100)	LA(1080Ti)	ACCURACY(%)	TIME(H)
12	12	26.10	14.67
16	8	25.96	13.92
14	10	25.97	12.65

Even if all GPUs are of the same type, we could have heterogeneous resource availability, for instance, when 12 V100 GPUs are available on on two machines connected by 100Gbps Ethernet—one supplying 8 GPUs and another 4 GPUs. Table 9 shows three different approaches of utilizing these resources for training the MxNet Transformer model: (1) Use both machines equally (i.e., 4 GPUs each) as supported over-the-shelf by current framework, (2) use only the 8 GPU machine, (3) heterogeneously split the work among the two machines. The first approach of running two equal workers on two machines resulted in a slow training of 32.06 hours. Using only the 8 GPUs on the single machine was faster, taking only 24.65 hours. By combining local accumulation and heterogeneous training, the same training was done in 20.76 hours with comparable quality, a reduction of around 11 hours and 4 hours with respective to first two approaches.

Table 9: Local accumulation (LA) and heterogeneous training using MxNet Transformer Model

GPUs	LA	ACCURACY(%)	TIME(H)
4 + 4 (HOMOGENEOUS)	8	26.56	32.06
8 (ONE MACHINE)	8	26.24	24.65
8 + 4	8	26.48	20.76

In the third approach, even though both workers used the same number of local accumulations before synchronizing with the kvstore (parameter server), the two per-worker batch sizes on these two workers differs significantly. Given the original code only supports training on a single machine, we made modifications to support distributed heterogeneous training: All workers shuffle (using a same sequence of seeds), split the dataset the same way, and are allocated the number of batches proportional to the product of its number of GPUs and its local accumulations (since all involved GPUs are of the same type in this case). We argue that such functionality can be easily incorporated into the underlying framework such as TensorFlow and MxNet so that neural network training can effectively leverage different types and different number of GPUs dynamically available in a cluster.

4.4 Monitoring and Cooling Until Early Stop

We experimented with controlling the cooling and early stop process using the TracEO algorithm on two MxNet models: ResNet-50 and Transformer.

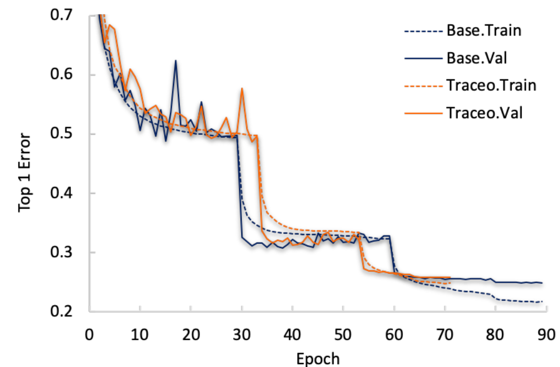

Figure 11: Train MxNet ResNet-50 model: 90-epoch baseline (Base) and using online algorithm (TracEO)

For training the ResNet-50 model, the TracEO algorithm expected a progress (or a fluctuation) of at least 0.002 within 5 epochs. Figure 11 depicts the top-1 training error (i.e., complement of accuracy) and validation error of this training compared with those of the statically fine-tuned 30-60-80 cooling schedule used in the literature. As the training cooled down by reducing the learning rate by a factor of 10, significant decrease of both metrics were observed. The TracEO algorithm chose different cooling points than the static baseline schedule, and stopped the training after 72 epochs when no significant progress was seen after the cool-down. As summarized in Table 10, it achieved a similar accuracy as the static baseline schedule, accelerating the training by 20% at the cost of 0.56% less accuracy.

