COMMUNICATION EFFICIENT DISTRIBUTED SGD

Drake Svoboda Anshul Aggarwal

ABSTRACT

We study communication efficient distributed SGD algorithms. Synchronized SGD requires significant network bandwidth for gradient exchange and is not robust to node slowdowns and network delays. Thus, many strategies have been proposed to mitigate both synchronization and network costs to speed-up training. We propose two new communication efficient algorithms: (1) *staggered EASGD* and (2) *significance compression*. Staggered EASGD is an optimization to elastic-averaging SGD that more efficiently utilizes available network bandwidth. Significance compression is a gradient sparcification method that compresses the size of gradients before transmission. We evaluate our methods on vision and NLP benchmarks and find 0.1-8.8× and 0.9-1.8× increase in training throughput for staggered EASGD and significance compression respectively when compared to synchronized SGD on commodity 1Gbps network links with minimal performance degradation. We additionally evaluate the performance of both the EASGD algorithm and *approximate-synchronous parallel* synchronization model on reinforcement learning workloads.

1 INTRODUCTION

Stochastic gradient descent (SGD) is the backbone of modern state-of-the-art supervised machine learning. Due to the increasing size of deep-neural network architectures and datasets, it has become imperative to distribute SGD across many nodes and parallelize gradient computation and aggregation. Synchronous SGD is widely used for distributed training and can reduce the computation time for the forward-backward passes and allow for increases in the amount of data processed per iteration. Although synchronous SGD can reduce computation time, it exposes SGD to node slowdowns and network delays. The additional synchronization and network costs of gradient exchange can dwarf computation savings. These problems are amplified in heterogeneous hardware or geo-distributed settings.

Our goal in this project is to investigate strategies for reducing the systems costs of distributed SGD. In particular, we focus on (1) elastic-averaging SGD (Zhang et al., 2015) and (2) approximate synchronous parallel (Hsieh et al., 2017). Elastic-averaging SGD (EASGD) is a communication efficient SGD algorithm that reduces the frequency with which parameter updates are communicated between the nodes. Approximate synchronous parallel (ASP) is a synchronization model that eliminates insignificant communication to reduce network costs. Inspired by these works, we additionally investigate two new techniques:

1. **Staggered EASGD.** We propose an optimization to EASGD that staggers the communication periods of each layer in a neural network. This amortizes the communication cost and eliminates network bursts thereby reducing network delays and speeding up training.
2. **Significance Compression.** We use the significance filter from ASP as a gradient compression in the synchronized all-reduce setting.

In section 3, we describe both staggered EASGD and significance compression. In section 4, we evaluate performance on vision and NLP workloads. Additionally, in section 4.1 we evaluate the performance of EASGD and ASP on several reinforcement learning workloads.

2 RELATED WORK

Two popular frameworks for data-parallel SGD are parameter server and all-reduce. In both schemes, each parallel worker computes gradients on a mini-batch of training examples for each training iteration; these gradients are aggregated and used to update a shared set of parameters. Synchronizing the parallel workers and aggregating the gradients is costly; training speed is limited by the slowest worker and gradient aggregation requires high network bandwidth. Much work has gone into reducing these costs (Wang & Joshi, 2019; Jayarajan et al., 2019; Ho et al., 2013; Li et al., 2014; Zhang et al., 2015; Hsieh et al., 2017; Alistarh et al., 2017; Lin et al., 2020).

Many works have propose flexible consistency models to speed things up (Ho et al., 2013; Zhang et al., 2015; Hsieh et al., 2017; Alistarh et al., 2017; Lin et al., 2020). These algorithms modify the computation of SGD and incur an accuracy penalty as a result. Total asynchronous parallel (otherwise known as Hogwild! (Niu et al., 2011)) re-

moves synchronization barriers all together and allows each parallel worker to continue running with best effort communication and a stale or inaccurate view of the shared parameters (Chilimbi et al., 2014). Like Hogwild!, stale-synchronous parallel (SSP) (Ho et al., 2013) removes synchronization barriers but bounds the number of iterations the slowest worker may be behind the fastest worker. Similarly, approximate-synchronous parallel (ASP) (Hsieh et al., 2017) bounds the difference between the globally shared parameters and a worker's view of the parameters. Empirically, these synchronization schemes do not slow down convergence too much per iteration, but significantly increase throughput. Mitliagkas et al. show that asynchrony introduces an *implicit momentum* similar to the explicit momentum used by many optimizers; this may partially explain this phenomenon (Mitliagkas et al., 2016).

