JMLR: Workshop and Conference Proceedings 80:1–16, 2017 ACML 2017

Accumulated Gradient Normalization

Joeri R. Hermans JOERI.HERMANS@DOCT.ULG.AC.BE
Gerasimos Spanakis JERRY.SPANAKIS@MAASTRICHTUNIVERSITY.NL
Rico Möckel RICO.MOCKEL@MAASTRICHTUNIVERSITY.NL
Department of Electrical Engineering and Computer Science, Liège University, Belgium
Department of Data Science & Knowledge Engineering, Maastricht University, The Netherlands

Abstract

This work addresses the instability in asynchronous data parallel optimization. It does so by introducing a novel distributed optimizer which is able to efficiently optimize a centralized model under communication constraints. The optimizer achieves this by pushing a normalized sequence of first-order gradients to a parameter server. This implies that the magnitude of a worker delta is smaller compared to an accumulated gradient, and provides a better direction towards a minimum compared to first-order gradients, which in turn also forces possible implicit momentum fluctuations to be more aligned since we make the assumption that all workers contribute towards a single minima. As a result, our approach mitigates the parameter staleness problem more effectively since staleness in asynchrony induces (implicit) momentum, and achieves a better convergence rate compared to other optimizers such as asynchronous EASGD and DYNSGD, which we show empirically.

Keywords: Distributed Optimization, Neural Networks, Gradient Descent

1. Introduction

Speeding up gradient based methods has been a subject of interest over the past years with many practical applications, especially with respect to Deep Learning. Despite the fact that many optimizations have been done on a hardware level, the convergence rate of very large models remains problematic. Therefore, data parallel methods next to mini-batch parallelism have been suggested Dean et al. (2012); Ho et al. (2013); Hadjis et al. (2016); Recht et al. (2011); Louppe and Geurts (2010); Jiang et al. (2017); Zhang et al. (2015) to further decrease the training time of parameterized models using gradient based methods. Nevertheless, asynchronous optimization was considered too unstable for practical purposes due to a lacking understanding of the underlying mechanisms, which is an issue this work addresses.

Data Parallelism is an inherently different methodology of optimizing parameters. As stated above, it is a technique to reduce the overall training time of a model. In essence, data parallelism achieves this by having n workers optimizing a central model, and at the same time, processing n different shards (partitions) of the dataset in parallel over multiple workers[1]. The workers are coordinated in such a way that they optimize the parameterization of a central model or central variable, which we denote by $\tilde{\theta}_t$. The coordination mechanism of

1. A worker in this work is a process on a single machine. However, it is possible that multiple workers share the same machine. Nevertheless, one could construct the distribution mechanism (even manually) in such a way every worker will be placed on a different machine.

the workers can be implemented in many different ways. A popular approach to coordinate workers is to employ a centralized *Parameter Server* (PS). The sole responsibility of the parameter server is to aggregate model updates coming from the workers (*worker commits*), and to handle parameter requests (*worker pulls*).

Recently, a theoretical contribution has been made Mitliagkas et al. (2016) which defines asynchronous optimization in terms of (implicit) *momentum* due to the presence of a queuing model of gradients based on past parameterizations. This paper mainly builds upon this work, and Zhang et al. (2015) to construct a better understanding why asynchronous optimization shows proportionally more divergent behavior when the number of parallel workers increases, and how this affects existing distributed optimization algorithms.

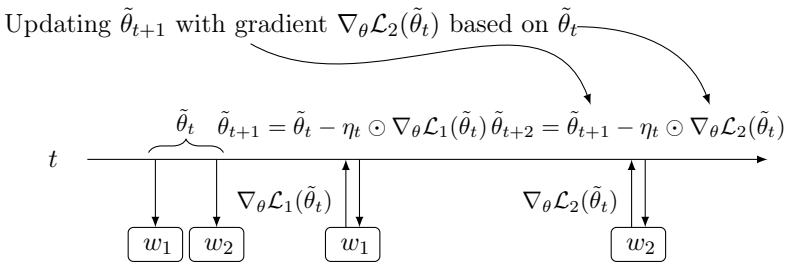
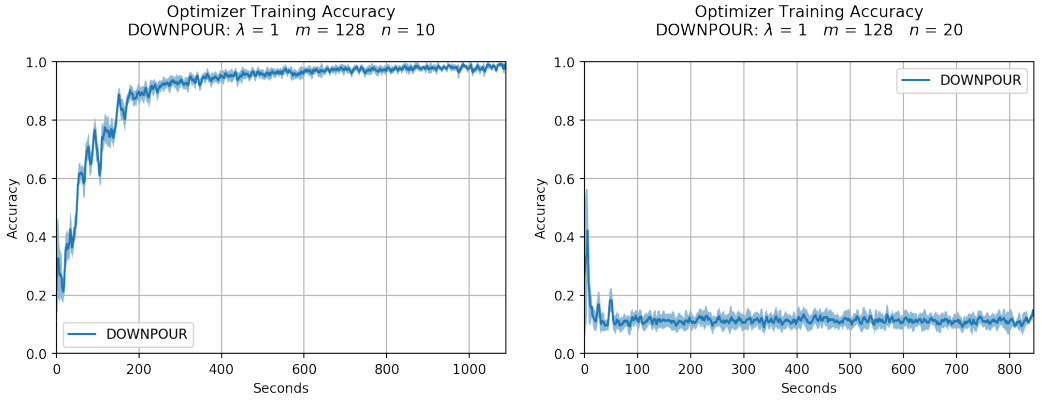


Figure 1: In Asynchronous Data Parallelism workers compute and commit gradients to the PS asynchronously. This has as a side-effect that some workers are computing, and thus committing, gradients based on old values. These gradients are called *stale gradients* in literature. In this particular example there are 2 workers w_1, and w_2. At the start of the optimization process, both workers pull the most recent parameterization, $\tilde{\theta}_t$, from the PS. Now all workers start computing gradients asynchronously based on the pulled parameterization. However, since the PS incorporates gradients into the center variable asynchronously as a simple queuing (FIFO) model, other workers will update the center variable with gradients based on a stale parameterization, as shown above. Finally, assuming that the computing cluster is homogeneous, we can derive from this figure that the expected staleness of a gradient update is $\mathbf{E}[\tau] = (n-1)$, as mentioned in Mitliagkas et al. (2016).

