Training machine learning models faster with Dask

Joesph Holt[‡], Scott Sievert[‡*]

Abstract—Machine learning (ML) relies on stochastic algorithms, all of which rely on gradient approximations with "batch size" examples. Growing the batch size as the optimization proceeds is a simple and usable method to reduce the training time, provided that the number of workers grows with the batch size. In this work, we provide a package that trains PyTorch models on Dask clusters, and can grow the batch size if desired. Our simulations indicate that for a particular model that uses GPUs for a popular image classification task, the training time can be reduced from about 120 minutes with standard SGD to 45 minutes with a variable batch size method.

Index Terms—machine learning, model training, distributed computation

Introduction

Training deep machine learning models takes a long time. For example, training a popular image classification model [RRSS19] to reasonable accuracy takes "around 17 hours" on Google servers.[1] Another example includes training an NLP model for 10 days on 8 high-end GPUs [RNSS18].[2] Notably, the number of floating point operations (FLOPs) required for "the largest AI training runs" doubles every 3.4 months.[3]

Model training is fundamentally an optimization problem: it tries to find a model $\widehat{\boldsymbol{w}}$ that minimizes a loss function F:

$$\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} F(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{w}; \boldsymbol{z}_i)$$

where there are n examples in the training set, and each example is represented by $\boldsymbol{z}_i$. For classification, $\boldsymbol{z}_i = (\boldsymbol{x}_i, y_i)$ for a label y_i and feature vector $\boldsymbol{x}_i$. The loss function F is the mean of the loss f over different examples. To compute this minimization for large scale machine learning, stochastic gradient descent (SGD) or a variant thereof is used [BCN18]. SGD is iterative, and the model update at each step k is computed via

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \frac{\gamma_k}{B_k}\sum_{i=1}^{B_k} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_{i_s})$$

where $\boldsymbol{g}$ is the gradient of the loss function f for some batch size $B_k \geq 1$, i_s is chosen uniformly at random and $\gamma_k > 0$ is the learning rate or step size. The objective function's gradient is approximated with B_k examples – the gradient approximation $\frac{1}{B_k}\sum_{i=1}^{B_k} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_{i_s})$ is an unbiased estimator of the loss function F's gradient. This computation is common in the vast majority of SGD variants, and is found in popular variants like Adam [KB14], RMSprop [ZSJ+19], Adagrad [DHS11], Adadelta [Zei12], and averaged SGD [PJ92]. Most variants make modifications to the learning rate γ_k [DHS11], [Zei12], [KB14], [ZSJ+19].

Increasing the batch size B_k will reduce the number of model updates while not requiring more FLOPs or gradient computations – both empirically [SKYL17] and theoretically [Sie20]. Typically, the number of FLOPs controls the training time because training is performed with a single processor. At first, fewer model updates seems like an internal benefit that doesn't affect training time.

The benefit comes when training with multiple machines, aka a distributed system. Notably, the time required to complete a single model update is (nearly) agnostic to the batch size provided the number of workers in a distributed system grows with the batch size. In one experiment, the time to complete a model update grows by 13% despite the batch size growing by a factor of 44 [GDG+17, Sec. 5.5]. This acceleration has also been observed with an increasing batch size schedule [SKYL17, Sec. 5.4].

Contributions

We provide software to accelerate machine learning model training, at least with certain distributed systems. For acceleration, the distributed system must be capable of assigning a different number of workers according to a fixed schedule. Specifically, this work provides the following:

- A Python software package to train machine learning models. The implementation[4] provides a Scikit-learn API [BLB+13] to PyTorch models [PGM+19].
- Our software works on any cluster that is configured to work with Dask, many of which can change the number of workers on demand.[5]
- Extensive experiments to illustrate that our software can accelerate model training in terms of wall-clock time when an appropriate Dask cluster is used.

A key component of our software is that the number of workers grows with the batch size. Then, the model update time is agnostic to the batch size provided that communication is instantaneous. This has been shown empirically: Goyal et al. grow the batch

‡ *University of Wisconsin–Madison*
∗ *Corresponding author: stsievert@wisc.com*

1. Specifically, a ResNet-50 model on the ImageNet database using a Google Tensor Proceesing Unit (TPU) (github.com/tensorflow/tpu/.../resnet/README.md).