Other approaches simply limit the frequency with which the workers are synchronized and updates are aggregated. In federated learning, parallel workers train for multiple iterations before transmitting their updates to a central server which takes a *federated average* of the updates (McMahan et al., 2017). Like federated learning, elastic-averaging SGD (EASGD) (Zhang et al., 2015) allows each worker to locally take τ SGD steps before communicating it's updated parameters with a parameter server; the parameter server updates the global parameters by taking an *elastic average*. The asynchronous variant of EASGD further removes synchronization barriers and is proven to be stable and empirically gives good results (Zhang et al., 2015). AdaComm (Wang & Joshi, 2019) is a similar algorithm that only periodically averages gradients every τ training iterations; Wang & Joshi show that reducing τ over-time improves the convergence rate.

Gradient compression methods use either a lossless or lossy compression on the gradients before transmission (Renggli et al., 2019; Alistarh et al., 2017; Vogels et al., 2019). For example, QSGD (Alistarh et al., 2017) reduces the number of bits used to represent the gradients and PowerSGD (Vogels et al., 2019) communicates a low-rank approximation of the gradients. A subset of gradient compression methods known as *gradient sparcification* only select a sparse subset of gradient components with the highest magnitude to be transmitted (Alistarh et al., 2018; Lin et al., 2020). Gradients which are not transmitted are accumulated and eventually become large enough to be transmitted. Gradient sparcification is closely related to asynchronous SGD; gradient components which are filtered out are delayed and become *stale*. The staleness of these updates is implicitly bounded by the selection criteria. Using this observation, Alistarh et al. prove that gradient specification methods provide convergence guarantees for both convex and non-convex objectives (Alistarh et al., 2018).

Other approaches efficiently parallelize computation with transmission by scheduling network communication to optimally overlap with computation (Jayarajan et al., 2019; Peng et al., 2019). These approaches do not modify the computation of synchronized SGD and therefore do not incur an accuracy penalty, however, they often do not achieve the speed ups of other methods. Priority-based Parameter Propagation (P3) (Jayarajan et al., 2019) ensures that parameters that must be read first in the forward pass are prioritized for transmission. ByteScheduler (Peng et al., 2019) similarly prioritizes transmission of gradients in earlier layers. Additionally, the authors show that prioritization in this way achieves an optimal network schedule.

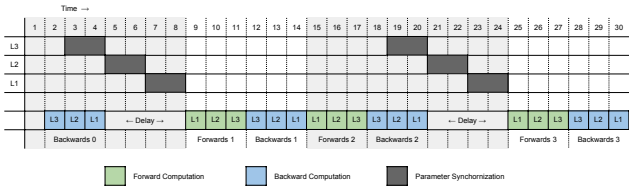
3 METHOD

In this section we describe both staggered EASGD and significance compression. Our implementations are available on github.[1]

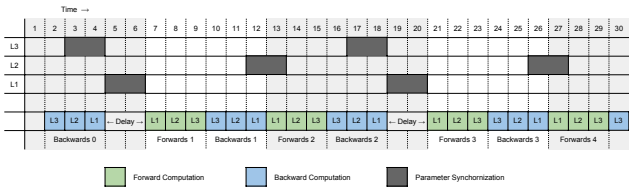
3.1 Staggered EASGD

In this section we describe an optimization to EASGD that more efficiently parallelizes communication with the forward and backward computations of DNNs. EASGD is a distributed SGD algorithm that reduces communication cost by limiting the frequency with which parameters are synchronized with a parameter server (Zhang et al., 2015). Algorithm 1 shows the algorithm for asynchronous EASGD. The frequency with which the parameters are synchronized is called the *communication period* and is defined by the hyper-parameter τ. Our key insight is that there is a network burst every τ iterations when the condition on line 4 of algorithm 1 is met; during other iterations, the network is completely unused. Thus, a network delay is caused every τ iterations and the network is under-utilized otherwise. We alleviate this problem by partitioning the parameters into chunks and staggering their communication periods. This amortizes the communication costs over the τ iterations that make up the communication period. In our case, we train DNNs and partition the network parameters by layer. Synchronizing the parameters of each layer is done in parallel to both the backwards and forwards passes of the previous layers. We call EASGD with our optimization *staggered EASGD*. Figure 1 shows a visualization of our optimization for a three layer network and a communication period of 2. A small speed-up is achieved in figure 1(b) since the transmission of layer 2 causes no delay and the transmission of layer 1 does not compete for network bandwidth with layer 2. In practice, this optimization should achieve the largest relative speed-up when transmitting the entire set of parameters exhausts the network bandwidth causing delay

[1] https://github.com/drakesvoboda/DistributedTrainingExperiments.