The rest of the paper is organized as follows. In Section 2 we present the intuition for our method. Section 3 describes the proposed method in full, followed by an experimental validation in Section 4. Finally, Section 5 concludes the paper by giving an overview of the contributions presented in this work.

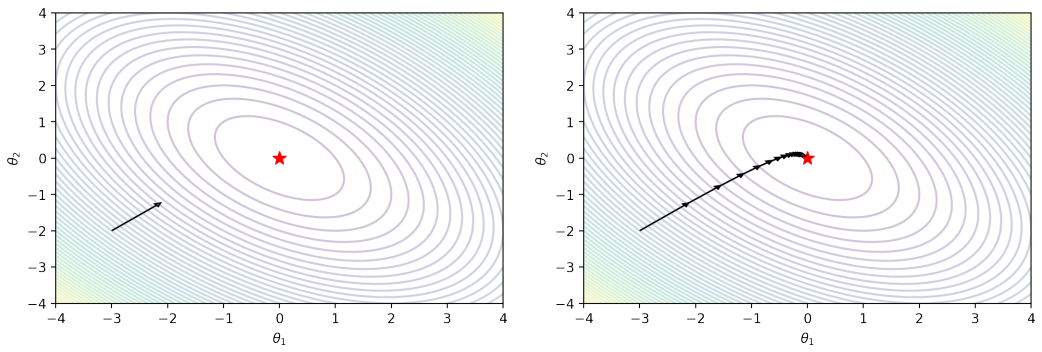
2. Concept & Intuition

The main issue with DOWNPOUR Dean et al. (2012) is the requirement of constant communication with the parameter server after every gradient computation. Furthermore, as the number of parallel workers increases, DOWNPOUR fails to converge due to the amount of *implicit momentum*, as shown in Figure 2. To reduce the amount of communication with the parameter server, one could take ideas from EASGD, and perform several iterations of local exploration before committing the gradients to the parameter server. However, in the



(*a*) $n = 10$ (*b*) $n = 20$

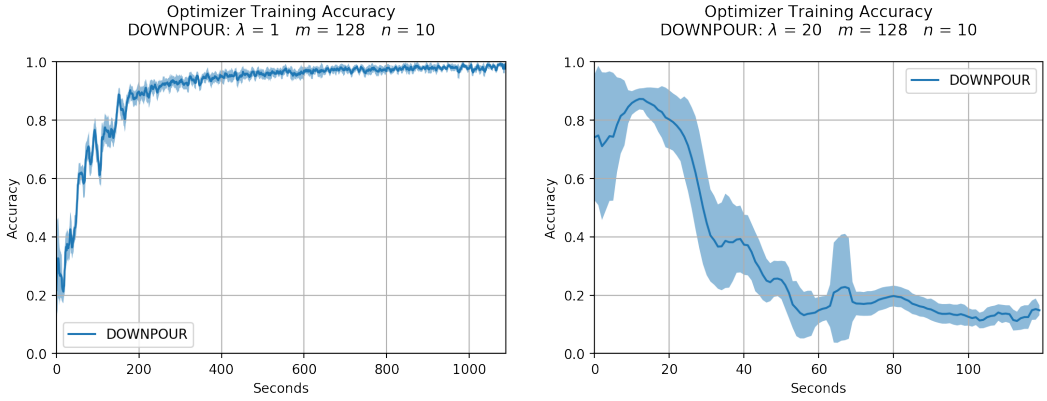
Figure 2: DOWNPOUR divergence due to number ($n = 20$) of asynchronous workers in the optimization process Mitliagkas et al. (2016) for this particular problem, and not dealing with parameter staleness in a more intelligent way. Lowering the number workers ($n = 10$) causes the central variable to converge.



(*a*) No Gradient Accumulation (*b*) Gradient Accumulation

Figure 3: This figure shows the difference between regular first-order gradients (a), and accumulated gradients (b). We observe that *accumulated gradients are proportionally larger to the number of exploration steps*. However, the accumulated gradient does provide a better direction compared to first-order gradients.

case of algorithms like DOWNPOUR, that is, where gradients are committed to the parameter server in an asynchronous fashion with no mechanisms in place to ensure convergence, more local exploration results in proportionally larger gradients and as a result, complicate the staleness and the implicit momentum problem even further. To intuitively show why this is an issue, let us consider Figure 3. In a DOWNPOUR setting, first-order gradients such as in Subfigure (a) are committed to the parameter server. However, when an algorithm allows for a certain amount of local exploration, the gradient that is committed to the parameter server is typically an *accumulated gradient* as shown in Subfigure (b).



(a) $\lambda = 1$ (b) $\lambda = 20$

Figure 4: Illustration of divergence due to gradient accumulation in DOWNPOUR. In Figure 2, we say that for $n = 10$ DOWNPOUR converged to a good solution. In order to reduce the training time, we decrease the communication frequency (increasing λ). However, due to the larger gradients that are committed to the parameter server, which increases the amount of implicit momentum, the central variable is not able to converge as before.