2. See OpenAI's blog post "Improving Language Understanding with Unsupervised Learning."

3. See OpenAI's blog post "AI and Compute."

4. https://github.com/stsievert/adadamp

5. Including the default usage (through LocalCluster), supercomputers (through Dask Job-Queue), YARN/Hadoop clusters (through Dask Yarn) and Kubernetes clusters (through Dask Kubernetes).

size (and the number of workers with it) by a factor of 44 but the time for a single model update only increases by a factor of 1.13 [GDG+17, Sec. 5.5].

Now, let's cover related work to gain understanding of why variable batch sizes provide a benefit in a distributed system. Then, let's cover the details of our software before presenting simulations. These simulations confirm that model training can be accelerated if the number of workers grows with the batch size. Methods to workaround limitations on the number of workers will be presented.

Related work

The data flow for distributed model training involves distributing the computation of the gradient estimate, $\frac{1}{B}\sum_{i=1}^{B} \boldsymbol{g}(\boldsymbol{w}_k; \boldsymbol{z}_i)$. Typically, each worker computes the gradients for B/P examples when there is a batch size of B and P machines. Then, the average of these gradients is taken and the model is updated.[6]

Clearly, Amdahl's law is relevant because there are diminishing returns as the number of workers P is increased [GVY+18]. This as referred to as "strong scaling" because the batch size is fixed and the number of workers is treated as an internal detail. By contrast, growing the amount of data with the number of workers is known as "weak scaling." Of course, relevant experiments show that weak scaling exhibits better scaling than strong scaling [QST17].

Constant batch sizes

To circumvent Amdahl's law, a common technique is to increase the batch size [ZLN+19] alongside the learning rate [JAGG20]. Using moderately large batch sizes yields high quality results more quickly and, in practice, requires no more computation than small batch sizes, both empirically [GDG+17] and theoretically [YPL+18].

There are many methods to choose the best constant batch size (e.g., [GGS19], [KSL+20]). Some methods are data dependent [YPL+18], and others depend on the model complexity. In particular, one method uses hardware topology (e.g., network bandwidth) in a distributed system [PKK+19].

Large constant batch sizes present generalization challenges [GDG+17]. The generalization error is hypothesized to come from "sharp" minima, strongly influenced by the learning rate and noise in the gradient estimate [KMN+16]. To match performance on the training dataset, careful thought must be given to hyperparameter selection [GDG+17, Sec. 3 and 5.2]. In fact, this has motivated algorithms specifically designed for large constant batch sizes and distributed systems [JAGG20], [JSH+18], [YGG17].

Increasing the batch size

Model quality greatly influences the amount of information in the gradient – which influences the batch size [Sie20]. For example, if models are poorly initialized, then using a large batch size has no benefit: the gradient—or direction to the optimal model—for each example will produce very similar numbers. An illustration is given in Figure 1.

Various methods to *adaptively* change the batch size based on model performance have been proposed [Sie20], [DYJG16], [BRH17], [BCNW12]. Of course, these methods are adaptive so computing the batch size requires computation (though there are workarounds [Sie20], [BRH17]).

Convergence results have been given for adaptive batch sizes [Sie20], [BCN18], [ZYF18]. Increasing the batch size is a provably good measure that requires far fewer model updates and no more computation than standard SGD for strongly convex functions [BCN18, Ch. 5], and all function classes if the batch size is provided by an oracle [Sie20]. Convergence proofs have also been given for the *passively* increasing the batch size, both for strongly convex functions [BCN18, Ch. 5] and for non-convex functions [ZYF18]. Both of these methods require fewer model updates than SGD *and* do not increase the number of gradient computations.

Notably, a geometric batch size increase schedule has shown great empirical performance in image classification [SKYL17]. Specifically, the number of model updates required to finish training decreased by a factor of 2.2 over standard SGD [SKYL17]. Smith et al. make an observation that increasing the batch size and decreasing the learning rate both decay the optimization's "noise scale" (or variance of the model update), which has connections to simulated annealing [SKYL17]. This motivates increasing the batch size by the same factor the learning rate decays [SKYL17].

Both growing the batch size and using large constant batch sizes should require the same number of floating point operations as using standard SGD with small batch sizes to reach a particular training loss (respectively [Sie20], [BCN18] and [JAGG20], [YLR+19], [YPL+18]). Some proof techniques suggest that variable batch size methods mirror gradient descent [Sie20], [KNS16], so correspondingly, the implementations do not require much additional hyperparameter tuning [SKYL17].