(a) EASGD synchronization



(b) Staggered EASGD synchronization

Figure 1. Parameter synchronization for a 3 layer network with (a) EASGD and (b) staggered EASGD. The communication interval is set to 2—that is, synchronization occurs every-other SGD iteration. For staggered EASGD, the communication interval of layer 2 is offset by one iteration.

as seen in time-steps 5-8 of figure 1(a). In cases where the number of parameters is low and network bandwidth is sufficient, we should not observe much speed-up. Note that staggered EASGD is not mathematically equivalent to standard EASGD. In section 4 we show that staggered EASGD gives empirically similar performance to standard EASGD on our tested workloads.

Algorithm 1: Asynchronous EASGD: Processing by worker p and the master

$\tilde{x}$ is initialized randomly, $x^p = \tilde{x}$, $t^p = 0$;
repeat
  $x \leftarrow x^p$;
  if τ *divides* t^p **then**
    $x^p \leftarrow x^p - \alpha(x - \tilde{x})$;
    $\tilde{x} \leftarrow \tilde{x} + \alpha(x - \tilde{x})$;
  end
  $x^p \leftarrow x^p - \eta G^p_{t^p}(x)$;
  $t^p \leftarrow t^p + 1$;
until *forever*;

We have implemented EASGD with a sharded parameter server using pytorch's RPC framework. The model's parameters are split by layer and each layer is further split into chunks of 100,000 parameters. These chunks are evenly distributed to each parameter server shard. Chunking each layer eliminates bottlenecks caused by a server shard being assigned a disproportionately large layer. Pytorch's `register_forward_hook` and `register_backward_hook` methods are used to execute arbitrary functions before the forward computation and after the backward computation of a layer. For each layer, we register a backwards hook that transmits the layers parameters to the parameter server every communication period and a forward hook that blocks until the communication with the parameter server is complete. Transmission is non-blocking and happens in parallel to the backwards and forward computation of earlier layers. The performance of staggered EASGD is evaluated in section 4.

3.2 Significance Compression

Drawing inspiration from the similarity between gradient sparcification and asynchronous SGD, we borrow the significance filter from ASP (Hsieh et al., 2017) and use it as a gradient compression. The significance filter is used to convert each gradient update to a sparse gradient by selecting components with large magnitude. The sparse gradient is then aggregated across the workers using all-reduce. This limits bandwidth since we are only sending the most important updates. The gradient components which are not transmitted are aggregated locally until eventually they are large enough in magnitude to be transmitted. Other gradient sparcification techniques (Lin et al., 2020) use only the gradient component's magnitude as a heuristic for significance; as in (Hsieh et al., 2017), we use the significance filter. The significance filter has two components, a significance function and a significance threshold. A gradient component is deemed significant if its significance is larger than the threshold. We use the gradient's magnitude relative to the current value ($|\frac{Update}{Value}|$) as the significance function. Algorithm 2 shows the algorithm for significance compression for a single worker. The variable acc^p_t is used to store the locally aggregated gradient; once the significance of this value passes the significance threshold thr, it is transmitted and reset to 0. The variable v_t stores the view of the model parameters; this view is the same for each worker. The function $G^p_t(\cdot)$ computes the gradient with respect to the model parameters v_t. Note that when $thr = 0$, this algorithm reduces to synchronized all-reduce SGD.

We have implemented significance compression using pytorch communication primitives. Pytorch's all-reduce primitive supports reduction of sparse tensors.