Now, imagine two asynchronous environments where respectively no gradient accumulation takes place, and one where does. In the environment where no gradient accumulation is performed, as in regular DOWNPOUR, first-order gradients are committed to the parameter server. However, we know that DOWNPOUR diverges when the number of asynchronous workers is too high due to the amount of implicit momentum Mitliagkas et al. (2016). As a result, careful tuning is required when no adaptive methods are applied in order to guarantee convergence. Nevertheless, given the fact that DOWNPOUR converges with $n = 10$ workers in Figure 2, and our knowledge about gradient accumulation, i.e., *accumulated gradients that are committed are proportional to the number of exploration steps for every worker, and provide better directions to a minimum*, we would expect that for some amount of local exploration while using the same hyperparameterization (with the exception of local exploration steps λ) DOWNPOUR would diverge again due to the magnitude of the accumulated gradients. This behaviour is illustrated in Figure 4, and confirms our hypothesis.

To reduce the magnitude of the accumulated gradients, and thereby reducing the amount of implicit momentum, while at the same time preserving the better direction that has been provided due to the amount of local exploration, we propose to normalize (average) the accumulated gradient with the amount of local steps that have been performed by the workers (λ), shown in Equation 1[2]. We call this technique of normalizing the accumulated gradient *Accumulated Gradient Normalization* or AGN. An initial critique of this technique would be that by normalizing the accumulated gradient, AGN would in effect be undoing the work that has been done by a single worker. This seems at first a valid criticism, however, one needs to take into account that AGN is actually using the worker exploration steps to compute a better gradient based on first-order gradients.

2. Note if $\lambda = 1$, AGN generalizes to DOWNPOUR.

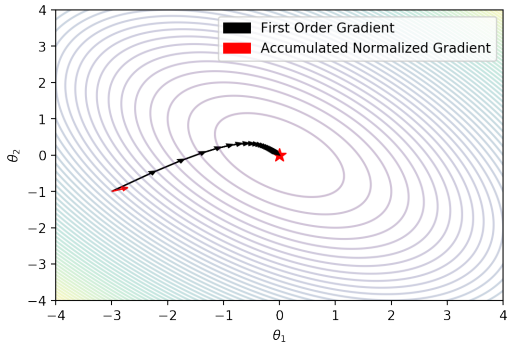


Figure 5: After pulling the most recent parameterization of the central variable from the parameter server, the worker starts accumulating λ first order gradients, and applies those gradients locally to explore the surrounding error space. Finally, after λ exploration steps have been performed, the accumulated is normalized w.r.t. λ and send to the parameter server.

3. Method

The AGN update rule for a worker and the parameter are described in Equation 1 and Equation 2 respectively. Since AGN is using local steps (λ) to compute a better gradient based on a *sequence* of first-order gradients, it can also be used under communication constraints like EASGD since less communication with the parameter server is required due to computation of a local sequence of first-order gradients. Figure 5 shows how an AGN gradient is obtained and computed using Equation 1 by following Algorithm 1. In this setting η_t denotes the learning rate of a worker at time t, and m denotes the size of the mini-batch which is identical across all workers.

$$\Delta\theta^k = -\frac{1}{\lambda}\sum_{i=0}^{\lambda}\eta_t\frac{1}{m}\sum_{j=0}^{m-1}\nabla_\theta\mathcal{L}(\theta_i; x_{ij}; y_{ij}) \tag{1}$$

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t + \Delta\theta^k \tag{2}$$

An interesting thought-experiment would be to explore what would happen if the workers would communicate with the parameter server after a very large number of steps, that is, when λ approaches ∞. How would the normalized accumulated gradients look like in such a situation, described by Equation 3?

$$\lim_{\lambda\to\infty} -\frac{\sum_{i=0}^{\lambda}\eta_t\frac{1}{m}\sum_{j=0}^{m-1}\nabla_\theta\mathcal{L}(\theta_i; x_{ij}; y_{ij})}{\lambda} \tag{3}$$

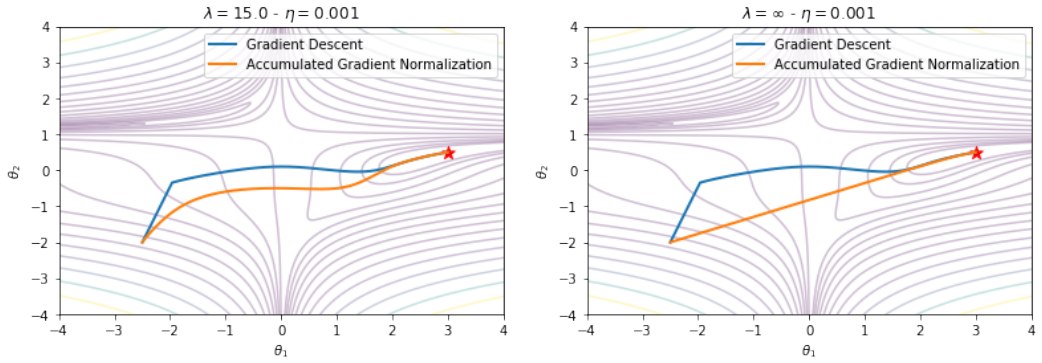
In order to completely understand how the worker deltas would look like after $\lambda = \infty$ steps, one first needs to understand the individual components of Equation 3. The most inner component, $\eta_t\frac{1}{m}\sum_{j=0}^{m-1}\nabla_\theta\mathcal{L}(\theta_i; x_{ij}; y_{ij})$, is just the computation of a mini-batch using $m-1$ samples, where index i denotes the current step in the gradient accumulation. Please note that a mini-batch can differ for different values of i as training samples are randomly

Algorithm 1 Worker procedure of AGN.

procedure AGNWorker(k)
  $\theta_0^k \leftarrow \tilde{\theta} \leftarrow$ Pull()
  $t \leftarrow 0$
  while not converged **do**
    $i \leftarrow 0$
    $a \leftarrow 0$
    while $i < \lambda$ **do**
      $\mathbf{x}, \mathbf{y} \leftarrow$ FetchNextMiniBatch()
      $g \leftarrow -\eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ ▷ Gradient from, e.g., Adam Kingma and Ba (2014)
      $a \leftarrow a + g$
      $\theta_{t+1}^k = \theta_t^k + g$
      $i \leftarrow i + 1$
      $t \leftarrow t + 1$
    end
    $a \leftarrow \frac{a}{\lambda}$ ▷ Normalization step.
    Commit(a)
    $\theta_t^k \leftarrow$ Pull()
  end
end procedure

retrieved from the dataset. After computing the gradient based on the mini-batch, the local model will be updated as $\theta_{i+1} = \theta_i - \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_\theta \mathcal{L}(\theta_i; x_{ij}; y_{ij})$. This process goes on for λ steps, while at the end, the accumulated is normalized with respect to λ.

