**Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour**

|  |  |  |  |
| --- | --- | --- | --- |
| Priya Goyal | Piotr Dolla´r | Ross Girshick | Pieter Noordhuis |
| Lukasz Wesolowski | Aapo Kyrola | Andrew Tulloch | Yangqing Jia Kaiming He |

Facebook

# Abstract

*Deep learning thrives with large neural networks and large datasets. However, larger networks and larger datasets result in longer training times that impede re- search and development progress. Distributed synchronous SGD offers a potential solution to this problem by dividing SGD minibatches over a pool of parallel workers. Yet to make this scheme efficient, the per-worker workload must be large, which implies nontrivial growth in the SGD mini- batch size. In this paper, we empirically show that on the ImageNet dataset large minibatches cause optimization dif- ficulties, but when these are addressed the trained networks exhibit good generalization. Specifically, we show no loss of accuracy when training with large minibatch sizes up to 8192 images. To achieve this result, we adopt a hyper- parameter-free linear scaling rule for adjusting learning rates as a function of minibatch size and develop a new warmup scheme that overcomes optimization challenges*

40

35

ImageNet top-1 validation error

30

25

20

64 128 256 512 1k 2k 4k 8k 16k 32k 64k

mini-batch size

Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed syn- chronous SGD to minibatches of up to 8k images *while maintain- ing the top-1 error of small minibatch training*. For all minibatch sizes we set the learning rate as a *linear* function of the minibatch size and apply a simple warmup phase for the first few epochs of training. All other hyper-parameters are kept fixed. Using this simple approach, accuracy of our models is invariant to minibatch

*early in training. With these simple techniques, our Caffe2- based system trains ResNet-50 with a minibatch size of 8192 on 256 GPUs in one hour, while matching small minibatch accuracy. Using commodity hardware, our implementation*

arXiv:1706.02677v2

[cs.CV]

30 Apr 2018

*achieves ∼90% scaling efficiency when moving from 8 to 256 GPUs. Our findings enable training visual recognition models on internet-scale data with high efficiency.*

# Introduction

Scale matters. We are in an unprecedented era in AI research history in which the increasing data and model scale is rapidly improving accuracy in computer vision [[22](#_bookmark59), [41](#_bookmark78), [34](#_bookmark71), [35](#_bookmark72), [36](#_bookmark73), [16](#_bookmark53)], speech [[17](#_bookmark54), [40](#_bookmark77)], and natural lan- guage processing [[7](#_bookmark44), [38](#_bookmark75)]. Take the profound impact in com- puter vision as an example: visual representations learned by deep convolutional neural networks [[23](#_bookmark60), [22](#_bookmark59)] show excel- lent performance on previously challenging tasks like Ima- geNet classification [[33](#_bookmark70)] and can be transferred to difficult perception problems such as object detection and segmen-

size (up to an 8k minibatch size). Our techniques enable a lin-

ear reduction in training time with *∼*90% efficiency as we scale to large minibatch sizes, allowing us to train an accurate 8k mini- batch ResNet-50 model in 1 hour on 256 GPUs.

tation [[8](#_bookmark45), [10](#_bookmark47), [28](#_bookmark65)]. Moreover, this pattern generalizes: larger datasets and neural network architectures consistently yield improved accuracy across all tasks that benefit from pre- training [[22](#_bookmark59), [41](#_bookmark78), [34](#_bookmark71), [35](#_bookmark72), [36](#_bookmark73), [16](#_bookmark53)]. But as model and data scale grow, so does training time; discovering the potential and limits of large-scale deep learning requires developing novel techniques to keep training time manageable.

The goal of this report is to demonstrate the feasibility of, and to communicate a practical guide to, large-scale train- ing with distributed *synchronous* stochastic gradient descent (SGD). As an example, we scale ResNet-50 [[16](#_bookmark53)] training, originally performed with a minibatch size of 256 images (using 8 Tesla P100 GPUs, training time is 29 hours), to larger minibatches (see Figure [1](#_bookmark0)). In particular, we show that *with a large minibatch size of 8192, we can train ResNet-50 in 1 hour using 256 GPUs while maintaining*

*the same level of accuracy as the 256 minibatch baseline*. While distributed synchronous SGD is now commonplace, no existing results show that generalization accuracy can be maintained with minibatches as large as 8192 or that such high-accuracy models can be trained in such short time.

To tackle this unusually large minibatch size, we employ a simple and hyper-parameter-free *linear scaling rule* to ad- just the learning rate. While this guideline is found in ear- lier work [[21](#_bookmark58), [4](#_bookmark41)], its empirical limits are not well under- stood and informally we have found that it is not widely known to the research community. To successfully apply this rule, we present a new *warmup* strategy, *i.e*., a strategy of using lower learning rates at the start of training [[16](#_bookmark53)], to overcome early optimization difficulties. Importantly, not only does our approach match the baseline *validation* error, *but also yields training error curves that closely match the*

# Large Minibatch SGD

We start by reviewing the formulation of Stochastic Gra- dient Descent (SGD), which will be the foundation of our discussions in the following sections. We consider super- vised learning by minimizing a loss *L*(*w*) of the form:

*L*(*w*) = 1 *l*(*x, w*)*.* (1)

Σ

|*X*| *x∈X*

Here *w* are the weights of a network, *X* is a labeled training set, and *l*(*x, w*) is the loss computed from samples *x X* and their labels *y*. Typically *l* is the sum of a classification loss (*e.g*., cross-entropy) and a regularization loss on *w*.

∈

*Minibatch Stochastic Gradient Descent* [[32](#_bookmark69)], usually re- ferred to as simply as SGD in recent literature even though it operates on minibatches, performs the following update:

*small minibatch baseline.* Details are presented in §[2](#_bookmark1).

Our comprehensive experiments in §[5](#_bookmark18) show that *opti- mization* difficulty is the main issue with large minibatches,

*wt*+1 = *w*

1

*t n t*

− *η* Σ ∇*l*(*x, w* )*.* (2)

*x∈B*

rather than poor *generalization* (at least on ImageNet), in contrast to some recent studies [[20](#_bookmark57)]. Additionally, we show that the linear scaling rule and warmup generalize to more complex tasks including object detection and instance seg- mentation [[9](#_bookmark46), [31](#_bookmark68), [14](#_bookmark51), [28](#_bookmark65)], which we demonstrate via the recently developed Mask R-CNN [[14](#_bookmark51)]. We note that a ro- bust and successful guideline for addressing a wide range of minibatch sizes has not been presented in previous work. While the strategy we deliver is simple, its successful application requires correct implementation with respect to seemingly minor and often not well understood implemen- tation details within deep learning libraries. Subtleties in the implementation of SGD can lead to incorrect solutions that are difficult to discover. To provide more helpful guidance we describe common pitfalls and the relevant implementa-

tion details that can trigger these traps in §[3](#_bookmark13).

Our strategy applies regardless of framework, but achieving efficient linear scaling requires nontrivial com- munication algorithms. We use the open-source *Caffe2*[1](#_bookmark4) deep learning framework and *Big Basin* GPU servers [[24](#_bookmark61)], which operates efficiently using standard Ethernet network- ing (as opposed to specialized network interfaces). We de- scribe the systems algorithms that enable our approach to operate near its full potential in §[4](#_bookmark15).

The practical advances described in this report are help- ful across a range of domains. In an industrial domain, our system unleashes the potential of training visual models from internet-scale data, enabling training with *billions of images per day*. Of equal importance, in a research domain, we have found it to simplify migrating algorithms from a single-GPU to a multi-GPU implementation without requir- ing hyper-parameter search, *e.g*. in our experience migrat- ing Faster R-CNN [[31](#_bookmark68)] and ResNets [[16](#_bookmark53)] from 1 to 8 GPUs.

1[http://www.caffe2.ai](http://www.caffe2.ai/)

Here is a *minibatch* sampled from *X* and *n* = is the

*minibatch size*, *η* is the *learning rate*, and *t* is the iteration index. Note that in practice we use momentum SGD; we return to a discussion of momentum in §[3](#_bookmark13).

B |B|

## Learning Rates for Large Minibatches

Our goal is to use large minibatches in place of small minibatches while *maintaining training and generalization accuracy*. This is of particular interest in distributed learn- ing, because it can allow us to scale to multiple workers[2](#_bookmark6) us- ing simple data parallelism without reducing the per-worker workload and without sacrificing model accuracy.