4 Experiments

We compare the performance of significance compression (SigComp), staggered EASGD, standard EASGD, and synchronized SGD as implemented in pytorch's `DistributedDataParallel` class (DDP). For both EASGD methods, we use the asynchronous variant as described by Zhang et al.. To evaluate the methods we train three models on two datasets: a 7 layer MLP with 3,465,514 trainable parameters and 2 layer CNN with 40,450 parame-

Algorithm 2: Significance Compression: Processing by worker p

for *t = 0,1,...* **do**
  $acc_t^p \leftarrow \epsilon_{t-1}^p + \eta G_t^p(v_{t-1})$;
  $g_t^p \leftarrow acc_t^p \odot (|acc_t^p|/|v_{t-1}^p| > thr)$;
  $\epsilon_t^p \leftarrow acc_t^p - g_t^p$;
  all-reduce $g_t^p : g_t \leftarrow \sum_{p=1}^N g_t^p/N$;
  $v_t \leftarrow v_{t-1} - g_t$;
end

ters, and a 1 layer LSTM with 5,668,781 parameters. The MLP and CNN are trained on the SVHN image classification dataset (Netzer et al., 2011) The LSTM is trained on the Wall Street Journal (WSJ) part-of-speech tagging dataset (Marcus et al., 1993). We train on a 3 node cluster of Intel Xeon E5-2660 V3 10 core CPUs; each node is connected via 1Gbps link. This bandwidth is on the low-end for network infrastructure in modern cloud services. We use a low bandwidth as it highlights the performance of our proposed methods. For both EASGD variants, we set the communication period to $\tau = 20$. For significance compression we set the significance threshold to $thr = 0.01$. Training is done using vanilla SGD with a learning rate of $1e-2$. For the SVHN benchmark we train for a total of 100,000 iterations, for the WSJ benchmark we train for 70,000 iterations. In both cases we use a batch size of of 16 examples and compute a validation accuracy every 10,000 iterations. For each benchmark, validation accuracy is plotted with respect to time in figure 3. The average throughput of each method are reported in figure 2.

Standard EASGD and staggered EASGD gave near identical performance in terms of accuracy. Both methods caused a drop in validation accuracy of roughly 1 and 4 percentage points compared to synchronized SGD on the SVHN and WSJ benchmarks respectively. For the MLP and LSTM experiments, staggered EASGD had a small throughput advantage of about 2% and 8% respectively; however, it had a marginal loss in throughput of roughly 0.5% for the small CNN model. This performance is expected as the CNN has significantly fewer parameters than the MLP and LSTM. Staggered EASGD should give better relative speedups when the forward and backward computations are fast relative to the time to transmit parameters and transmission of the entire set of parameters exhausts the network bandwidth; thus, we hypothesize that our optimization will give more significant performance improvements for larger models trained on the GPU. Our experiments validate this hypothesis since staggered EASGD achieved a throughput advantage for the larger MLP and LSTM models.

Significance compression gives a speed up over synchronized SGD on the MLP and LSTM models, however, it is much slower on the CNN benchmark. Gradient compression techniques require additional computation to compress the gradient; in the CNN experiment, the added cost of compressing the gradient is larger than the reduced communication cost. Significance compression caused roughly a 1 percentage point drop in accuracy on SVHN but caused no performance degradation on the WSJ benchmark.

4.1 Reinforcement Learning

We additionally evaluate the performance of EASGD and ASP on two reinforcement learning workloads. We have implemented the EASGD and ASP algorithms as agents in the RLLib library (Liang et al., 2018). RLLib is a reinforcement learning library built on top of Ray (Moritz et al., 2017). Our implementations are available on github.[2]

We compare our implementations of ASP and EASGD against existing implementations of PPO, A3C, and IMPALA.

The PPO algorithm (Schulman et al., 2017) was introduced by OpenAI in 2017 and quickly became one of the most popular RL methods usurping Deep-Q learning. It involves collecting a small batch of experiences by interacting with the environment and using that batch to update a decision-making policy. Once the policy is updated, the experiences are thrown away and a new batch is collected. RLLib's implementation of distributed PPO is a fully synchronized; each worker always maintains the most up to date version of the policy. Thus, PPO tends to have low throughput in distributed environments. Reinforcement learning algorithms tend to be very sample inefficient and in some cases require training on tens of millions of examples before convergence. Making matters worse, data collection requires computationally expensive environment simulation; thus, distributing the computation is imperative.

To speed things up, many algorithms have been proposed that use similar ideas to the asynchronous and communication efficient methods described previously. Asynchronous Advantage Actor Critic (A3C) (Wang et al., 2016) removes synchronization barriers between the workers and allows each worker to sample the environment and apply gradient updates to a global model without locks. Thus, updates occur with respect to a stale version of the model. Staleness hurts RL algorithms in two ways. Firstly, like with supervised-learning, gradients computed with respect to a stale model are inaccurate. Additionally, with reinforcement learning, data gathering requires inferencing the model; thus, a stale model adds additional inaccuracies when the training data is gathered. IMPALA, or the Importance Weighted Actor Learner Architecture (Espeholt et al., 2018), is similarly

[2] https://github.com/drakesvoboda/SysForAIProject.

asynchronous but introduces a novel V-trace algorithm that corrects for data gathered using a stale view of the model.