Let us assume we have a smooth convex error space, or a smooth non-convex error space with at least a single minima. Due to the existence of a minima in both cases, first order gradients will eventually converge to, or in the neighbourhood of said minima. Furthermore, we make the assumption that the hyperparameterization during the training procedure will not change. For instance, no learning rate decay after x number of steps. Under these assumptions, it is trivial to realize that applying gradient descent for ∞ steps will cause the parameterization to converge in a minima. Of course, given that the hyperparameterization, and the data allow for convergence to occur. As a result, the term $\sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_\theta \mathcal{L}(\theta_i; x_{ij}; y_{ij})$ is finite, even after applying ∞ steps of mini-batch updates. To simplify our problem, let us denote $\boldsymbol{c}$ as the *finite* result of the top term in Equation 3 for $\lambda = \infty$. Furthermore, since $\boldsymbol{c}$ is finite, the equation can be treated as an instance of $\frac{1}{\infty}$, which approaches 0. This implies that for a very large λ, the normalized accumulated gradients will basically be $\mathbf{0}$. However, what is interesting is that the normalized accumulated gradients directly point towards a minima due to the large amount of exploration steps that have been performed. Subsequently, one can view a normalized accumulated gradient when λ approaches ∞ as a point, but with a direction. Therefore, when convergence is obtained, the path the central variable traversed is a straight line towards the minima, as shown in Figure 6.



(a) $\lambda = 15$ (b) $\lambda = \infty$

Figure 6: AGN for different values of λ. This small experiment shows that when $\lambda = \infty$, the path the central variable traverses is equal to a straight line towards the minima.

The thought experiments described above help us in several ways if we make several additional assumptions. The first assumes that normalized accumulated gradients with $\lambda = \infty$ can be computed immediately, that is, without a delay. This is of course an unrealistic assumption. However, one needs to consider realistic communication constraints. Given a certain network throughput, what is the amount of local communication that needs to be performed in order for a parameter commit to be "worth it"? As mentioned above, $\lambda = \infty$ is not a very good solution since the normalized accumulated gradient will converge to $\mathbf{0}$ in the limit. Nevertheless, if the normalized accumulated gradient could be computed immediately, as we assumed, the central variable would traverse the shortest path to a minima, in contrast to first order gradients. Of course, this is not a realistic assumption. Furthermore, this issue is quite similar to *stochastic gradient descent* vs. *mini-batch gradient descent*, since in AGN we also have to make the decision between more frequent parameter updates, and more "local" iterations to compute a better gradient, where better in the case of mini-batch gradient descent means less-noisy.

In most settings, the size of a mini-batch is determined empirically, and is dependent on the noise of the gradients. Furthermore, when using mini-batch gradient descent, a trade-off is made between more frequent parameter updates, i.e., a smaller mini-batch, or more robust and consistent gradients by increasing the size of a mini-batch which results in a more accurate approximation of the first order curvature. This is similar to our situation. Yet, in mini-batch gradient descent you are basically trying to estimate a hyperparameter based on several unknowns, i.e., convergence based on error space and noise of individual gradients. However, AGN is balancing the amount of local computation to produce a better gradient, with the throughput of the network, which is a known variable. For instance, imagine a hypothetical communication infrastructure which is able to apply the commits of the workers directly into the workers with no delay. In this situation, one could apply DOWNPOUR. However, remember from Figure 4 that DOWNPOUR does not handle an increased amount of asynchronous parallelism ($n = 20$). As a result, even in an ideal situation DOWNPOUR

will not be able to converge due to the amount of implicit momentum.

Nevertheless, the situation in AGN is different as will become apparent in Section 4. Contrary to DOWNPOUR, AGN does not commit first order gradients to the parameter server, but rather a normalized sequence of first order gradients which result in better directions towards a minima, as discussed above. Therefore, AGN worker deltas will point more or less in more optimal direction and thereby reducing the negative effects of implicit momentum in first order gradients.