As we will show in comprehensive experiments, we found that the following learning rate scaling rule is sur- prisingly effective for a broad range of minibatch sizes:

***Linear Scaling Rule:*** *When the minibatch size is multiplied by k, multiply the learning rate by k.*

All other hyper-parameters (weight decay, *etc*.) are kept un- changed. As we will show in §[5](#_bookmark18), the *linear scaling rule* can help us to not only match the accuracy between using small and large minibatches, but equally importantly, to largely match their training curves, which enables rapid debugging and comparison of experiments prior to convergence.

**Interpretation.** We present an informal discussion of the linear scaling rule and why it may be effective. Consider a network at iteration *t* with weights *wt*, and a sequence of *k* minibatches *j* for 0 *j < k* each of size *n*. We

B ≤

compare the effect of executing *k* SGD iterations with *small minibatches* B*j* and learning rate *η* versus a single iteration with a *large minibatch* ∪*j*B*j* of size *kn* and learning rate *η*ˆ.

2We use the terms ‘worker’ and ‘GPU’ interchangeably in this work, al- though other implementations of a ‘worker’ are possible. ‘Server’ denotes a set of 8 GPUs that does not require communication over a network.

According to ([2](#_bookmark3)), after *k* iterations of SGD with learning rate *η* and a minibatch size of *n* we have:

## Warmup

As we discussed, for large minibatches (*e.g*., 8k) the lin-

*wt*+*k*

= *wt*

*η* 1 *l*(*x, w*

*n*

Σ Σ− ∇

*j<k x∈Bj*

*t*+*j*

)*.* (3)

ear scaling rule breaks down when the network is changing rapidly, which commonly occurs in early stages of train- ing. We find that this issue can be alleviated by a properly

On the other hand, taking a single step with the large mini-

batch ∪*j*B*j* of size *kn* and learning rate *η*ˆ yields:

*w*ˆ = *w* − *η*ˆ 1 Σ Σ ∇*l*(*x, w* )*.* (4)

*t*+1

*t*

*kn*

*t*

*j<k x∈Bj*

designed *warmup* [[16](#_bookmark53)], namely, a strategy of using less ag- gressive learning rates at the start of training.

**Constant warmup.** The warmup strategy presented in [[16](#_bookmark53)]

uses a low *constant* learning rate for the first few epochs of

As expected, the updates differ, and it is *unlikely* that

*w*ˆ*t*+1 = *wt*+*k*. However, if we *could* assume *l*(*x, wt*) *l*(*x, wt*+*j*) for *j < k*, then setting *η*ˆ = *kη* would yield

∇

∇ ≈

*w*ˆ*t*+1 *wt*+*k*, and the updates from small and large mini- batch SGD would be similar. Although this is a strong as- sumption, we emphasize that if it were true the two updates are similar *only* if we set *η*ˆ = *kη*.

≈

The above interpretation gives intuition for one case where we may hope the linear scaling rule to apply. In our experiments with *η*ˆ = *kη* (and warmup), small and large minibatch SGD not only result in models with the same fi- nal accuracy, but also, the training curves match closely. Our empirical results suggest that the above approximation *might* be valid in large-scale, real-world data.

However, there are at least two cases when the condition *l*(*x, wt*) *l*(*x, wt*+*j*) will clearly not hold. First, in ini- tial training when the network is changing rapidly, it does not hold. We address this by using a warmup phase, dis- cussed in §[2.2](#_bookmark7). Second, minibatch size cannot be scaled in- definitely: while results are stable for a large range of sizes, beyond a certain point accuracy degrades rapidly. Interest- ingly, this point is as large as *∼*8k in ImageNet experiments.

∇ ≈ ∇

**Discussion.** The above linear scaling rule was adopted by Krizhevsky [[21](#_bookmark58)], if not earlier. However, Krizhevsky re- ported a 1% increase of error when increasing the minibatch size from 128 to 1024, whereas we show how to maintain accuracy across a much broader regime of minibatch sizes. Chen *et al*. [[5](#_bookmark42)] presented a comparison of numerous dis- tributed SGD variants, and although their work also em- ployed the linear scaling rule, it did not establish a small minibatch baseline. Li [[25](#_bookmark62)] (§4.6) showed distributed Ima- geNet training with minibatches up to 5120 without a loss in accuracy after convergence. However, their work did not demonstrate a hyper-parameter search-free rule for adjust- ing the learning rate as a function of minibatch size, which is a central contribution of our work.

In recent work, Bottou *et al*. [[4](#_bookmark41)] (§4.2) review theoretical

tradeoffs of minibatching and show that with the linear scal-

training. As we will show in §[5](#_bookmark18), we have found constant

warmup particularly helpful for prototyping object detec- tion and segmentation methods [[9](#_bookmark46), [31](#_bookmark68), [26](#_bookmark63), [14](#_bookmark51)] that fine-tune pre-trained layers together with newly initialized layers.

In our ImageNet experiments with a large minibatch of size *kn*, we have tried to train with the low learning rate of *η* for the first 5 epochs and then return to the target learn- ing rate of *η*ˆ = *kη*. However, given a large *k*, we find that this constant warmup is not sufficient to solve the optimiza- tion problem, and a transition out of the low learning rate warmup phase can cause the training error to spike. This leads us to propose the following gradual warmup.

**Gradual warmup.** We present an alternative warmup that *gradually* ramps up the learning rate from a small to a large value. This ramp avoids a sudden increase of the learning rate, allowing healthy convergence at the start of training. In practice, with a large minibatch of size *kn*, we start from a learning rate of *η* and increment it by a constant amount at each iteration such that it reaches *η*ˆ = *kη* after 5 epochs (re- sults are robust to the exact duration of warmup). After the warmup, we go back to the original learning rate schedule.

## Batch Normalization with Large Minibatches

Batch Normalization (BN) [[19](#_bookmark56)] computes statistics along the minibatch dimension: this breaks the independence of each sample’s loss, and changes in minibatch size change the underlying definition of the loss function being opti- mized. In the following we will show that a commonly used ‘shortcut’, which may appear to be a practical consideration to avoid communication overhead, is actually necessary for preserving the loss function when changing minibatch size. We note that ([1](#_bookmark2)) and ([2](#_bookmark3)) assume the per-sample loss *l*(*x, w*) is independent of all other samples. This is *not* the

case when BN is performed and activations are computed across samples. We write *lB*(*x, w*) to denote that the loss of a single sample *x* depends on the statistics of all samples in

its minibatch B. We denote thΣe loss over a single minibatch

B of size *n* as *L*(B*, w*) = 1

*x∈B lB*(*x, w*). With BN, the

*n*

ing rule, solvers follow the same training curve as a function of number of examples seen, and suggest the learning rate should not exceed a maximum rate independent of mini-

training set can be thought of as containing all distinct sub-

sets of size *n* drawn from the original training set *X*, which we denote as *Xn*. The training loss *L*(*w*) then becomes:

batch size (which justifies warmup). Our work empirically tests these theories with unprecedented minibatch sizes.

1

*L*(*w*) = |*Xn*|

*B*Σ*∈Xn*

*L*(B*, w*)*.* (5)

If we view as a ‘single sample’ in *Xn*, then the loss of each single sample is computed *independently*.

B

B

Note that the minibatch size *n* over which the BN statis- tics are computed is a key component of the loss: if the per- worker minibatch sample size *n* is changed, *it changes the underlying loss function L that is optimized*. More specif- ically, the mean/variance statistics computed by BN with different *n* exhibit different levels of random variation.