Distributed training with Dask

We have written "AdaDamp," a software package to to train a PyTorch model with a Scikit-learn API on any Dask cluster.[7] It supports the use of constant or variable batch sizes, which fits nicely with Dask's ability to change the number of workers.[8] In this section, we will walk through the basic architecture of our software and an example usage. We will defer showing the primary benefit of our software to the experimental results.

Architecture

Our software uses a centralized synchronous parameter server and controls the data flow of the optimization with Dask (and does not rely on PyTorch's distributed support). Specifically, the following happen on every model update:

1) The master node broadcasts the model to every worker.
2) The workers calculate the gradients.
3) The workers communicate the gradients back to the master.
4) The master performs a model update with the aggregated gradients.

We use Dask to implement this data flow, which adds some overhead.[9] AdaDamp supports static batch sizes; however, there is little incentive to use AdaDamp with a static batch sizes: the

6. Related but tangential methods include methods to efficiently communicate the gradient estimates [AGL+17], [GTAZ18], [WSL+18].

7. While our software works with a constant batch size, the native implementations work with constant batch sizes and very likely have less overhead (e.g., PyTorch Distributed [LZV+20]).

8. https://github.com/stsievert/adadamp

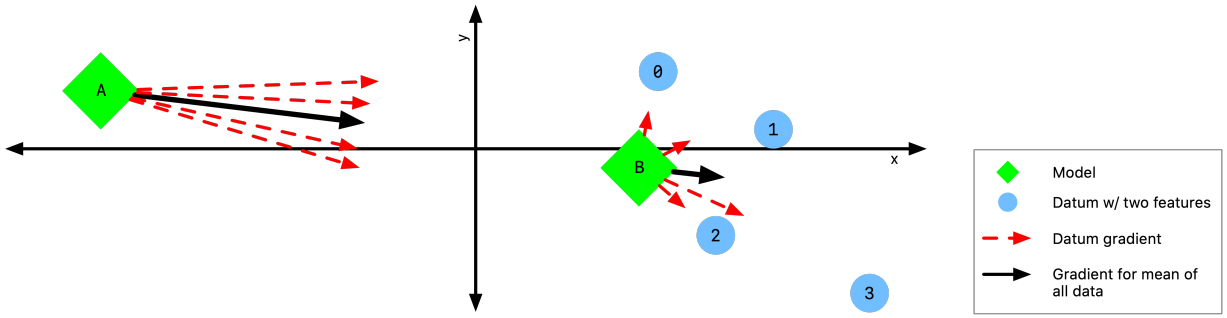

Fig. 1: *An illustration of why the batch size should increase. Here, let's find a model* $\mathbf{w} = [w_x, w_y]$ *that minimizes the function* $f(w_x, w_y) = \sum_{i=0}^{3}(w_x - x_i)^2 + (w_y - y_i)^2$ *where* x_i *and* y_i *are the x and y coordinates of each datum. When closer to the optimum at model A, the gradients are more "diverse," so the magnitude and orientation of each datum's gradient varies more [YPL+18].*

native solution has PyTorch less overhead [LZV+20], and already has a Dask wrapper.[10]

The key component of AdaDamp is that the number of workers grows with the batch size. Then, the model update time is agnostic to the batch size (provided communication is instantaneous). This has been shown empirically: Goyal et al. grow the batch size (and the number of workers with it) by a factor of 44 but the time for a single model update only increases by a factor of 1.13 [GDG+17, Sec. 5.5].

Example usage

First, let's create a standard PyTorch model. This is a simple definition; a more complicated model or one that uses GPUs can easily be substituted.

```
import torch.nn as nn
import torch.nn.functional as F

class HiddenLayer(nn.Module):
    def __init__(self, features=4, hidden=2, out=1):
        super().__init__()
        self.hidden = nn.Linear(features, hidden)
        self.out = nn.Linear(hidden, out)

    def forward(self, x, *args, **kwargs):
        return self.out(F.relu(self.hidden(x)))
```

Now, let's create our optimizer:

```
from adadamp import DaskRegressor
import torch.optim as optim

est = DaskRegressor(
    module=HiddenLayer, module__features=10,
    optimizer=optim.Adadelta,
    optimizer__weight_decay=1e-7,
    max_epochs=10
)
```

So far, a PyTorch model and optimizer have been specified. As per the Scikit-learn API, we specify parameters for the model/optimizer with double underscores, so in our example `HiddenLayer(features=10)` will be created. We can set the batch size increase parameters at initialization if desired, or inside `set_params`.

```
from adadamp.dampers import GeoDamp
est.set_params(
    batch_size=GeoDamp, batch_size__delay=60,
    batch_size__factor=5)
```