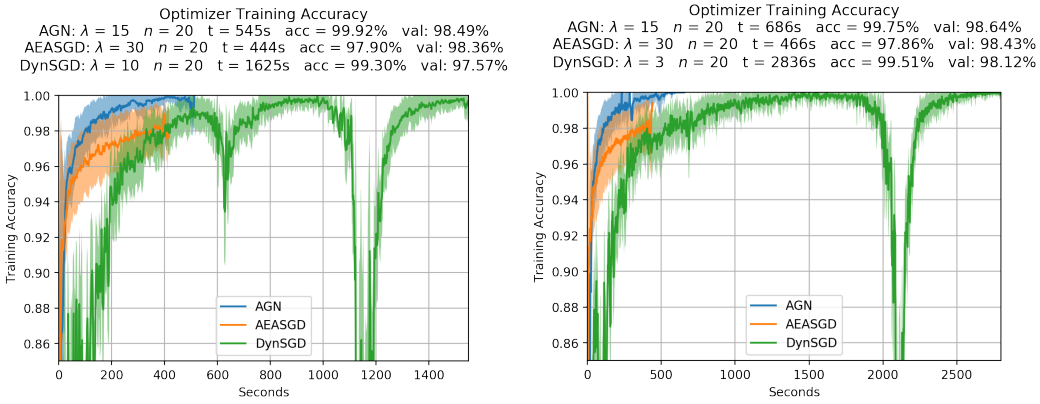
4. Experimental Validation

This Section evaluates AGN against different distributed optimization algorithms. MNIST LeCun et al. (1998) is used as a benchmark dataset, and all optimizers the same model with identical parameterization of the weights. Furthermore, we will set the mini-batch size to $m = 128$ in all optimizers, and use *40 epochs* worth of training data that will be equally distributed over all n workers. Our computing infrastructure consists a relatively small cluster of *15 nodes* with a *10Gbps interconnect*, most of them in the same rack, each having 2 Intel® Xeon® CPU E5-2650 v2 @ 2.60GHz CPU's, where every CPU has 8 cores and 2 threads. No GPU's are used during training, and no learning rate decay is applied. Our optimizer and experiments are implemented and executed using *dist-keras*[3], which are available in the package.

Our initial experiment, shown in Figure 7, shows the training accuracy of AGN, AEASGD, and DYNSGD over time. In this experiment, we use a near-optimal hyperparameterization for all optimizers to ensure convergence. Looking at Figure 7, we observe an increase in training performance for AGN, both in training accuracy, and in training time when compared to current state-of-the-art algorithms such as AEASGD and DYNSGD. Furthermore, DYNSGD scales the gradients down with respect to staleness τ, which in effect is $(n-1)^{-1}$ since $\mathbf{E}[\tau] = n-1$. As a result, DYNSGD does not handle the parameter staleness problem appropriately since it does not take the distance between the parameterizations into account. This is validated in Figure 7 because of the observed divergent behaviour of the optimizer. Furthermore, due to the relatively high communication frequency ($\lambda = 10$, $\lambda = 3$), DYNSGD will take longer to process all data since more communication with the parameter server is required.

Contrary to AEASGD, AGN is able to cope more effectively with an increased amount of parallelism, as its training accuracy only starts to decline from $n = 25$ asynchronous workers, while the validation accuracy is barely fluctuating, as shown in Figure 8. An obvious follow-up question to this result would be to question the fact whether increasing the amount of workers really improves the temporal efficiency optimizer, i.e., the amount of time it takes to reach a certain training accuracy. In fact, it does reduce the training time to reach a certain training accuracy, as shown in Figure 9. However, several factors have to be taken into account.

3. github.com/cerndb/dist-keras



(*a*) DYNSGD $\lambda = 10$ (*b*) DYNSGD $\lambda = 3$

Figure 7: In this experiment we train all optimizers on 40 epochs worth of data with a mini-batch of $m = 128$. We observe that AGN significantly outperforms all other optimizers. Furthermore, due to staleness-handling method of DYNSGD, the optimizer is not able to handle accumulated gradients which results in non-stale accumulated gradients being incorporated directly into the central variable with the disadvantage that other workers are even more stale in terms of parameter distance. Which is the root cause of this divergent behaviour. In Subfigure (b) we reduce the amount of local exploration steps, which in turn reduces the length of the accumulated gradient. Therefore causing other workers to be less stale, and consequently reducing the divergent effects we observed in Subfigure (a).

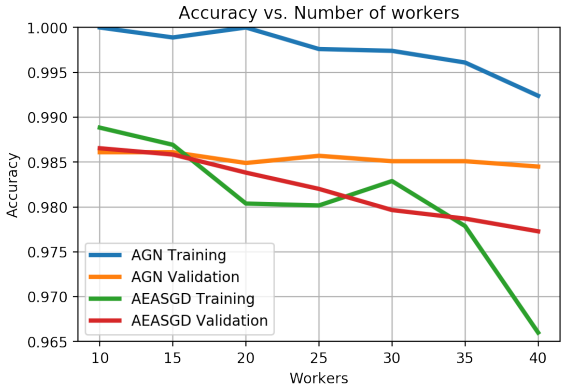
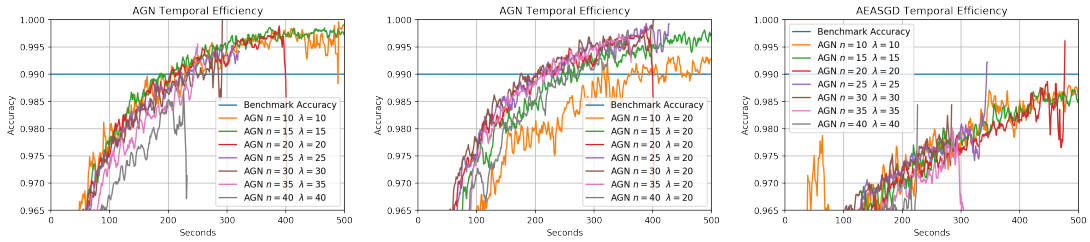


Figure 8: Decline of the training accuracy of both AGN and AEASGD as the number of asynchronous workers increases. From these experiments, we observe that AGN is more robust to an increased amount of asynchrony as the training accuracy only starts to decrease from $n = 25$ workers, while the validation accuracy remains stable even with 40 workers.



(*a*) AGN varying λ (*b*) AGN static λ (*c*) AEASGD varying λ

Figure 9: Training accuracy plots for different configurations of the distributed hyperparameters. In the case of a varying λ with respect to the number of workers (to minimize the noise of the commits), we observe that optimizers with a higher communication frequency (small λ), are actually benefiting from the more frequent updates with the parameter server. However, as the number of asynchronous workers grows, a low communication frequency increases the noise in the commits due to parameter staleness. Furthermore, if the *lambda* is too large, less frequent parameter server updates occur, which results in a slower convergence rate since more time is spent locally. As a result, a balance is required similar to determining the size of a mini-batch.