In the case of distributed (and multi-GPU) training, if the per-worker sample size *n* is kept fixed and the total mini- batch size is *kn*, it can be viewed a minibatch of *k* samples with each sample B*j* independently selected from *Xn*, so

the underlying loss function is unchanged and is still de-

contributed by *ε*(*x, wt*). If there is no weight decay term, there are many equivalent ways of scaling the learning rate, including scaling the term *ε*(*x, wt*). However, as can be seen from ([8](#_bookmark14)), in general this is *not* the case. We summarize these observations in the following remark:

*Remark 1: Scaling the cross-entropy loss is*

not *equivalent to scaling the learning rate.*

**Momentum correction.** Momentum SGD is a commonly adopted modification to the vanilla SGD in ([2](#_bookmark3)). A reference implementation of momentum SGD has the following form: *u* = *mu* + 1 Σ ∇*l*(*x, w* )

*t*+1

*t*

*n*

*t*

(9)

fined in *Xn*. Under this point of view, in the BN setting

*wt*+1

= *wt*

− *ηu*

*x∈B*

*t*+1

*.*

after seeing *k* minibatches B*j*, ([3](#_bookmark8)) and ([4](#_bookmark9)) become:

Σ

*wt*+*k* = *wt* − *η* ∇*L*(B*j, wt*+*j*)*,* (6)

*j<k*

Here *m* is the momentum decay factor and *u* is the update tensor. A popular variant absorbs the learning rate *η* into the update tensor. Substituting *vt* for *ηut* in ([9](#_bookmark11)) yields:

*w*ˆ = *w*

− *η*ˆ1 Σ ∇*L*(B *, w* )*.* (7)

1 Σ

*v* = *mv* + *η* ∇*l*(*x, w* )

*t*+1

*t k j t j<k*

*t*+1 *t*

*w* = *w* − *v*

*t*+1

*t*

*t*+1

*n t*

*.x∈B*

(10)

*keep the per-worker sample size n constant when we change the number of workers k*.

In this work, we use *n* = 32 which has performed well for a wide range of datasets and networks [[19](#_bookmark56), [16](#_bookmark53)]. If *n* is adjusted, it should be viewed as a hyper-parameter of BN,

For a fixed *η*, the two are equivalent. However, we note that while *u* only depends on the gradients and is independent of *η*, *v* is entangled with *η*. When *η* changes, to maintain equivalence with the reference variant in ([9](#_bookmark11)), the update for

*v* should be: *vt*+1 = *m ηt*+1 *vt* + *ηt*+1 1 ∇*l*(*x, wt*). We

Σ

Following similar logic as in §[2.1](#_bookmark5), we set *η*ˆ = *kη* and *we*

*ηt*

*n*

not of distributed training. We also note that *the BN statis-*

*tics should* not *be computed across all workers*, not only for

refer to the factor *ηt*+1

*t*

*η*

as the *momentum correction*. We

the sake of reducing communication, but also for maintain- ing the same underlying loss function being optimized.

# Subtleties and Pitfalls of Distributed SGD

In practice a distributed implementation has many sub- tleties. Many common implementation errors change the definitions of hyper-parameters, leading to models that train but whose error may be higher than expected, and such is-

found that this is especially important for stabilizing train- ing when *ηt*+1 *ηt*, otherwise the history term *vt* is too small which leads to instability (for *ηt*+1 *< ηt* momentum correction is less critical). This leads to our second remark:

*Remark 2: Apply momentum correction after changing learning rate if using (*[*10*](#_bookmark12)*).*

**Gradient aggregation.** For *k* workers each with a per- worker minibatch of size *n*, following ([4](#_bookmark9)), gradient aggre- gation must be performed over the entire set of *kn* examples

Σ Σ

sues can be difficult to discover. While the remarks below

according to 1

*j x∈B*

*kn*

*j*

*l*(*x, wt*). Loss layers are typi-

are straightforward, they are important to consider explic- itly to faithfully implement the underlying solver.

**Weight decay.** Weight decay is actually the outcome of the gradient of an L2-regularization term *in the loss function*. More formally, the per-sample loss in ([1](#_bookmark2)) can be written as

*l*(*x, w*) = *λ* ǁ*w*ǁ2 + *ε*(*x, w*). Here *λ* ǁ*w*ǁ2 is the sample-

cally implemented to compute an average loss over their *lo- cal* input, which amounts to computing a per-worker loss of *l*(*x, wt*)*/n*. Given this, correct aggregation requires *av- eraging* the *k* gradients in order to recover the missing 1*/k* factor. However, standard communication primitives like allreduce [[11](#_bookmark48)] perform summing, not averaging. Therefore,

2 2 it is more efficient to absorb the 1*/k* scaling into the loss,

Σ

independent L2 regularization on the weights and *ε*(*x, w*) is a sample-dependent term such as the cross-entropy loss. The SGD update in ([2](#_bookmark3)) can be written as:

*w* = *w* − *ηλw* − *η* 1 Σ ∇*ε*(*x, w* )*.* (8)

*t*+1

*t*

*t*

*n*

*t*

*x∈B*

in which case only the loss’s gradient with respect to its in- put needs to be scaled, removing the need to scale the entire gradient vector. We summarize this as follows:

*Remark 3: Normalize the per-worker loss by*

total *minibatch size kn,* not *per-worker size n.*

In practice, usually only the sample-dependent term *ε*(*x, wt*) is computed by backprop; the term *λwt* is computed *separately* and added to the aggregated gradients

Σ ∇

We also note that it may be incorrect to ‘cancel *k*’ by setting *η*ˆ = *η* (not *kη*) and normalizing the loss by 1*/n* (not 1*/kn*), which can lead to incorrect weight decay (see *Remark 1*).

**Data shuffling.** SGD is typically analyzed as a process that samples data randomly *with replacement*. In practice, com- mon SGD implementations apply *random shuffling* of the training set during each SGD epoch, which can give better results [[3](#_bookmark40), [13](#_bookmark50)]. To provide fair comparisons with baselines that use shuffling (*e.g*., [[16](#_bookmark53)]), we ensure the samples in one epoch done by *k* workers are from a single consistent ran- dom shuffling of the training set. To achieve this, for each epoch we use a random shuffling that is partitioned into *k* parts, each of which is processed by one of the *k* workers. Failing to correctly implement random shuffling in multiple workers may lead to noticeably different behavior, which may contaminate results and conclusions. In summary:

*Remark 4: Use a single random shuffling of the training data (per epoch) that is divided amongst all k workers.*

# Communication

In order to scale beyond the 8 GPUs in a single Big Basin server [[24](#_bookmark61)], gradient aggregation has to span across servers on a network. To allow for near perfect linear scaling, the aggregation must be performed *in parallel* with backprop. This is possible because there is no data dependency be- tween gradients across layers. Therefore, as soon as the gra- dient for a layer is computed, it is aggregated across work- ers, while gradient computation for the next layer continues (as discussed in [[5](#_bookmark42)]). We give full details next.

## Gradient Aggregation

For every gradient, aggregation is done using an *allre- duce* operation (similar to the MPI collective operation *MPI Allreduce* [[11](#_bookmark48)]). Before allreduce starts every GPU has its locally computed gradients and after allreduce completes every GPU has the sum of all *k* gradients. As the number of parameters grows and compute performance of GPUs in- creases, it becomes harder to hide the cost of aggregation in the backprop phase. Training techniques to overcome these effects are beyond the scope of this work (*e.g*., quantized gradients [[18](#_bookmark55)], Block-Momentum SGD [[6](#_bookmark43)]). However, at the scale of this work, collective communication was not a bottleneck, as we were able to achieve near-linear SGD scaling by using an optimized allreduce implementation.

Our implementation of allreduce consists of three phases for communication within and across servers: (1) buffers from the 8 GPUs within a server are summed into a sin- gle buffer for each server, (2) the results buffers are shared and summed across all servers, and finally (3) the results are broadcast onto each GPU. For the local reduction and broadcast in phases (1) and (3) we used NVIDIA Collective Communication Library (NCCL)[3](#_bookmark16) for buffers of size 256 KB or more and a simple implementation consisting of a

3<https://developer.nvidia.com/nccl>

number of GPU-to-host memory copies and a CPU reduc- tion otherwise. NCCL uses GPU kernels to accelerate in- traserver collectives, so this approach dedicates more time on the GPU to backprop while using the CPU resources that would otherwise have been idle to improve throughput.

For interserver allreduce, we implemented two of the best algorithms for bandwidth-limited scenarios: the *re- cursive halving and doubling algorithm* [[30](#_bookmark67), [37](#_bookmark74)] and the *bucket algorithm* (also known as the ring algorithm) [[2](#_bookmark39)].