Our implementations of EASGD and ASP modify the communication of A3C. For EASGD, each worker only infrequently updates the global set of parameters by taking an elastic-average. For ASP, each worker only transmits the significant updates; we additionally implement the mirror-clock from Gaia (Hsieh et al., 2017) to ensure that the slowest worker does not get too far behind the fastest worker.

We test our implementations on a CloudLab cluster with 3 worker nodes. We train on three OpenAI gym environments: Acrobot, Cart-Pole, and QBert. For each environment, we evaluate performance using both 5 and 10 worker threads per node. All algorithms train for a total of 1 hour on each environment. Training plots are shown in figure 4 and training throughput is shown in figure 5.

A3C has the highest throughput of the methods; our implementations of EASGD and ASP follow closely behind. PPO and IMPALA have significantly lower throughput compared to other algorithms in all of the experiments. IMPALA does not train well in any of our experiments and gives the lowest mean reward in all of our tests. IMPALA is designed for achieving high throughput for hardware configurations with few accelerator resources but many CPU resources; thus, this is an unfair comparison. For the Acrobot environment, PPO, A3C, ASP and EASGD all train well and all of them approach the maximum possible reward. Although PPO has a significantly lower throughput, it converges quickly with respect to wall time in the Cart-Pole experiments. This suggests that the inaccuracies introduced by the asynchronous methods harm convergence in this case.

While ASP was able to achieve the maximum possible mean reward in Acrobot, it fails to converge to an optimal policy for Cart-Pole and is unstable during training; this suggests a dependence between the particular environment and training stability. PPO still performs the best achieving the highest reward quickly.

The QBert experiment shows the advantage of the asynchronous methods. PPO's throughput is far too low for this environment and it fails to train entirely. For QBert, we found that training was very unstable for the asynchronous methods and their convergence was very sensitive to initialization. In the 5 worker case, EASGD significantly outperforms the other methods. In the 10 worker case, A3C performs best. We believe that this performance is somewhat arbitrary as it is not consistent between runs. The performance on the other environments was far more consistent.

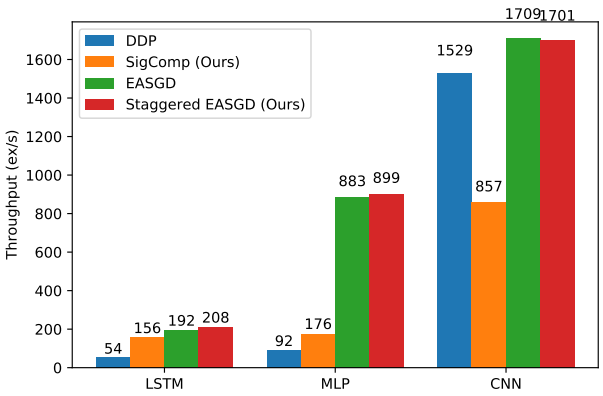


Figure 2. Training throughput (examples per second).

5 Discussion and Future Work

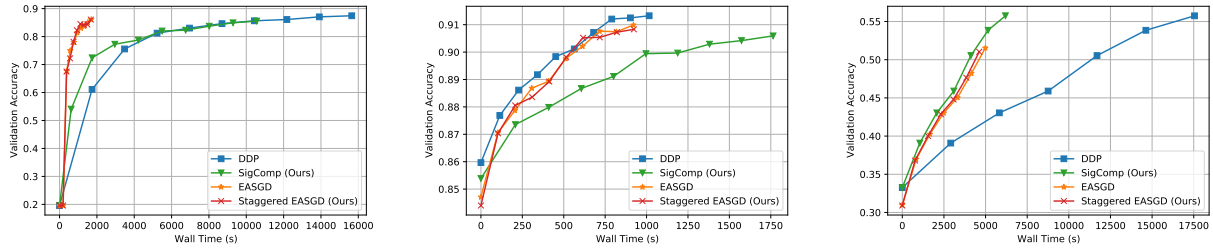
There is a lot of future work to be done and this project could take multiple different directions.