The first being an increased amount of staleness that is inserted into the system as the number of asynchronous workers increase. This effect is difficult to mitigate. Previous approaches Jiang et al. (2017) propose to scale gradient commits down proportionally to the number of stale steps. However, as previously shown, this is not an effective solution since accumulating gradients locally, is in effect making the gradients larger, and as a result, committing accumulated gradients increases the *distance* between the central variable and other workers. The second and final issue is the balance between updating the central variable with a certain frequency, and the amount of local work to effectively reduce the training time due to high communication costs. In effect, this resembles the situation usually one has when selecting a mini-batch size m, i.e., do we allow for more frequent parameter updates (small m), or do we compute a less noisy first order gradient by increasing m, thereby reducing the frequency of parameter updates and the convergence of a model? In Figure 11, we evaluate varying values of λ for a specific number of asynchronous workers n to show that similar manual tuning is required. In all cases, we observe configurations with $\lambda = 40$ usually have the slowest convergence rate with respect to other configurations with higher communication frequencies.

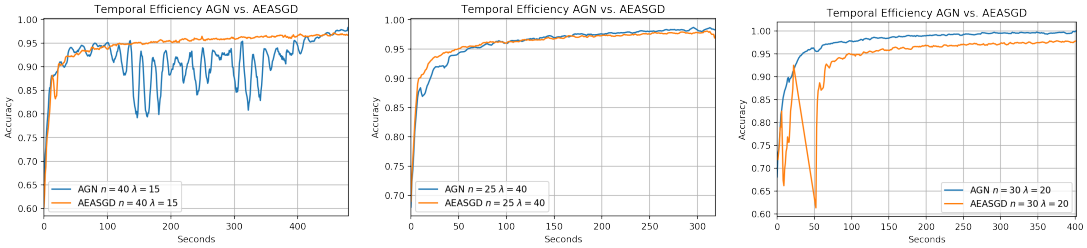
Nevertheless, what is really interesting, is why configurations with low communication frequencies actually do converge, in contrast to configurations with high communication frequencies (with respect to the number of workers). Since our definition of staleness relates to the distance between the *current* parameterization of a worker, and the *current* parameterization of the central variable. One can imagine that increasing the number of asynchronous workers, effectively increases the *distance* between the parameterizations of the workers and the *current* central variable due to the queuing model discussed before, i.e., worker deltas are incorporated in the central variable in a queuing fashion. Yet, parameter staleness still does not explain why configurations with low communication frequencies converge, as opposed

to configurations with higher communication frequencies. The question begs, is convergence guaranteed due to the amount of local exploration, thus providing the parameter server with a better "direction", as shown in Figure 5. Or, due to limit condition described above, which eventually scales the worker deltas down to 0 as the communication frequency decreases (λ increases)? This is a rather difficult question to answer, since there might be a synergy since large gradients are usually considered a bad thing. However, the normalization step takes the average gradient of all accumulated gradients. As a result, the limit condition does not apply here. Summarized, the stability property of AGN arises from the fact that *implicit momentum* fluctuations will mostly be in line with a minima due to the better directions AGN provides.

To compare AGN against AEASGD (since this optimizer shows almost no divergent behaviour for a wide range of hyperparameters), we introduce *temporal efficiency* in terms of the surface described by a training metric. This means that for some optimizer a, we have a function $f_a(t)$ which describes the performance of a model at time t, e.g., $f_a(t)$ describes the training accuracy of the model at time t. If we integrate over t, we obtain a surface representing the performance of a model over time. If we would do this for an other optimizer b, and divide the surface of optimizer a by the performance surface of optimizer b, we get a ratio which describes how optimizer a is performing compared to optimizer b. If this ratio is larger then 1, it means that optimizer a is outperforming optimizer b, else, it is the other way around (unless the surfaces are equal of course). However, in order to compute a *fair* surface area, we have to limit the computation to the *minimal shared training time* m. This is done to prevent that optimizers with a longer training time have a significant advantage, since they have more time to produce a better model. To summarize, we define the temporal efficiency $\mathcal{E}$ of two optimizers a and b as the ratio of their performance surface, as stated in Equation 4. Using *temporal efficiency*, we can make a more qualitative judgment which optimizer is performing better in different scenarios *since it also incorporates the stability of the optimizer*.

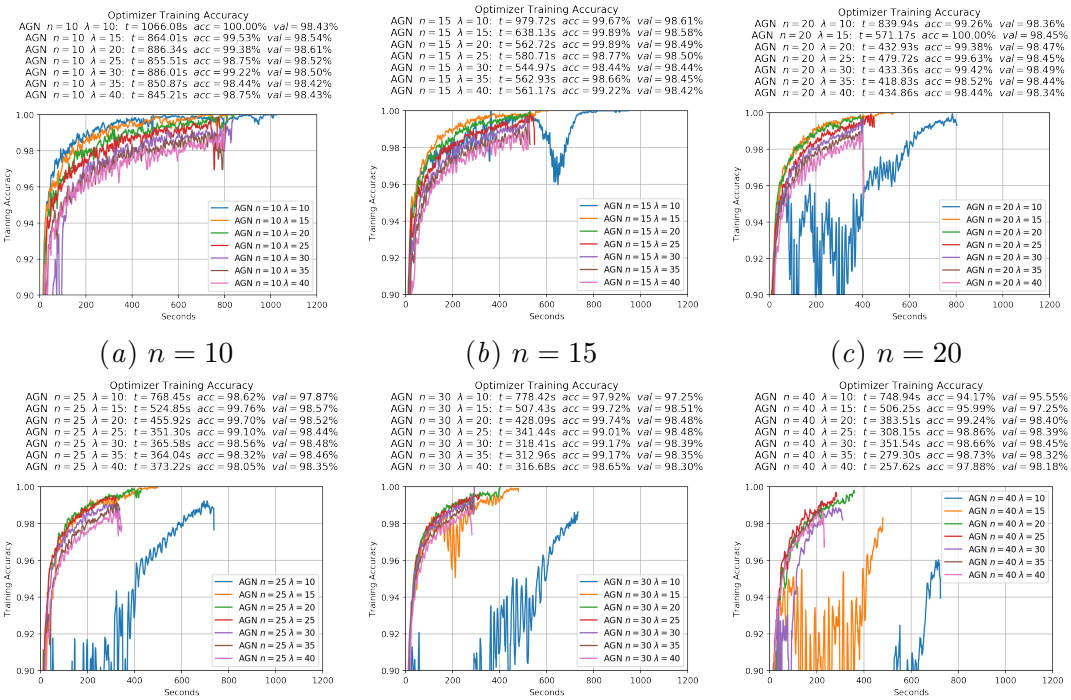
$$\mathcal{E}(a, b) = \frac{\int_0^m f_a(t)\, dt}{\int_0^m f_b(t)\, dt} \tag{4}$$