For both, each server sends and receives 2 *p−*1 *b* bytes of data, where *b* is the buffer size in bytes and *p* is the num-

*p*

ber of servers. While the halving/doubling algorithm con- sists of 2 log2(*p*) communication steps, the ring algorithm consists of 2(*p* 1) steps. This generally makes the halv- ing/doubling algorithm faster in latency-limited scenarios (*i.e*., for small buffer sizes and/or large server counts). In practice, we found the halving/doubling algorithm to per- form much better than the ring algorithm for buffer sizes up to a million elements (and even higher on large server counts). On 32 servers (256 GPUs), using halving/doubling

−

led to a speedup of 3× over the ring algorithm.

The halving/doubling algorithm consists of a reduce- scatter collective followed by an allgather. In the first step of reduce-scatter, servers communicate in pairs (rank 0 with 1, 2 with 3, *etc*.), sending and receiving for different halves of their input buffers. For example, rank 0 sends the second half of its buffer to 1 and receives the first half of the buffer from 1. A reduction over the received data is performed be- fore proceeding to the next step, where the distance to the destination rank is doubled while the data sent and received is halved. After the reduce-scatter phase is finished, each server has a portion of the final reduced vector.

This is followed by the allgather phase, which retraces the communication pattern from the reduce-scatter in re- verse, this time simply concatenating portions of the final reduced vector. At each server, the portion of the buffer that was being sent in the reduce-scatter is received in the all- gather, and the portion that was being received is now sent. To support non-power-of-two number of servers, we used the *binary blocks algorithm* [[30](#_bookmark67)]. This is a generalized version of the halving/doubling algorithm where servers are partitioned into power-of-two blocks and two additional communication steps are used, one immediately after the intrablock reduce-scatter and one before the intrablock all- gather. Non-power-of-two cases have some degree of load imbalance compared to power-of-two, though in our runs

we did not see significant performance degradation.

## Software

The allreduce algorithms described are implemented in

*Gloo*[4](#_bookmark17), a library for collective communication. It supports

4<https://github.com/facebookincubator/gloo>

multiple communication contexts, which means no addi- tional synchronization is needed to execute multiple allre- duce instances in parallel. Local reduction and broadcast (described as phases (1) and (3)) are pipelined with inter- server allreduce where possible.

*Caffe2* supports multi-threaded execution of the compute graph that represents a training iteration. Whenever there is no data dependency between subgraphs, multiple threads can execute those subgraphs in parallel. Applying this to backprop, local gradients can be computed in sequence, without dealing with allreduce or weight updates. This means that during backprop, the set of *runnable subgraphs* may grow faster than we can execute them. For subgraphs that contain an allreduce run, all servers must choose to exe- cute the same subgraph from the set of runnable subgraphs. Otherwise, we risk distributed deadlock where servers are attempting to execute non-intersecting sets of subgraphs. With allreduce being a *collective* operation, servers would time out waiting. To ensure correct execution we impose a partial order on these subgraphs. This is implemented using a cyclical control input, where completion of the *n*-th allre- duce unblocks execution of the (*n* + *c*)-th allreduce, with *c* being the maximum number of concurrent allreduce runs. Note that this number should be chosen to be lower than the number of threads used to execute the full compute graph.

## Hardware

We used Facebook’s Big Basin [[24](#_bookmark61)] GPU servers for our experiments. Each server contains 8 NVIDIA Tesla P100 GPUs that are interconnected with NVIDIA NVLink. For local storage, each server has 3.2TB of NVMe SSDs. For network connectivity, the servers have a Mellanox ConnectX-4 50Gbit Ethernet network card and are con- nected to Wedge100 [[1](#_bookmark38)] Ethernet switches.

We have found 50Gbit of network bandwidth sufficient for distributed synchronous SGD for ResNet-50, per the following analysis. ResNet-50 has approximately 25 mil- lion parameters. This means the total size of parameters is

25 106 sizeof(float) = 100MB. Backprop for ResNet-50

· ·

on a single NVIDIA Tesla P100 GPU takes 120 ms. Given that allreduce requires *∼*2 bytes on the network compared to the value it operates on, this leads to a peak bandwidth re- quirement of 200MB*/*0*.*125s = 1600MB/s, or 12.8 Gbit/s, not taking into account communication overhead. When we add a smudge factor for network overhead, we reach a peak bandwidth requirement for ResNet-50 of *∼*15 Gbit/s.

×

As this peak bandwidth requirement only holds during backprop, the network is free to be used for different tasks that are less latency sensitive then aggregation (*e.g*. reading data or saving network snapshots) during the forward pass.

# Main Results and Analysis

Our main result is that we can train ResNet-50 [[16](#_bookmark53)] on ImageNet [[33](#_bookmark70)] using 256 workers in one hour, while match- ing the accuracy of small minibatch training. Applying the linear scaling rule along with a warmup strategy allows us to seamlessly scale between small and large minibatches (up to 8k images) without tuning additional hyper-parameters or impacting accuracy. In the following subsections we:

(1) describe experimental settings, (2) establish the effec- tiveness of large minibatch training, (3) perform a deeper experimental analysis, (4) show our findings generalize to object detection/segmentation, and (5) provide timings.

## Experimental Settings

The 1000-way ImageNet classification task [[33](#_bookmark70)] serves as our main experimental benchmark. Models are trained on the *∼*1.28 million training images and evaluated by top- 1 error on the 50,000 validation images.

We use the ResNet-50 [[16](#_bookmark53)] variant from [[12](#_bookmark49)], noting that the stride-2 convolutions are on 3 3 layers instead of on

×

1 1 layers as in [[16](#_bookmark53)]. We use Nesterov momentum [[29](#_bookmark66)] with *m* of 0.9 following [[12](#_bookmark49)] but note that standard mo- mentum as was used in [[16](#_bookmark53)] is equally effective. We use a weight decay *λ* of 0.0001 and following [[16](#_bookmark53)] we do not ap- ply weight decay on the learnable BN coefficients (namely, *γ* and *β* in [[19](#_bookmark56)]). In order to keep the training objective fixed, which depends on the BN batch size *n* as described in §[2.3](#_bookmark10), we use *n* = 32 throughout, regardless of the overall minibatch size. As in [[12](#_bookmark49)], we compute the BN statistics using running average (with momentum 0.9).

×

All models are trained for 90 epochs regardless of mini- batch sizes. We apply the *linear scaling rule* from §[2.1](#_bookmark5) and use a learning rate of *η* = 0*.*1 *kn* that is linear in the mini- batch size *kn*. With *k* = 8 workers (GPUs) and *n* = 32 samples per worker, *η* = 0*.*1 as in [[16](#_bookmark53)]. We call this num- ber (0*.*1 *kn* ) the *reference learning rate*, and reduce it by 1*/*10 at the 30-th, 60-th, and 80-th epoch, similar to [[16](#_bookmark53)].

256

·

256

·

We adopt the initialization of [[15](#_bookmark52)] for all convolutional layers. The 1000-way fully-connected layer is initialized by drawing weights from a zero-mean Gaussian with standard deviation of 0.01. We have found that although SGD with a small minibatch is not sensitive to initialization due to BN, this is not the case for a substantially large minibatch. Addi- tionally we require an appropriate warmup strategy to avoid optimization difficulties in early training.

For BN layers, the learnable scaling coefficient *γ* is ini- tialized to be 1, *except for each residual block’s last BN where γ is initialized to be 0.* Setting *γ* = 0 in the last BN of each residual block causes the forward/backward signal ini- tially to propagate through the identity shortcut of ResNets, which we found to ease optimization at the start of training. This initialization improves all models but is particularly helpful for large minibatch training as we will show.

We use scale and aspect ratio data augmentation [[36](#_bookmark73)] as in [[12](#_bookmark49)]. The network input image is a 224 224 pixel ran- dom crop from an augmented image or its horizontal flip. The input image is normalized by the per-color mean and standard deviation, as in [[12](#_bookmark49)].

×

**Handling random variation.** As models are subject to random variation in training, we compute a model’s error rate as the *median* error of the final 5 epochs. Moreover, we report the mean and standard deviation (std) of the error from *5 independent runs*. This gives us more confidence in our results and also provides a measure of model stability.

The random variation of ImageNet models has generally not been reported in previous work (largely due to resource limitations). We emphasize that ignoring random variation may cause unreliable conclusions, especially if results are from a single trial, or the best of many.