This will increase the batch size by a factor of 5 every 60 epochs, which is used in the experiments. Now, we can train:

```
from sklearn.datasets import make_regression
X, y = make_regression(n_features=10)
X = torch.from_numpy(X.astype("float32"))
y = torch.from_numpy(y.astype("float32")).reshape(-1, 1)

est.fit(X, y)
```

Experiments

In this section, we present two sets of experiments.[11] Both experiments will use the same setup, a Wide-ResNet model in a "16-4" architecture [ZK16] to perform image classification on the CIFAR10 dataset [KH09]. This is a deep learning model with about 2.75 million weights that requires a GPU to train.[12] The experiments will provide evidence for the following points:

1) Increasing the batch size reduces the number of model updates.
2) The time required for model training is roughly proportional to the number of model updates (presuming the distributed system is configured correctly).

To provide evidence for these points, let's run one set of experiments that varies the batch size increase schedule. These experiments will mirror the experiments by Smith et al. [SKYL17]. Additionally, let's ensure that our software accelerates model training as the number of GPUs increase.

We train each batch size increase schedule once, and then write the historical performance to disk. This reduces the need for many GPUs, and allows us to simulate different networks and highlight the performance of Dask. That means that in our simulations, we simulate model training by having the computer sleep for an appropriate and realistic amount of time.

9. An opportunity for future work.
10. https://github.com/saturncloud/dask-pytorch-ddp
11. Full detail on these experiments can be found at https://github.com/stsievert/adadamp-experiments
12. Specifically, we used a NVIDIA T4 GPU with an Amazon `g4dn.xlarge` instance. Training consumes 2.2GB of GPU memory with a batch size of 32, and 5.5GB with a batch size of 256.

Batch size increase

To illustrate the primary benefit of our software, let's perform several trainings that require a different number of model updates. These experiments explicitly mirror the experiments by Smith et al. [SKYL17, Sec. 5.1], which helps reduce the parameter tuning.

Largely, the same hyperparameters are used. These experiments only differ in the choice of batch size and learning rate, as shown in Figure 2. As in the Smith et al. experiments, every optimizer uses Nesterov momentum [Nes98] and the same momentum (0.9) and weight decay ($0.5 \cdot 10^{-3}$). They start with the same initial learning rate (0.05),[13] and either the learning rate is decreased or the batch size increases by a specified factor (5) at particular intervals (epochs 60, 120 and 180). This means that the variance of the model update is reduced by a constant factor at each update.

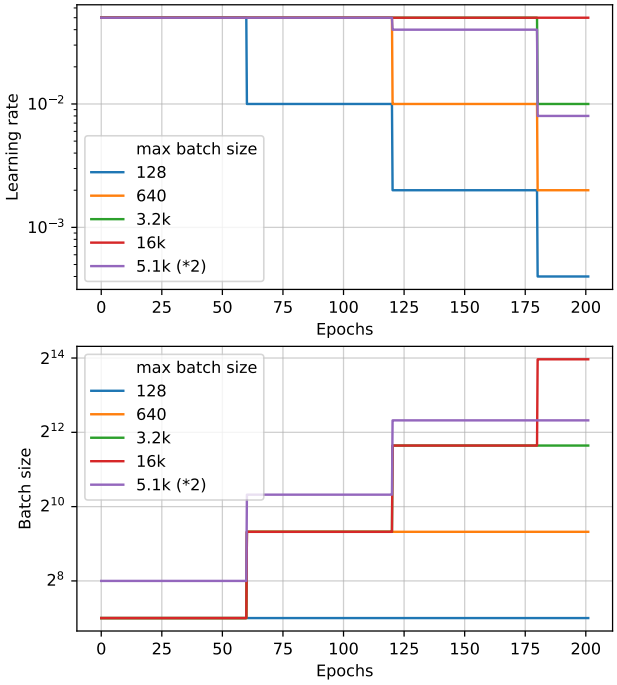

Fig. 2: *The learning rate and batch size decrease/increase schedules for various optimizers. After the maximum batch size is reached, the learning rate decays. A postfix of "(*2)" means the initial batch size twice as large (256 instead of 128)*

These different decay schedules exhibit the same performance in terms of number of epochs, which is proportional to the number of FLOPs, as shown in Figure 3. The number of FLOPs is (approximately) to the cost, at least on Amazon EC2 where the cost to rent a server tends to be proportional to the number of GPUs.