Finally, we apply *temporal efficiency* to compare AGN against AEASGD and summarize the results in Table 1. We make the observation that increasing the amount of asynchrony results in a deterioration of the training accuracy (which is expected since more staleness, and thereby, implicit momentum is induced). However, the rather unexpected property is that increasing the amount of asynchronous workers results in an early *flattening* of the training accuracy in AEASGD. This is due to an equilibrium condition which is present in EASGD Hermans (2017). Since we increase the amount of asynchrony in the optimization procedure, workers will reach the equilibrium condition faster because the elastic difference is computed based on the most recent parameterization of the central variable. Meaning, as soon as AEASGD is done computing λ iterations, the central variable is pulled to the worker where the elastic difference is computed based on the recently pulled central variable, which is very stale due to the low communication frequency and high number of asynchronous



(a) $\mathcal{E} = 0.964$ (b) $\mathcal{E} = 0.999$ (c) $\mathcal{E} = 1.044$

Figure 10: Several accuracy plots of AGN and AEASGD. All subfigures show the computed *temporal efficiency* of AGN, which were obtained by applying Equation 4.



(a) $n = 10$ (b) $n = 15$ (c) $n = 20$

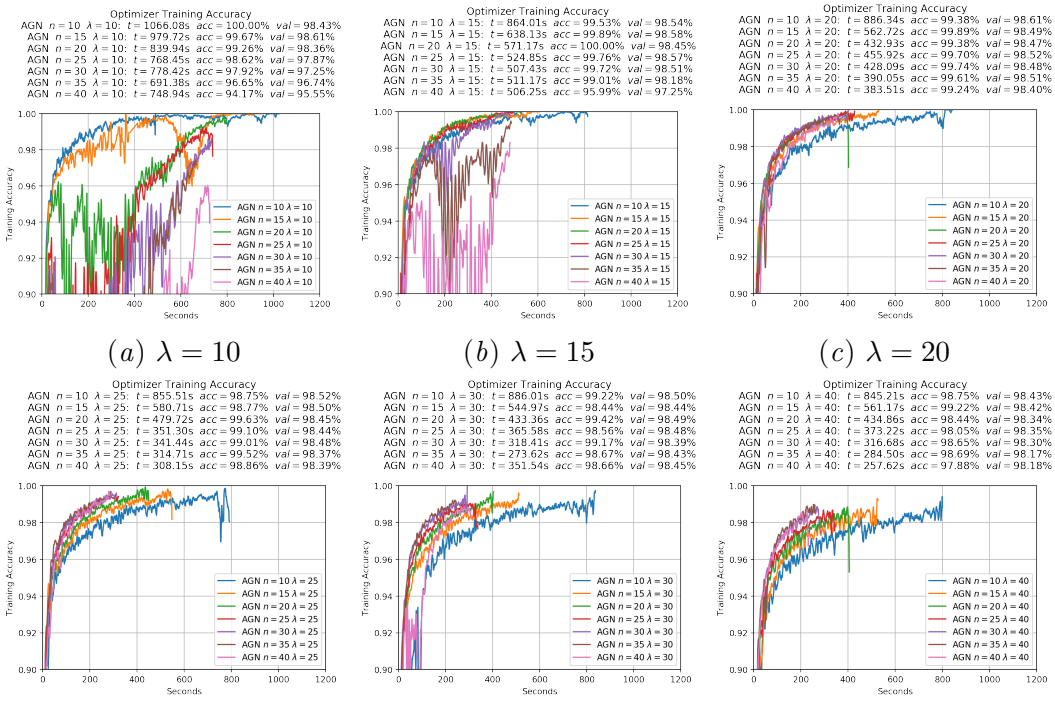
(d) $n = 25$ (e) $n = 30$ (f) $n = 40$

Figure 11: This Figure shows several experiments were we fixed the number of workers, but vary the communication frequency. From this we observe that AGN performs well when a relatively equal high communication frequency is used with respect to the number of workers. Furthermore, increasing the amount of workers, and maintaining a high communication frequency deteriorates the performance of the central variable as well. As a result, a balance between the communication frequency, and the number of asynchronous workers is required.

workers. As a result, AEASGD is rather slow reaching a better training accuracy in the presence of small gradients compared to other optimization algorithms.