Overall, we should study these algorithms on more learning problems and different hardware. Our staggered EASGD optimization did not achieve much speed-up, but there may be a more apparent speed-up with different configurations. Further, our staggering optimization could be used for many methods that limit the frequency with which parameters are transmitted including federated learning (McMahan et al., 2017) and AdaComm (Wang & Joshi, 2019).

A very important thing to test empirically is significance compression's performance compared to other gradient sparcification methods. This comparison is crucial, but we ran out of time to run experiments to include in this report. Further, we believe that Alistarh et al.'s proof of convergence for other gradient sparcification methods can be used to provide convergence guarantees for significance compression.

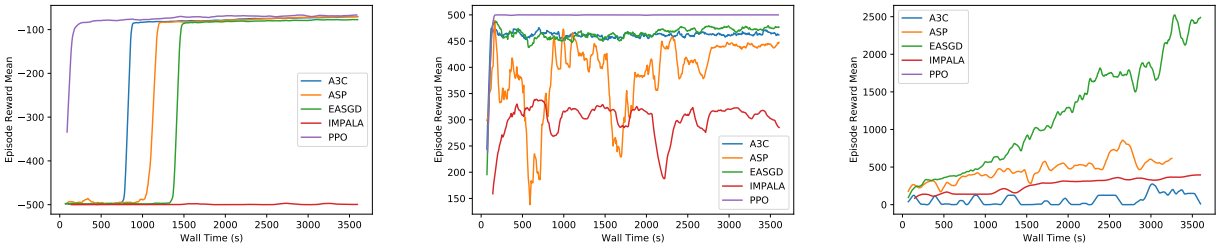
Other asynchronous SGD and gradient sparcification methods employ clever tricks when using explicit momentum terms (Lin et al., 2020; Mitliagkas et al., 2016). The authors of EASGD show empirically that it performs best with Nesterov momentum (Zhang et al., 2015). We did not train with momentum, for future work, we should tune the momentum parameter and relate the optimal momentum with the significance threshold value or communication period. Following Mitliagkas et al.'s finding that asynchronous SGD induces momentum, we predict that the optimal momentum value will decrease as the significance threshold or communication period increases.

EASGD and Meta Learning As an aside, there is an apparent relationship between EASGD the the first-order meta learning algorithm REPTILE (Nichol et al., 2018). As a brief introduction, meta learning methods attempt to learn a

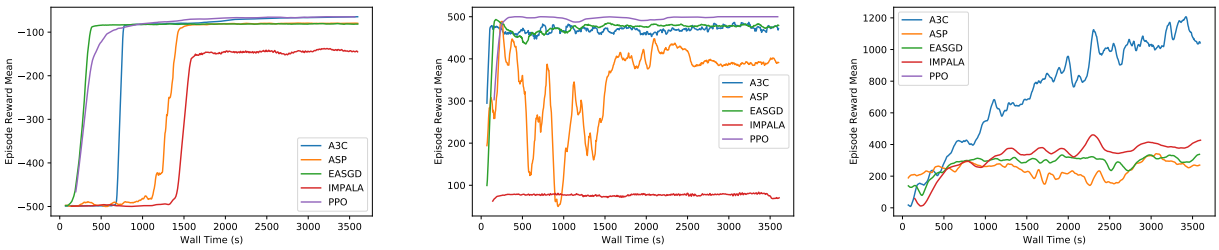


(a) MLP (b) CNN (c) LSTM

Figure 3. Validation accuracy during training for (a) and 7 layer MLP (b) a 2 layer CNN and (c) a 1 layer LSTM.



(a) Acrobot 5 workers (b) Cart Pole 5 workers (c) QBert 5 workers



(d) Acrobot 10 workers (e) Cart Pole 10 workers (f) QBert 10 workers

Figure 4. Reinforcement learning training.