Importantly, this work focuses on increasing the number of workers with the batch size – the effect of which is hidden in Figure 3. However, the fact that the performance does not change with different schedules means that choosing a different batch size increase schedule will not require more wall-clock time if only a single worker is available. Combined with the hyperparameter similarity between the different schedules, this reduces deployment and debugging concerns.

13. These are the same as Smith et al. [SKYL17] with the exception of learning rate (which had to be reduced by a factor of 2).

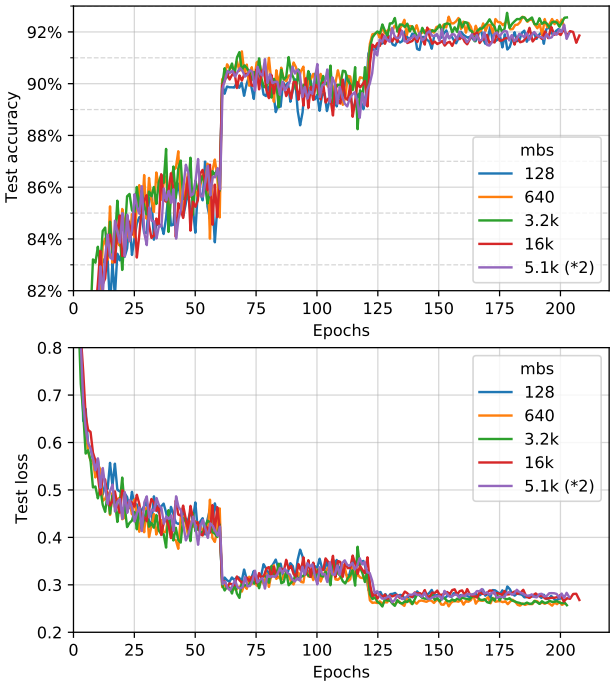

Fig. 3: *The performance of the LR/BR schedules in Figure 2, plotted with epochs—or passes through the dataset—on the x-axis.*

Maximum batch size	Model updates	Training time (min)	Max. workers
5.1k (*2)	14,960	69.87	40
3.2k	29,480	107.17	25
16k	29,240	107.49	125
640	34,520	116.86	5
128	78,200	200.19	1

TABLE 1: *A summary of the simulations in Figures 3 and 4. All training require approximately 200 epochs, so they all require the same number of FLOPs.*

If the number of workers grows with the batch size, then the number of model updates is relevant to the wall-clock time. Figure 4 shows the number of model updates and wall-clock time required to reach a model of a particular test accuracy. Of course, there is some overhead to our current framework, which is why the number of model updates does not exactly correlate with the wall-clock time required to complete training. In summary, the time required to complete training is shown in Table 1.

Future work

Architecture

Fundamentally, the model weights can be either be held on a master node (centralized), or on every node (decentralized). Respectively, these storage architectures typically use point-to-point communication or an "all-reduce" communication. Both centralized [LAP+14], [ABC+16] and decentralized [LZV+20], [SDB18] communication architectures are common.

Future work is to avoid the overhead introduced by manually having Dask control the model update workflow. With any synchronous centralized system, the time required for any one model update is composed of the time required for the following tasks:

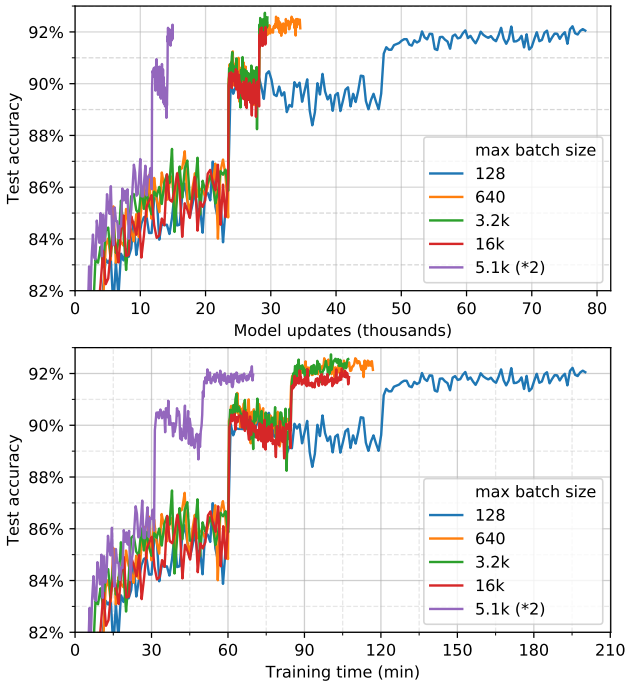

Fig. 4: *The same simulations as in Figure 3, but plotted with the number of model updates and wall-clock time plotted on the x-axis (the loss obeys a similar behavior as illustrated in the Appendix).*