n	λ	AGN t	AGN Acc.	AEASGD t	AEASGD Acc.	$\mathcal{E}$(AGN, AEASGD)
10	10	1066.08s	**100.00%**	**953.61s**	99.22%	**1.009**
10	15	864.01s	**99.53%**	**846.86s**	99.38%	**1.012**
10	20	886.34s	**99.38%**	**804.07s**	98.91%	**1.003**
10	25	855.51s	98.75%	**784.46s**	**98.91%**	0.999
10	30	**886.01s**	**99.22%**	930.73s	98.91%	0.988
10	35	850.87s	98.44%	**798.74s**	**99.22%**	0.990
10	40	845.21s	**98.75%**	**791.04s**	97.66%	0.990
20	10	839.94s	**99.26%**	**821.12s**	97.90%	0.983
20	15	**571.17s**	**100.00%**	610.88s	98.52%	**1.018**
20	20	**432.93s**	**99.38%**	510.72s	97.78%	**1.022**
20	25	479.72s	**99.63%**	**421.50s**	97.86%	**1.009**
20	30	433.36s	**99.42%**	**429.16s**	98.36%	**1.007**
20	35	418.83s	**98.52%**	**409.86s**	98.19%	**1.002**
20	40	434.86s	**98.44%**	**420.46s**	97.66%	0.997
40	10	**748.94s**	94.17%	1256.09s	**96.57%**	**1.044**
40	15	**506.25s**	95.99%	534.42s	**96.88%**	0.964
40	20	**383.51s**	**99.24%**	412.37s	96.65%	**1.027**
40	25	**308.15s**	**98.86%**	347.50s	96.65%	**1.025**
40	30	351.54s	**98.66%**	**305.50s**	96.47%	0.997
40	35	279.30s	**98.73%**	**252.70s**	96.32%	**1.009**
40	40	257.62s	**97.88%**	**250.74s**	96.65%	**1.009**

Table 1: Summary of AGN and AEASGD experiments using different distributed hyperparameters (n and λ). From these experiments we find that AGN performs better in terms of training and validation accuracy in the presence of a higher number of asynchronous workers, and a reduced communication frequency. We also include the temporal efficiency of AGN and AEASGD compared to different distributed hyperparameters. Using this information, we can deduce that AGN is outperforming AEASGD in 69.73% of the cases, which is significantly better. Furthermore, this statistic includes cases which are known where AGN is performing badly, i.e., small amount of asynchrony, low communication frequency, and high amount of asynchrony, and high communication frequency.



(a) $\lambda = 10$ (b) $\lambda = 15$ (c) $\lambda = 20$

(d) $\lambda = 25$ (e) $\lambda = 30$ (f) $\lambda = 40$

Figure 12: In this experiment we clamp the communication frequency, but vary the number of asynchronous workers. Due to the equal communication frequency, we can observe good scaling properties of AGN. In most cases doubling the number of workers, reducing the training time by half and is more temporally efficient. However, for larger number of workers $n > 30$ we do not observe a reduction of training time. This is due to the implementation of the parameter server used in our experiments, which is based on Python threads instead of Python processes. Furthermore, note that reducing the amount of computational resources might actually benefit the training accuracy of the central variable, as a smaller number of asynchronous workers reduces the amount of staleness that can be Incorporated in the central variable.

5. Conclusion

This work introduces a novel distributed optimization procedure. We make identical *practical* assumptions with respect to constraints as EASGD, i.e., high communication costs. However, AGN is not effected by equilibrium conditions which incapacitate the converge rate of the optimization process, which are present in EASGD Hermans (2017). As a result, we turned to DOWNPOUR, and allowed for more local exploration by sending *accumulated gradients* to the parameter server. However, this approach diverged even faster than regular DOWNPOUR, with the difference that data was processed significantly faster due to the reduced waits.

Therefore, DOWNPOUR was adapted to use the time between parameter updates more efficiently by computing a *better* gradient based on a normalized sequence of first-order gradients, thus obtaining Accumulated Gradient Normalization. Furthermore, we show that AGN outperforms existing distributed optimizers in terms of convergence rate in the presence of a large amount of concurrency and communication constraints. Since *stability* is also important in distributed optimization, we introduce a new metric called *temporal efficiency* which is defined as the ratio between the integrated area of training metrics of two different optimizers. As a result, not only the final training accuracy is considered, but also the stability, and time required to reach an accuracy level.

To conclude, AGN achieves this result by computing a better *direction* of the gradient based on a sequence of first order gradients which can be computed locally without any communication with the centralized model. This direction also reduces the negative effects of implicit momentum in highly concurrent environment as most workers will point in the general direction of a minimum, thereby making AGN more robust to distributed hyperparameterization such as the number of workers and communication frequency. Although, as the number of workers increases, a decline in training and validation accuracy is still observed due to the amount of implicit momentum. A natural response would be to lower the communication frequency (increase λ). However, this would increase the amount of local exploration with the possibility that a subset of workers might end up in different minima.

For possible future work, it would be of interest to explore if adaptive communication frequencies might benefit the optimization process. In fact, in small gradient environments (e.g., close to a minima) it might be beneficial to *not* normalize the accumulated gradients since the first-order gradient updates are relatively small anyway.

6. Acknowledgements

We would like the team members of CERN IT-DB for supporting this work, especially Zbigniew Baranowski and Luca Canali. We also thank Vlodimir Begy for the insightful discussions. Finally, we thank the Dutch National Science Foundation NWO for financial support through the project SWARMPORT (NWO project number 439.16.108).

References

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. Omnivore: An optimizer for multi-device deep learning on CPUs and GPUs. *arXiv preprint arXiv:1606.04487*, 2016.

Joeri Hermans. On Scalable Deep Learning and Parallelizing Gradient Descent. Master's thesis, Jul 2017. URL http://cds.cern.ch/record/2276711. Presented 06 Jul 2017.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 463–478. ACM, 2017.

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

Yann LeCun, Corinna Cortes, and Christopher JC Burges. The MNIST database of handwritten digits, 1998.

Gilles Louppe and Pierre Geurts. A zealous parallel gradient descent algorithm. *Learning on Cores, Clusters and Clouds workshop, NIPS, 2010*, 2010.

Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, pages 997–1004. IEEE, 2016.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.