**Baseline.** Under these settings, we establish a ResNet-50 baseline using *k* = 8 (8 GPUs in one server) and *n* = 32 images per worker (minibatch size of *kn* = 256), as in [[16](#_bookmark53)]. Our baseline has a top-1 validation error of 23.60% 0.12. As a reference, ResNet-50 from fb.resnet.torch [[12](#_bookmark49)] has 24.01% error, and that of the original ResNet paper [[16](#_bookmark53)] has 24.7% under weaker data augmentation.

±

## Optimization or Generalization Issues?

We establish our main results on large minibatch train- ing by exploring optimization and generalization behaviors. We will demonstrate that with a proper warmup strategy, large minibatch SGD can both match the *training curves* of small minibatch SGD and also match the *validation* error. In other words, in our experiments both *optimization* and *generalization* of large minibatch training matches that of small minibatch training. Moreover, in §[5.4](#_bookmark35) we will show that these models exhibit good generalization behavior to the object detection/segmentation transfer tasks, matching the transfer quality of small minibatch models.

For the following results, we use *k* = 256 and *n* = 32, which results in a minibatch size *kn* = 8k (we use ‘1k’ to denote 1024). As discussed, our baseline has a mini- batch size of *kn* = 256 and a reference learning rate of *η* = 0*.*1. Applying the linear scaling rule gives *η* = 3*.*2 as the reference learning rate for our large minibatch runs. We test three warmup strategies as discussed in §[2.2](#_bookmark7): *no warmup*, *constant warmup* with *η* = 0*.*1 for 5 epochs, and *gradual warmup* which starts with *η* = 0*.*1 and is linearly increased to *η* = 3*.*2 over 5 epochs. All models are trained from scratch and all other hyper-parameters are kept fixed. We emphasize that while better results for any particular minibatch size could be obtained by optimizing hyper-parameters for that case; *our goal is to match er- rors across minibatch sizes by using a general strategy that avoids hyper-parameter tuning for each minibatch size*.

256 32 8k 3.2 24.84 *±*0.37

256 32 8k 3.2 25.88 *±*0.56

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *k* | *n* | *kn* | *η* | top-1 error (%) |
| baseline (single server) | 8 | 32 | 256 | 0.1 | 23.60 *±*0.12 |
| no warmup, Figure [2a](#_bookmark21) constant warmup, Figure [2b](#_bookmark22)  gradual warmup, Figure [2c](#_bookmark23) |  | | | | |

256 32 8k 3.2 23.74 *±*0.09

Table 1. **Validation error on ImageNet using ResNet-50** (mean and std computed over 5 trials). We compare the small minibatch model (*kn*=256) with large minibatch models (*kn*=8k) with vari- ous warmup strategies. Observe that the top-1 validation error for small and large minibatch training (with gradual warmup) is quite close: 23.60% *±*0.12 *vs*. 23.74% *±*0.09, respectively.

**Training error.** Training curves are shown in Figure [2](#_bookmark24). With no warmup ([2a](#_bookmark21)), the training curve for large minibatch of *kn* = 8k is inferior to training with a small minibatch of *kn* = 256 across all epochs. A constant warmup strategy ([2b](#_bookmark22)) actually degrades results: although the small constant learning rate can decrease error during warmup, the error spikes immediately after and training never fully recovers.

Our main result is that with gradual warmup, large mini- batch training error matches the baseline training curve ob- tained with small minibatches, see Figure [2c](#_bookmark23). Although the large minibatch curve starts higher due to the low *η* in the warmup phase, it catches up shortly thereafter. Af- ter about 20 epochs, the small and large minibatch training curves match closely. The comparison between no warmup and gradual warmup suggests that *large minibatch sizes are challenged by optimization difficulties in early training* and if these difficulties are addressed, the training error and its curve can match a small minibatch baseline closely.

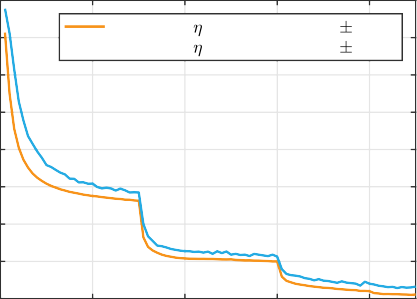
**Validation error.** Table [1](#_bookmark20) shows the *validation error* for the three warmup strategies. The no-warmup variant has

*∼*1.2% higher validation error than the baseline which is likely caused by the *∼*2*.*1% increase in training error (Fig- ure [2a](#_bookmark21)), rather than overfitting or other causes for poor gen-

eralization. This argument is further supported by our grad- ual warmup experiment. The gradual warmup variant has a *validation* error within 0.14% of the baseline (noting that std of these estimates is *∼*0.1%). Given that the final train- ing errors (Figure [2c](#_bookmark23)) match nicely in this case, it shows that *if the optimization issues are addressed, there is no apparent generalization degradation observed using large minibatch training*, even if the minibatch size goes from 256 to 8k.

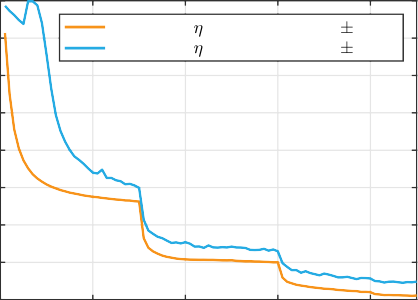
Finally, Figure [4](#_bookmark26) shows both the training and valida- tion curves for the large minibatch training with gradual warmup. As can be seen, validation error starts to match the baseline closely after the second learning rate drop; ac- tually, the validation curves can match earlier if BN statis- tics are recomputed prior to evaluating the error instead of using the running average (see also caption in Figure [4](#_bookmark26)).

100



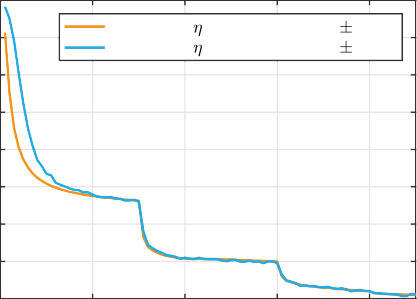
*kn*=256, = 0.1, 23.60% 0.12

*kn*= 8k, = 3.2, 24.84% 0.37



*kn*=256, = 0.1, 23.60% 0.12

*kn*= 8k, = 3.2, 25.88% 0.56



*kn*=256, = 0.1, 23.60% 0.12

*kn*= 8k, = 3.2, 23.74% 0.09

90

80

70

training error %

60

50

40

30

20

0 20 40 60 80

epochs

1. no warmup

0 20 40 60 80

epochs

1. constant warmup

0 20 40 60 80

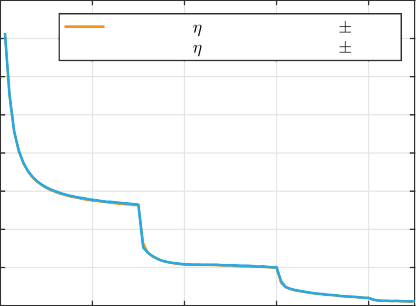
epochs

1. gradual warmup

Figure 2. **Warmup.** Training error curves for minibatch size 8192 using various warmup strategies compared to minibatch size 256.

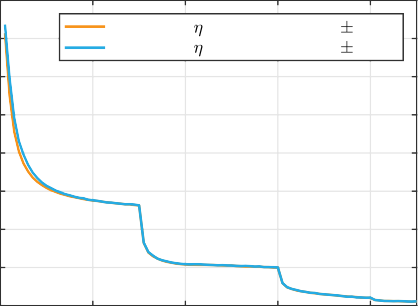
*Validation* error (mean*±*std of 5 runs) is shown in the legend, along with minibatch size *kn* and reference learning rate *η*.