1) Broadcasting the model from the master node to all workers
2) Finishing gradient computation on all workers.
3) Communicating gradients back to master node.
4) Various overhead tasks (e.g., serialization, worker scheduling, etc).
5) Computing the model update after all gradients are computed & gathered.

Items (1), (3) and (4) are a large concern in our implementation. Decentralized communication has the advantage of eliminating items (1) and (4), and mitigates (3) with a smarter communication strategy (all-reduce vs. point-to-point). Item (2) is still a concern with straggler nodes [DCM+12], but recent work has achieved "near-linear scalability with 256 GPUs" in a homogeneous computing environment [LZV+20]. Items (2) and (5) can be avoided with asynchronous methods (e.g., [RRWN11], [ZHA16]).

That is, most of the concerns in our implementation will be resolved with a distributed communication strategy. The PyTorch distributed communication package uses a synchronous decentralized strategy, so the model is communicated to each worker and gradients are sent between workers with an all-reduce scheme [LZV+20]. It has some machine learning specific features to reduce the communication time, including performing both computation and communication concurrently as layer gradients become available [LZV+20, Sec. 3.2.3].

The software library dask-pytorch-ddp[14] allows use of the PyTorch decentralized communications [LZV+20] with Dask clusters, and is a thin wrapper around PyTorch's distributed communication package. Future work will likely involve ensuring training can efficiently use a variable number of workers.

14. https://github.com/saturncloud/dask-pytorch-ddp

Maximum batch size	Centralized	Decentralized (moderate)	Decentralized (high)
5.1k (*2)	69.9	45.1	43.5
3.2k	107.2	67.7	65.5
16k	107.5	67.7	65.7
640	116.9	73.6	71.8
128	200.2	121.7	121.5

TABLE 2: *Simulations that indicate how the training time (in minutes) will change under different architectures and networks. The "centralized" architecture is the currently implemented architecture, and has the same numbers as "training time" in Table 4.*

Simulations

We have simulated the expected gain from the work of enabling decentralized communication with two networks that use a decentralized all-reduce strategy:

- `decentralized-medium` It assumes an a network with inter-worker bandwidth of 54Gb/s and a latency of 0.05μs.
- `centralized` uses a centralized communication strategy (as implemented) and the same network as `decentralized-medium`.
- `decentralized-high` has the same network as `decentralized-medium` but has an inter-worker bandwidth of 800Gb/s and a latency of 0.025μs.

To provide baseline performance, we also show the results with the current implementation:

- `centralized` uses the same network as `decentralized-medium` but with the centralized communication scheme that is currently implemented.

`decentralized-medium` is most applicable for clusters that have decent bandwidth between nodes. It's also applicable to for certain cases when Amazon EC2 is used with one GPU per worker,[15] or workers have a very moderate Infiniband setup.[16] `decentralized-high` is a simulation of the network used by the PyTorch developers to illustrate their distributed communication [LZV+20]. We have run simulations to illustrate the effects of these networks. Of course, changing the underlying networks does not affect the number of epochs or model updates, so Figures 3 and 4 also apply here.

A summary of how different networks affect training time is shown in Table 2. We show the training time for a particular network (`decentralized-moderate`) in Figure 6; `decentralized-high` shows similar performance as illustrated in Table 2. A visualization of 2 is shown in Figure 5. This shows how network quality affects the performance of different optimization methods in Figure 6. Clearly, the optimization method (and the maximum number of workers) is more important than the network.

Finally, let's show how the number of Dask workers affects the time required to complete a single epoch with a constant

15. 50Gb/s and 25Gb/s networks can be obtained with `g4dn.8xlarge` and `g4dn.xlarge` instances respectively. `g4dn.xlarge` machines have 1 GPU each and are the least expensive for a fixed number of FLOPs on the GPU.