Table 11 compares the training of the MxNet Transformer model using its hand-crafted schedule (baseline) and the TracEO algorithm. The baseline adjusted the learning rate at every step, with a gradual warmup in approximately 5 epochs followed by the cool-down scheme of keeping the learning rate inversely proportional to square root of the step number to the 30th epoch. The training

Table 10: Static vs TracEO schedule on MxNet ResNet-50

TECHNIQUE	EPOCHS	ACCURACY(%)	SPEEDUP(%)
STATIC	90	75.19	
TRACEO	72	74.63	20

took 24.65 hours and achieved a highest bleu score 26.24%. Since the TracEO algorithm operates at the boundary of epochs instead of steps, adjustment of hyperparameters is at a coarse granularity. We increased the warmup stage to 8 epochs to avoid abrupt change of the learning rate. We gave it a target bleu of 26%, similar to the baseline outcome. During the training, the algorithm reduced the learning rate by half when not enough progress was observed in the previous 5 epochs. As a result, even though the training warmed up slower than the baseline, it gradually caught up and was able to reach a bleu score of 26.26% in 22.86 hours after 28 epochs. In this sense, the algorithm was on par with the hand-crafted schedule despite using the discrete cooling-down scheme at the coarser granularity.

Table 11: Train Transformer Model using online algorithm.

TECHNIQUE	EPOCHS	BLEU(%)	TIME(H)
BASELINE	30	26.24	24.65
TRACEO	28	26.26	22.86

5 SUMMARY

In this paper we studied several software techniques to expedite training of DNNs, including use of local accumulation training, proportional batching to bridge the resource heterogeneity, and online monitoring and cooling algorithm for adjusting batch size and learning rate and for making early stop decisions.

Each of the techniques uses a different approach to reducing training time. Local accumulation training allows for large batches which reduces the number of synchronizations. Proportional batching accelerates training when running on a heterogeneous set of GPUs by avoiding the time that a faster GPU would need to wait for a slower one to finish. The TracEO algorithm advances the training schedule to the next change point when a setting appears to be unproductive and stops the training when no further progress is observed. The early termination of the training saves time over a fixed schedule.

These software techniques can be easily implemented in the DNN frameworks or applications. The techniques mostly complement each other as they all take different approaches to reducing training time. We studied the performance of these techniques experimentally. Our experiences indicated that integration of the above techniques can reduce the overall training time by 10% to 30% with comparable training quality.

REFERENCES

[1] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. In *NIPS'17 Workshop: Deep Learning at Supercomputer Scale*.

[2] Amodei, D., *et. al.* 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 48.

[3] C. Atkinson, B. McCane, and L. Szymanski. 2017. Increasing the accuracy of convolutional neural networks with progressive reinitialisation. In *2017 International Conference on Image and Vision Computing New Zealand (IVCNZ)*.

[4] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. 2017. FreezeOut: Accelerate Training by Progressively Freezing Layers. In *10th NIPS Workshop on Optimization for Machine Learning*.

[5] V. Codreanu, D. Podareanu, and V. Saletore. 2017. Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train. https://arxiv.org/abs/1711.04291

[6] David K. Duvenaud, Dougal Maclaurin, and Ryan P. Adams. 2016. Early Stopping as Nonparametric Variational Inference. In *Proc.of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016*.

[7] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. http://arxiv.org/abs/1706.02677

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. https://arxiv.org/abs/1712.00409

[10] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. https://arxiv.org/abs/1705.08741

[11] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. *Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications*. Technical Report MSR-TR-2018-13. Microsoft.

[12] Jia, X., *et. al.* 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. https://arxiv.org/abs/1807.11205

[13] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Machine Learning (ICLR)*.

[14] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*.

[15] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. 2017. Early Stopping without a Validation Set. http://arxiv.org/abs/1703.09580

[16] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. http://arxiv.org/abs/1804.07612

[17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations (ICLR)*.

[18] M. Ott, S. Edunov, D. Grangier, and M. Auli. 2018. Scaling Neural Machine Translation. https://arxiv.org/abs/1806.00187

[19] Lutz Prechelt. 1998. Early Stopping-But When?. In *Neural Networks: Tricks of the Trade*. Springer-Verlag.

[20] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. 2017. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc.

[21] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vision* (Dec. 2015).

[22] C. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network Training. https://arxiv.org/abs/1811.03600

[23] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. 2018. Don't Decay the Learning Rate, Increase the Batch Size. In *International Conference on Learning Representations (ICLR)*.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 1–9.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems 30*. https://arxiv.org/pdf/1706.03762.pdf

[26] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*.