100



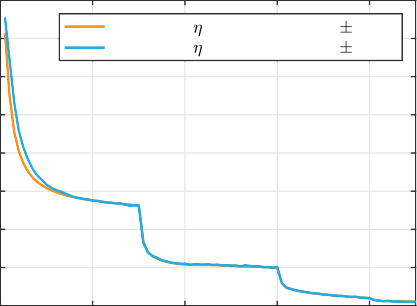
*kn*=256, = 0.1, 23.60% 0.12

*kn*=128, = 0.05 23.49% 0.12



*kn*=256, = 0.1, 23.60% 0.12

*kn*=512, = 0.2, 23.48% 0.09



*kn*=256, = 0.1, 23.60% 0.12

*kn*= 1k, = 0.4, 23.53% 0.08

90

80

70

training error %

60

50

40

30

20

100

90

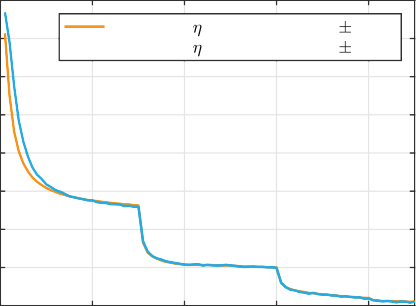
80

0 20 40 60 80

0 20 40 60 80

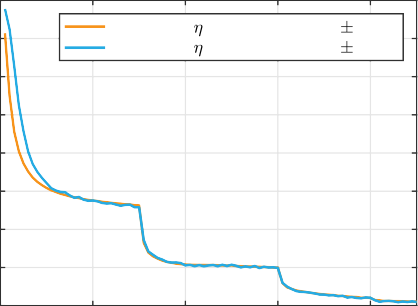
0 20 40 60 80

70



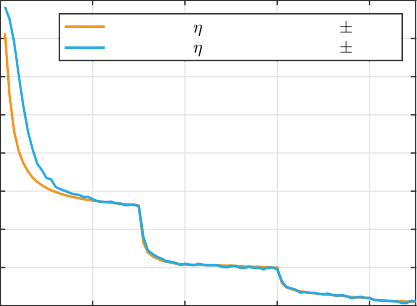
*kn*=256, = 0.1, 23.60% 0.12

*kn*= 2k, = 0.8, 23.49% 0.11



*kn*=256, = 0.1, 23.60% 0.12

*kn*= 4k, = 1.6, 23.56% 0.12



*kn*=256, = 0.1, 23.60% 0.12

*kn*= 8k, = 3.2, 23.74% 0.09

training error %

60

50

40

30

20

100

90

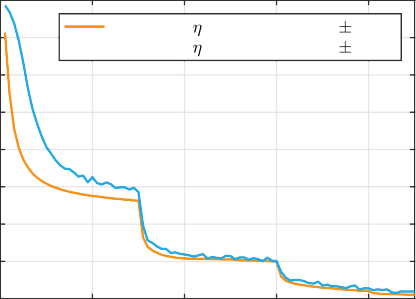
80

0 20 40 60 80

0 20 40 60 80

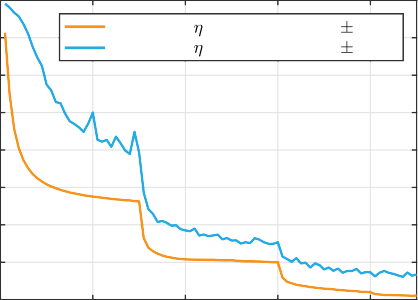
0 20 40 60 80

70



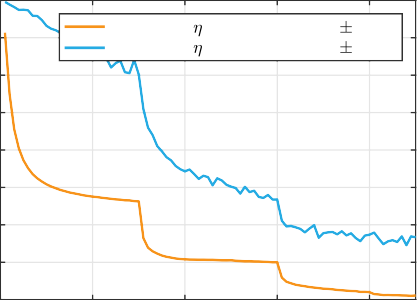
*kn*=256, = 0.1, 23.60% 0.12

*kn*=16k, = 6.4, 24.79% 0.27



*kn*=256, = 0.1, 23.60% 0.12

*kn*=32k, =12.8, 27.55% 0.28



*kn*=256, = 0.1, 23.60% 0.12

*kn*=64k, =25.6, 33.96% 0.80

training error %

60

50

40

30

20

0 20 40 60 80

epochs

0 20 40 60 80

epochs

0 20 40 60 80

epochs

Figure 3. **Training error *vs*. minibatch size.** Training error curves for the 256 minibatch baseline and larger minibatches using gradual warmup and the linear scaling rule. Note how the training curves closely match the baseline (aside from the warmup period) up through 8k minibatches. *Validation* error (mean*±*std of 5 runs) is shown in the legend, along with minibatch size *kn* and reference learning rate *η*.

100 100

*kn*=256, =0.1 [train]

*kn*=256, =0.1 [val]

*kn*=8k, =3.2 [train]

*kn*=8k, =3.2 [val]



*kn*=256, = 0.1, 23.60% 0.12

*kn*=256, = 0.2, 23.68% 0.09

80 80

training error %

60 60

error %

40 40

20

0 20 40 60 80

epochs

20

0 20 40 60 80

epochs

Figure 4. **Training and validation curves** for large minibatch SGD with gradual warmup *vs*. small minibatch SGD. Both sets of curves match closely after training for sufficient epochs. We note that the BN statistics (for inference only) are computed us- ing *running* average, which is updated less frequently with a large minibatch and thus is noisier in early training (this explains the larger variation of the validation error in early epochs).

## Analysis Experiments

**Minibatch size *vs*. error.** Figure [1](#_bookmark0) (page 1) shows top- 1 validation error for models trained with minibatch sizes ranging from of 64 to 65536 (64k). For all models we used the linear scaling rule and set the reference learning rate as *η* = 0*.*1 *kn* . For models with *kn >* 256, we used the gradual warmup strategy always starting with *η* = 0*.*1 and increasing linearly to the reference learning rate after 5 epochs. Figure [1](#_bookmark0) illustrates that validation error remains stable across a broad range of minibatch sizes, from 64 to 8k, after which it begins to increase. Beyond 64k training diverges when using the linear learning rate scaling rule.[5](#_bookmark28)

256

·

**Training curves for various minibatch sizes.** Each of the nine plots in Figure [3](#_bookmark25) shows the top-1 training error curve for the 256 minibatch baseline (orange) and a second curve corresponding to different size minibatch (blue). Valida- tion errors are shown in the plot legends. As minibatch size increases, all training curves show some divergence from the baseline at the start of training. However, in the cases where the final validation error closely matches the base- line (*kn* 8k), the training curves also closely match after the initial epochs. When the validation errors do not match (*kn* 16k), there is a noticeable gap in the training curves for all epochs. This suggests that when comparing a new setting, the training curves can be used as a reliable proxy for success well before training finishes.

≤

≥

**Alternative learning rate rules.** Table [2a](#_bookmark29) shows results for multiple learning rates. For small minibatches (*kn* = 256),

5We note that because of the availability of hardware, we *simulated* dis- tributed training of very large minibatches ( 12k) on a single server by us- ing multiple gradient accumulation steps between SGD updates. We have thoroughly verified that gradient accumulation on a single server yields equivalent results relative to distributed training.

*≥*

Figure 5. **Training curves for small minibatches with different learning rates** *η***.** As expected, changing *η* results in curves that *do not match*. This is in contrast to changing batch-size (and linearly scaling *η*), which results in curves that *do match*, *e.g*. see Figure [3](#_bookmark25).

*η* = 0*.*1 gives best error but slightly smaller or larger *η* also work well. When applying the linear scaling rule with a minibatch of 8k images, the optimum error is also achieved with *η* = 0*.*1 32, showing the successful application of the linear scaling rule. However, in this case results are more sensitive to changing *η*. In practice we suggest to use a minibatch size that is not close to the breaking point.

·

Figure [5](#_bookmark27) shows the training curves of a 256 minibatch using *η* = 0*.*1 or 0*.*2. It shows that changing the learning rate *η* in general changes the overall shapes of the train- ing curves, even if the final error is similar. Contrasting this result with the success of the linear scaling rule (that can match both the final error and the training curves when minibatch sizes change) may reveal some underlying invari- ance maintained between small and large minibatches.

We also show two alternative strategies: keeping *η* fixed at 0.1 or using 0*.*1 32 according to the square root scaling rule that was justified theoretically in [[21](#_bookmark58)] on grounds that it scales *η* by the inverse amount of the reduction in the gradi- ent estimator’s standard deviation. For fair comparisons we also use gradual warmup for 0*.*1 32. Both policies work poorly in practice as the results show.