16. A 2011 Infiniband setup with 4 links (https://en.wikipedia.org/wiki/InfiniBand#Performance)

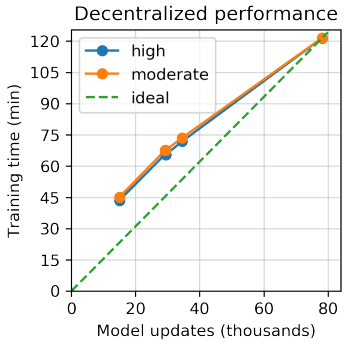

Fig. 5: *A single point represents one run in Figure 6. The point with about 80k model updates represents a single worker, so there's no overhead in this decentralized simulation. Different network qualities are shown with different colors, and the "ideal" line is as if every model update is agnostic to batch size.*

batch size. This simulation will use the `decentralized-high` network and has the advantage of removing any overhead. The results in Figure 7 show that the speedups start saturating around 128 examples/worker for the model used with a batch size of 512. Larger batch sizes will likely mirror this performance – computation is bottleneck with this model/dataset/hardware.

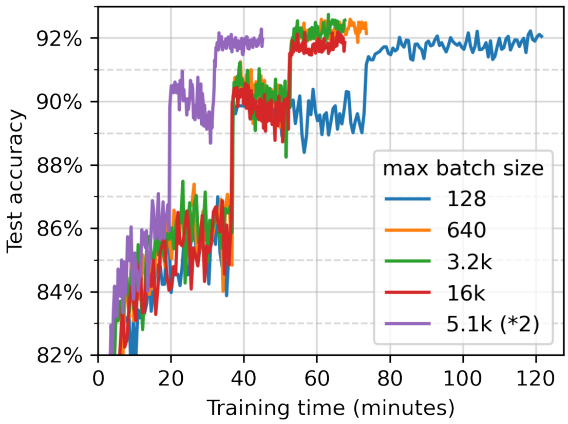

Fig. 6: *The training time required for different optimizers under the* `decentralized-moderate` *network.*

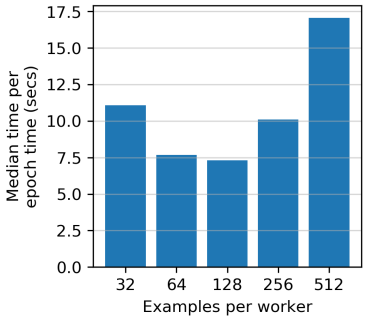

Fig. 7: *The median time to complete a pass through the training set with a batch size of 512. As expected, the speedups diminish when there is little computation and much communication (say with 32 examples per worker).*

Conclusion

In this work, we have provided a package to train PyTorch ML models with Dask cluster. This package reduces the amount of time required to train a model with the current centralized setup. However, it can be further accelerated by integration with PyTorch's distributed communication package as illustrated by extensive simulations. For a particular model, only 45 minutes is required for training – an improvement over the 120 minutes required with standard SGD.

APPENDIX

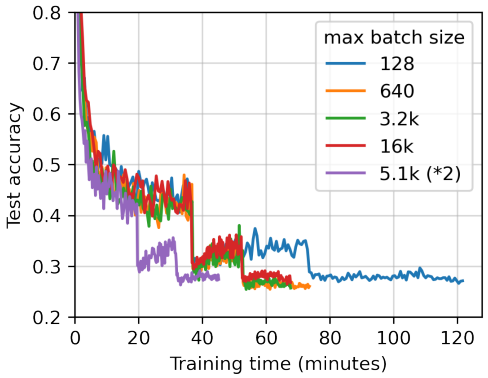

Fig. 8: *The training time required for different optimizers under the* `decentralized-moderate` *network.*

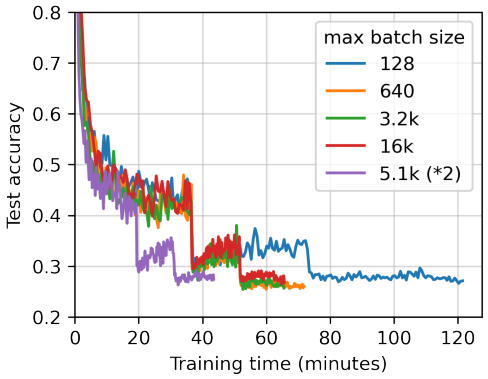

Fig. 9: *The training time required for different optimizers under the* `decentralized-high` *network.*

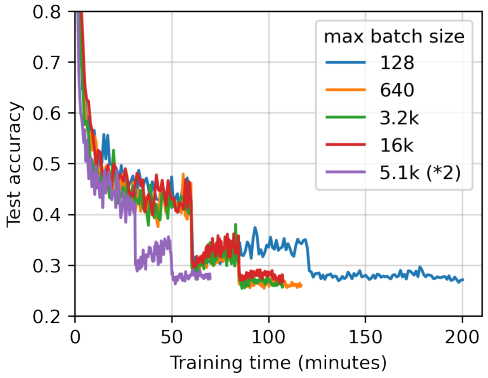

Fig. 10: *The training time required for different optimizers under the* `centralized` *network.*

REFERENCES

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/6c340f25839e6acdc73414517203f5f0-Paper.pdf.