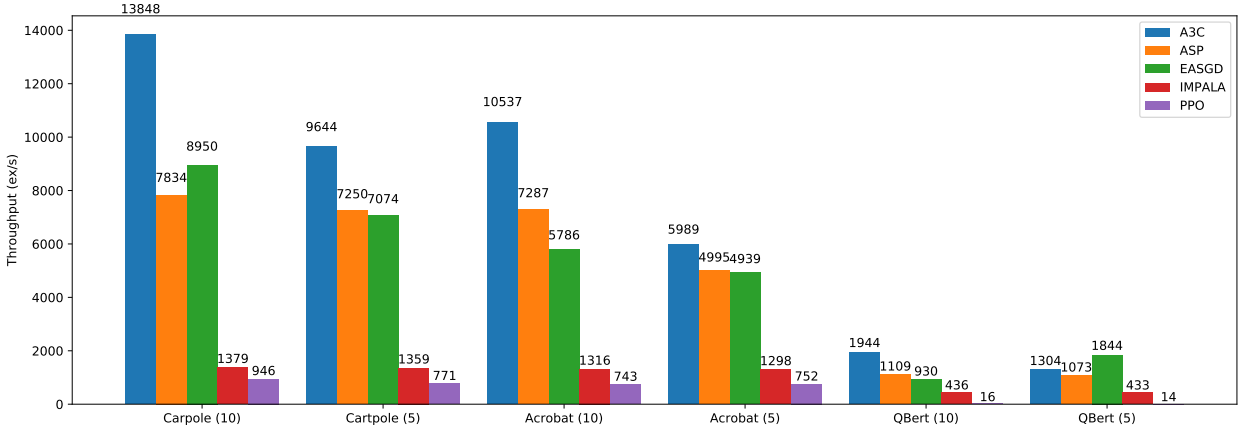


Figure 5. RL training throughput (examples per second).

learning algorithm itself that can quickly specialize to new tasks. This usually means learning some initialization to a DNN that can achieve high accuracy on some task by fine-tuning for a small number of iterations. REPTILE works by maintaining a set of master parameters ($\tilde{x}$), sampling a particular task i and training for a fixed number of iterations on that task to achieve an updated set of parameters (x^i), then taking an elastic average to update the master parameters ($\tilde{x} \leftarrow \tilde{x} + \alpha(x^i - \tilde{x})$). When multiple tasks are sampled and trained in parallel, this is equivalent to EASGD. There may be interesting insights to gain by analyzing EASGD as a meta learning algorithm; we leave this to future work. Chen, Luo et al. have studied the performance of the meta learning algorithms MAML, FOMAML, and Meta-SGD in a federated learning setting (Chen et al., 2019). It would be interesting to study the performance of elastic-averaging as a drop-in replacement in this setting.

REFERENCES

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding, 2017.

Alistarh, D., Hoefler, T., Johansson, M., Khirirat, S., Konstantinov, N., and Renggli, C. The convergence of sparsified gradient methods, 2018.

Chen, F., Luo, M., Dong, Z., Li, Z., and He, X. Federated meta-learning with fast convergence and efficient communication, 2019.

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 571–582, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018. URL http://arxiv.org/abs/1802.01561.

Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pp. 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 629–647, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh.

Jayarajan, A., Wei, J., Gibson, G., Fedorova, A., and Pekhimenko, G. Priority-based parameter propagation for distributed dnn training. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 132–145, 2019. URL https://proceedings.mlsys.org/paper/2019/file/d09bf41544a3365a46c9077ebb5e35c3-Paper.pdf.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 583–598, USA, 2014. USENIX Association. ISBN 9781931971164.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. Rllib: Abstractions for distributed reinforcement learning, 2018.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training, 2020.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993. ISSN 0891-2017.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data, 2017.

Mitliagkas, I., Zhang, C., Hadjis, S., and Ré, C. Asynchrony begets momentum, with an application to deep learning, 2016.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017. URL http://arxiv.org/abs/1712.05889.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. 2011.

Nichol, A., Achiam, J., and Schulman, J. On first-order meta-learning algorithms, 2018.

Niu, F., Recht, B., Re, C., and Wright, S. J. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.

Peng, Y., Zhu, Y., Chen, Y., Bao, Y., Yi, B., Lan, C., Wu, C., and Guo, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 16–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359642. URL https://doi.org/10.1145/3341301.3359642.

Renggli, C., Ashkboos, S., Aghagolzadeh, M., Alistarh, D., and Hoefler, T. Sparcml: High-performance sparse communication for machine learning, 2019.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. *CoRR*, abs/1905.13727, 2019. URL http://arxiv.org/abs/1905.13727.

Wang, J. and Joshi, G. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 212–229, 2019. URL https://proceedings.mlsys.org/paper/2019/file/c8ffe9a587b126f152ed3d89a146b445-Paper.pdf.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

Zhang, S., Choromanska, A., and LeCun, Y. Deep learning with elastic averaging sgd. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 685–693, Cambridge, MA, USA, 2015. MIT Press.