√

·

√

·

**Batch Normalization** *γ* **initialization.** Table [2b](#_bookmark30) controls for the impact of the new BN *γ* initialization introduced in

§[5.1](#_bookmark19). We show results for minibatch sizes 256 and 8k with the standard BN initialization (*γ* = 1 for all BN layers) and with our initialization (*γ* = 0 for the final BN layer of each residual block). The results show improved per- formance with *γ* = 0 for both minibatch sizes, and the improvement is slightly larger for the 8k minibatch size. This behavior also suggests that large minibatches are more easily affected by optimization difficulties. We expect that improved optimization and initialization methods will help push the boundary of large minibatch training.

**ResNet-101.** Results for ResNet-101 [[16](#_bookmark53)] are shown in Ta- ble [2c](#_bookmark31). Training ResNet-101 with a batch-size of *kn* = 8k

*kn η* top-1 error (%) 40

ImageNet top-1 validation error

256 0*.*05 23.92 *±*0.10

256 0*.*10 23.60 *±*0.12 35

256 0*.*20 23.68 *±*0.09

8k 0*.*05 *·* 32 24.27 *±*0.08

8k 0*.*10 *·* 32 23.74 *±*0.09 30

8k 0*.*20 *·* 32 24.05 *±*0.18

8k 0*.*1*√*0 41.67 *±*0.10 25

8k 0*.*10 *·* 32 26.22 *±*0.03

1. **Comparison of learning rate scaling rules.** A reference learning rate of *η* = 0*.*1 works best for *kn* = 256 (23.68% error). The linear scal- ing rule suggests *η* = 0*.*1 32 when *kn* = 8k, which again gives best performance (23.74% error). Other ways of scaling *η* give worse results.

*·*

*kn η γ*-init top-1 error (%) 256 0.1 1.0 23.84 *±*0.18

256 0.1 0.0 23.60 *±*0.12

8k 3.2 1.0 24.11 *±*0.07

8k 3.2 0.0 23.74 *±*0.09

1. **Batch normalization** *γ* **initialization.** Initializing *γ* = 0 in the *last* BN layer of each residual block improves results for both small and large minibatches. This initialization leads to better optimization behavior which has a larger positive impact when training with large minibatches.

|  |  |  |  |
| --- | --- | --- | --- |
| model type | *kn* | *η* | top-1 error (%) |
| ResNet-101  ResNet-101 | 256  8k | 0.1  3.2 | 22.08 *±*0.06  22.36 *±*0.09 |

1. **The linear scaling rule applied to ResNet-101.** The difference in error is about 0.3% between small and large minibatch training.

Table 2. **ImageNet classification experiments.** Unless noted all experiments use ResNet-50 and are averaged over 5 trials.

and a linearly scaled *η* = 3*.*2 results in an error of 22.36% *vs*. the *kn* = 256 baseline which achieves 22.08% with *η* = 0*.*1. In other words, ResNet-101 trained with mini- batch 8k has a small 0.28% increase in error *vs*. the baseline. It is likely that the minibatch size of 8k lies on the edge of the useful minibatch training regime for ResNet-101, simi- larly to ResNet-50 (see Figure [1](#_bookmark0)).

The training time of ResNet-101 is 92.5 minutes in our implementation using 256 Tesla P100 GPUs and a mini- batch size of 8k. We believe this is a compelling result if the speed-accuracy tradeoff of ResNet-101 is preferred.

**ImageNet-5k.** Observing the sharp increase in validation error between minibatch sizes of 8k and 16k on ImageNet- 1k (Figure [1](#_bookmark0)), a natural question is if the position of this ‘elbow’ in the error curve is a function of dataset infor- mation content. To investigate this question, we adopt the ImageNet-5k dataset suggested by Xie *et al*. [[39](#_bookmark76)] that extends ImageNet-1k to 6.8 million images (roughly 5 larger) by adding 4k additional categories from ImageNet- 22k [[33](#_bookmark70)]. We evaluate the 1k-way classification error on the original ImageNet-1k validation set as in [[39](#_bookmark76)].

×

The minibatch size *vs*. validation error curve for ImageNet-5k is shown in Figure [6](#_bookmark32). Qualitatively, the curve

20

256 512 1k 2k 4k 8k 16k 32k 64k

mini-batch size

Figure 6. **ImageNet-5k** top-1 validation error *vs*. minibatch size with a fixed 90 epoch training schedule. The curve is qualitatively similar to results on ImageNet-1k (Figure [1](#_bookmark0)) showing that a 5 increase in training data does not lead to a significant change in the maximum effective minibatch size.

*×*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *kn* | ImageNet  *η* | pre-training  top-1 error (%) | COCO  box AP (%) mask AP (%) | |
| 256 | 0.1 | 23.60 *±*0.12 | 35.9 *±*0.1 | 33.9 *±*0.1 |
| 512 | 0.2 | 23.48 *±*0.09 | 35.8 *±*0.1 | 33.8 *±*0.2 |
| 1k | 0.4 | 23.53 *±*0.08 | 35.9 *±*0.2 | 33.9 *±*0.2 |
| 2k | 0.8 | 23.49 *±*0.11 | 35.9 *±*0.1 | 33.9 *±*0.1 |
| 4k | 1.6 | 23.56 *±*0.12 | 35.8 *±*0.1 | 33.8 *±*0.1 |
| 8k | 3.2 | 23.74 *±*0.09 | 35.8 *±*0.1 | 33.9 *±*0.2 |
| 16k | 6.4 | 24.79 *±*0.27 |  | |

35.1 *±*0.3 33.2 *±*0.3

1. **Transfer learning of large minibatch pre-training to Mask R-CNN.** Box and mask AP (on COCO minival) are nearly identical for ResNet- 50 models pre-trained with minibatches from 256 to 8k examples. With a minibatch pre-training size of 16k both ImageNet validation error *and* COCO AP deteriorate. This indicates that as long as ImageNet error is matched, large minibatches do not degrade transfer learning performance.

# GPUs 1

2

4

8

*kn*

2

4

8

16

*η ·* 1000

2.5

5.0

10.0

20.0

iterations box AP (%) mask AP (%) 1,280,000 35.7 33.6

640,000 35.7 33.7

320,000 35.7 33.5

160,000 35.6 33.6

1. **Linear learning rate scaling applied to Mask R-CNN.** Using the sin- gle ResNet-50 model from [[16](#_bookmark53)] (thus no std is reported), we train Mask R-CNN using using from 1 to 8 GPUs following the linear learning rate scaling rule. Box and mask AP are nearly identical across all configurations showing the successful generalization of the rule beyond classification.

Table 3. **Object detection on COCO with Mask R-CNN [**[**14**](#_bookmark51)**].**

is very similar to the ImageNet-1k curve, showing that for practitioners it is unlikely that even a 5 increase in dataset size will automatically lead to a meaningful increase in use- able minibatch size. Quantitatively, using an 8k minibatch increases the validation error by 0.26% from 25.83% for a 256 minibatch to 26.09%. An understanding of the precise relationship between generalization error, minibatch size, and dataset information content is open for future work.

×

## Generalization to Detection and Segmentation

A low error rate on ImageNet is not typically an end goal. Instead, the utility of ImageNet training lies in learn-

0.3

time per iteration (secs)

0.28

0.26

0.24

0.22

16

32k

ideal

actual

time per epoch (mins)

8 16k

images / second

4 8k

2 4k

1 2k

0.2 0.5

256 512 1k 2k 4k 8k 11k

mini-batch size

Figure 7. **Distributed synchronous SGD timing.** Time per itera- tion (seconds) and time per ImageNet epoch (minutes) for training with different minibatch sizes. The baseline (*kn* = 256) uses 8 GPUs in a single server , while all other training runs distribute training over (*kn/*256) server. With 352 GPUs (44 servers) our implementation completes one pass over all *∼*1.28 million Ima- geNet training images in about 30 seconds.

ing good features that transfer, or generalize well, to re- lated tasks. A question of key importance is if the features learned with large minibatches generalize as well as the fea- tures learned with small minibatches?