[BCN18] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60:223–223, 2018. doi:10.1137/16M1080173.

[BCNW12] Richard H Byrd, Gillian M Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012. doi:10.1007/s10107-012-0572-5.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[BRH17] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. In *33rd Conference on Uncertainty in Artificial Intelligence (UAI 2017)*, pages 675–684. Curran Associates, Inc., 2017. URL: http://auai.org/uai2017/proceedings/papers/141.pdf.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. 25, 2012. URL: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL: http://jmlr.org/papers/v12/duchi11a.html.

[DYJG16] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Big Batch SGD: Automated inference using adaptive batch sizes. *arXiv preprint arXiv:1610.05792*, 2016.

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[GGS19] Nidham Gazagnadou, Robert Gower, and Joseph Salmon. Optimal mini-batch and step sizes for SAGA. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2142–2150. PMLR, 09–15 Jun 2019. URL: http://proceedings.mlr.press/v97/gazagnadou19a.html.

[GTAZ18] Demjan Grubic, Leo K Tam, Dan Alistarh, and Ce Zhang. Synchronous multi-gpu deep learning with low-precision communication: An experimental study. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 145–156. OpenProceedings, 2018. doi:10.3929/ethz-b-000319485.

[GVY+18] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

[JAGG20] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A user-friendly algorithm for distributed training. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4911–4920. PMLR, 13–18 Jul 2020. URL: http://proceedings.mlr.press/v119/johnson20a.html.

[JSH+18] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong

Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, and Liwei Yu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[KNS16] Hamed Karimi, Julie Nutini, and Mark Schmidt. Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, page 795–795, 2016. doi:10.1007/978-3-319-46128-1_50.

[KSL+20] Ahmed Khaled, Othmane Sebbouh, Nicolas Loizou, Robert M Gower, and Peter Richtárik. Unified analysis of stochastic gradient methods for composite convex and smooth optimization. *arXiv preprint arXiv:2006.11573*, 2020.

[LAP+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[LZV+20] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020. URL: https://doi.org/10.14778/3415478.3415530, doi:10.14778/3415478.3415530.

[Nes98] Yurii Nesterov. *Introductory lectures on convex programming volume i: Basic course*, volume 3. 1998.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[PJ92] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992. doi:10.1137/0330046.

[PKK+19] Michael P Perrone, Haidar Khan, Changhoan Kim, Anastasios Kyrillidis, Jerry Quinn, and Valentina Salapura. Optimal mini-batch size selection for fast gradient descent. *arXiv preprint arXiv:1911.06459*, 2019.

[QST17] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR (Poster)*, 2017. URL: https://talwalkarlab.github.io/paleo/.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[RRSS19] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 09–15 Jun 2019. URL: http://proceedings.mlr.press/v97/recht19a.html.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

[SDB18] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[Sie20] Scott Sievert. Improving the convergence of sgd through adaptive batch sizes, 2020.

[SKYL17] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc Vgra Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[WSL+18] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/33b3214d792caf311e1f00fd22b392c5-Paper.pdf.

[YGG17] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[YLR+19] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[YPL+18] Dong Yin, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. Gradient diversity: a key ingredient for scalable distributed learning. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1998–2007. PMLR, 09–11 Apr 2018. URL: http://proceedings.mlr.press/v84/yin18a.html.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[ZHA16] Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. pages 629–638, 2016. doi:10.1109/ICDM.2016.0074.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[ZLN+19] Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/e0eacd983971634327ae1819ea8b6214-Paper.pdf.

[ZSJ+19] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of adam and rmsprop. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11127–11135, 2019. doi:10.1109/cvpr.2019.01138.

[ZYF18] Pan Zhou, Xiaotong Yuan, and Jiashi Feng. New insight into hybrid stochastic gradient descent: Beyond with-replacement sampling and convexity. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/67e103b0761e60683e83c559be18d40c-Paper.pdf.