To test this, we adopt the object detection and in- stance segmentation tasks on COCO [[27](#_bookmark64)] as these advanced perception tasks benefit substantially from ImageNet pre- training [[10](#_bookmark47)]. We use the recently developed Mask R-CNN

[[14](#_bookmark51)] system that is capable of learning to detect and segment object instances. We follow all of the hyper-parameter set- tings used in [[14](#_bookmark51)] and only change the ResNet-50 model used to initialize Mask R-CNN training. We train Mask R- CNN on the COCO trainval35k split and report results on the 5k image minival split used in [[14](#_bookmark51)].

It is interesting to note that the concept of minibatch size in Mask R-CNN is different from the classification setting. As an extension of the *image-centric* Fast/Faster R-CNN [[9](#_bookmark46), [31](#_bookmark68)], Mask R-CNN exhibits *different minibatch sizes for different layers*: the network backbone uses two images (per GPU), but each image contributes 512 Regions- of-Interest for computing classification (multinomial cross- entropy), bounding-box regression (smooth-L1/Huber), and pixel-wise mask (28 28 binomial cross-entropy) losses. This diverse set of minibatch sizes and loss functions pro- vides a good test case to the robustness of our approach.

×

**Transfer learning from large minibatch pre-training.** To test how large minibatch *pre-training* effects Mask R- CNN, we take ResNet-50 models trained on ImageNet-1k with 256 to 16k minibatches and use them to initialize Mask R-CNN training. For each minibatch size we pre-train 5 models and then train Mask R-CNN using all 5 models on COCO (35 models total). We report the mean box and mask APs, averaged over the 5 trials, in Table [3a](#_bookmark33). The results show that as long as ImageNet validation error is kept low, which is true up to 8k batch size, generalization to object de-

8 16 32 64 128 256 352

# GPUs

Figure 8. **Distributed synchronous SGD throughput.** The small overhead when moving from a single server with 8 GPUs to multi- server distributed training (Figure [7](#_bookmark36), blue curve) results in linear throughput scaling that is marginally below ideal scaling (*∼*90% efficiency). Most of the allreduce communication time is hid- den by pipelining allreduce operations with gradient computation. Moreover, this is achieved with commodity Ethernet hardware.

tection matches the AP of the small minibatch baseline. We emphasize that we observed *no generalization issues* when transferring across datasets (from ImageNet to COCO) and across tasks (from classification to detection/segmentation) using models trained with large minibatches.

**Linear scaling rule applied to Mask R-CNN.** We also show evidence of the generality of the linear scaling rule us- ing Mask R-CNN. In fact, this rule was already used with- out explicit discussion in [[16](#_bookmark53)] and was applied effectively as the default Mask R-CNN training scheme when using 8 GPUs. Table [3b](#_bookmark34) provides experimental results showing that when training with 1, 2, 4, or 8 GPUs the linear learning rate rule results in constant box and mask AP. For these experi- ments, we initialize Mask R-CNN from the released MSRA ResNet-50 model, as was done in [[14](#_bookmark51)].

## Run Time

Figure [7](#_bookmark36) shows two visualizations of the run time char- acteristics of our system. The blue curve is the time per iteration as minibatch size varies from 256 to 11264 (11k). Notably this curve is relatively flat and the time per itera- tion increases only 12% while scaling the minibatch size by

44 . Visualized another way, the orange curve shows the approximately linear decrease in time per epoch from over 16 minutes to just 30 seconds. Run time performance can also be viewed in terms of throughput (images / second), as shown in Figure [8](#_bookmark37). Relative to a perfectly efficient extrapo- lation of the 8 GPU baseline, our implementation achieves

×

*∼*90% scaling efficiency.

**Acknowledgements.** We would like to thank Leon Bottou for helpful discussions on theoretical background, Jerry Pan and Christian Puhrsch for discussions on efficient data loading, An- drew Dye for help with debugging distributed training, and Kevin Lee, Brian Dodds, Jia Ning, Koh Yew Thoon, Micah Harris, and John Volk for Big Basin and hardware support.

# References

1. J. Bagga, H. Morsy, and Z. Yao. Opening designs for 6-pack and Wedge 100. [https:](https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100)

[//code.facebook.com/posts/203733993317833/](https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100) [opening-designs-for-6-pack-and-wedge-100](https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100), 2016.

1. M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communica- tion library (intercom). In *Scalable High-Performance Com-* *puting Conference*, 1994.
2. L. Bottou. Curiously fast convergence of some stochastic gradient descent algorithms. Unpublished open problem of- fered to the attendance of the SLDS 2009 conference, 2009.
3. L. Bottou, F. E. Curtis, and J. Nocedal. Opt. methods for large-scale machine learning. *arXiv:1606.04838*, 2016.
4. J. Chen, X. Pan, R. Monga, S. Bengio, and R. Joze- fowicz. Revisiting Distributed Synchronous SGD. *arXiv:1604.00981*, 2016.
5. K. Chen and Q. Huo. Scalable training of deep learning ma- chines by incremental block training with intra-block par- allel optimization and blockwise model-update filtering. In *ICASSP*, 2016.
6. R. Collobert, J. Weston, L. Bottou, M. Karlen,

K. Kavukcuoglu, and P. Kuksa. Natural language pro- cessing (almost) from scratch. *JMLR*, 2011.

1. J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang,

E. Tzeng, and T. Darrell. Decaf: A deep convolutional acti- vation feature for generic visual recognition. In *ICML*, 2014.

1. R. Girshick. Fast R-CNN. In *ICCV*, 2015.
2. R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich fea- ture hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.
3. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
4. S. Gross and M. Wilber. Training and investigating Resid- ual Nets. [https://github.com/facebook/fb.](https://github.com/facebook/fb.resnet.torch) [resnet.torch](https://github.com/facebook/fb.resnet.torch), 2016.
5. M. Gu¨rbu¨zbalaban, A. Ozdaglar, and P. Parrilo. Why random reshuffling beats stochastic gradient descent. *arXiv:1510.08560*, 2015.
6. K. He, G. Gkioxari, P. Dolla´r, and R. Girshick. Mask R- CNN. *arXiv:1703.06870*, 2017.
7. K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.
8. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
9. G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed,

N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE* *Signal Processing Magazine*, 2012.

1. I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and

Y. Bengio. Quantized neural networks: Training neu- ral networks with low precision weights and activations. *arXiv:1510.08560*, 2016.

1. S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.
2. N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and

P. T. P. Tang. On large-batch training for deep learning: Gen- eralization gap and sharp minima. *ICLR*, 2017.

1. A. Krizhevsky. One weird trick for parallelizing convolu- tional neural networks. *arXiv:1404.5997*, 2014.
2. A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classi- fication with deep convolutional neural nets. In *NIPS*, 2012.
3. Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural compu-* *tation*, 1989.
4. K. Lee. Introducing Big Basin: Our next-generation AI hardware. [https://code.facebook.com/posts/](https://code.facebook.com/posts/1835166200089399/introducing-big-basin) [1835166200089399/introducing-big-basin](https://code.facebook.com/posts/1835166200089399/introducing-big-basin), 2017.
5. M. Li. *Scaling Distributed Machine Learning with System and Algorithm Co-design*. PhD thesis, Carnegie Mellon Uni- versity, 2017.
6. T.-Y. Lin, P. Dolla´r, R. Girshick, K. He, B. Hariharan, and

S. Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017.

1. T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ra- manan, P. Dolla´r, and C. L. Zitnick. Microsoft COCO: Com- mon objects in context. In *ECCV*. 2014.
2. J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.
3. Y. Nesterov. *Introductory lectures on convex optimization: A* *basic course*. Springer, 2004.
4. R. Rabenseifner. Optimization of collective reduction oper- ations. In *ICCS*. Springer, 2004.
5. S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: To- wards real-time object detection with region proposal net- works. In *NIPS*, 2015.
6. H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, 1951.
7. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh,

S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein,

A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 2015.

1. P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *ICLR*, 2014.
2. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
3. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed,

D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

1. R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective comm. operations in MPICH. *IJHPCA*, 2005.
2. Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi,

W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google’s neural machine translation system: Bridg- ing the gap between human and machine translation. *arXiv:1609.08144*, 2016.

1. S. Xie, R. Girshick, P. Dolla´r, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *CVPR*, 2017.
2. W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stol- cke, D. Yu, and G. Zweig. The Microsoft 2016 Conversa- tional Speech Recognition System. *arXiv:1609.03528*, 2016.
3. M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional neural networks. In *ECCV*, 2014.