Making Asynchronous Stochastic Gradient Descent Work for Transformers

Alham Fikri Aji and **Kenneth Heafield**
School of Informatics, University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB
Scotland, European Union
a.fikri@ed.ac.uk, kheafiel@inf.ed.ac.uk

Abstract

Asynchronous stochastic gradient descent (SGD) converges poorly for Transformer models, so synchronous SGD has become the norm for Transformer training. This is unfortunate because asynchronous SGD is faster at raw training speed since it avoids waiting for synchronization. Moreover, the Transformer model is the basis for state-of-the-art models for several tasks, including machine translation, so training speed matters. To understand why asynchronous SGD under-performs, we blur the lines between asynchronous and synchronous methods. We find that summing several asynchronous updates, rather than applying them immediately, restores convergence behavior. With this method, the Transformer attains the same BLEU score 1.36 times as fast.

1 Introduction

Models based on Transformers (Vaswani et al., 2017) achieve state-of-the-art results on various machine translation tasks (Bojar et al., 2018). Distributed training is crucial to training these models in a reasonable amount of time, with the dominant paradigms being asynchronous or synchronous stochastic gradient descent (SGD). Prior work (Chen et al., 2016, 2018; Ott et al., 2018) commented that asynchronous SGD yields low quality models without elaborating further; we confirm this experimentally in Section 2.1. Rather than abandon asynchronous SGD, we aim to repair convergence.

Asynchronous SGD and synchronous SGD have two key differences: batch size and staleness. Synchronous SGD increases the batch size in proportion to the number of processors because gradients are summed before applying one update. Asynchronous SGD updates with each gradient as it arises, so the batch size is the same as on a single processor. Asynchronous SGD also has stale gradients because parameters may update several times while a gradient is being computed.

To tease apart the impact of batch size and stale gradients, we perform a series of experiments on both recurrent neural networks (RNNs) and Transformers manipulating batch size and injecting staleness. Out experiments show that small batch sizes slightly degrade quality while stale gradients substantially degrade quality.

To restore convergence, we propose a hybrid method that computes gradients asynchronously, sums gradients as they arise, and updates less often. Gradient summing has been applied to increase batch size or reduce communication (Dean et al., 2012; Lian et al., 2015; Ott et al., 2018; Bogoychev et al., 2018); we find it also reduces harmful staleness. In a sense, updating less often increases staleness because gradients are computed with respect to parameters that could have been updated. However, if staleness is measured by the number of intervening updates to the model, then staleness is reduced because updates happen less often. Empirically, our hybrid method converges comparably to synchronous SGD, preserves final model quality, and runs faster because processors are not idle.

2 Exploring Asynchronous SGD

2.1 Baseline: The Problem

To motivate this paper and set baselines, we first measure how poorly Transformers perform when trained with baseline asynchronous SGD (Chen et al., 2016, 2018; Ott et al., 2018). We train a Transformer model under both synchronous and asynchronous SGD, contrasting the results with an RNN model. Moreover, we sweep learning rates to verify this effect is not an artifact of choosing

Proceedings of the 3rd Workshop on Neural Generation and Translation (WNGT 2019), pages 80–89
Hong Kong, China, November 4, 2019.
www.aclweb.org/anthology/D19-56%2d

	Trans. BLEU		RNN BLEU	
Learn Rate	Sync.	Async.	Sync.	Async.
0.0002	35.08	13.27	34.11	33.77
0.0003	35.66	30.72	33.79	33.95
0.00045	35.59	5.21	33.68	33.68
0.0006	35.42	0.00	34.30	33.76
0.0009	34.79	0.00	34.28	33.47
0.0012	33.96	0.00	34.37	33.23
0.0024	29.35	0.00	33.98	32.83
0.00375	25.25	0.00	33.80	31.89

Table 1: Performance of the Transformer and RNN model trained synchronously and asynchronously, across different learning rates.

hyperparameters that favor one scenario. Further experimental setup appears in Section 4.1.

Results in Table 1 confirm that asynchronous SGD generally yields lower-quality systems than synchronous SGD. For Transformers, the asynchronous results are catastrophic, often yielding 0 BLEU. We can also see that Transformers and asynchronous SGD are more sensitive to learning rates compared to RNNs and synchronous SGD.

To understand why asynchronous SGD underperforms, we run series of ablation experiments based on the differences between synchronous and asynchronous SGD. We focus on two main aspects: batch size and stale gradient updates.

2.2 Batch Size

In asynchronous SGD, each update uses a gradient from one processor. Synchronous SGD sums gradients from all processors, which is mathematically equivalent running a larger batch on one processor (though it might not fit in RAM). Therefore, the effective batch size in N-workers synchronous training is N times larger compared to its asynchronous counterparts.

Using a larger batch size reduces noise in estimating the overall gradient (Wang et al., 2013), and has been shown to slightly improve performance (Smith et al., 2017; Popel and Bojar, 2018). To investigate whether small batch sizes are the main issue with asynchronous Transformer training, we sweep batch sizes and compare with synchronous training.

2.3 Gradient Staleness

In asynchronous training, a computed gradient update is applied immediately to the model, without having to wait for other processors to finish. This approach may cause a stale gradient, where parameters have updated while a processor was computing its gradient. Staleness can be defined as the number of updates that occurred between the processor pulling parameters and pushing its gradient. Under the ideal case where every processor spends equal time to process a batch, asynchronous SGD with N processors produces gradients with staleness $N-1$. Empirically, we can also expect an average staleness of $N-1$ with normally distributed computation time (Zhang et al., 2016).

An alternative way to interpret staleness is the distance between the parameters with which the gradient was computed and the parameters being updated by the gradient. Therefore, higher learning rate contributes to the staleness, as the parameters move faster.

Prior work has shown that neural models can still be trained on stale gradients, albeit with potentially slower convergence or a lower quality. Furthermore, Zhang et al. (2016); Srinivasan et al. (2018) report that model performance degrades in proportion to the gradient staleness. We introduce artificial staleness to confirm the significance of gradient staleness towards the Transformer performance.

3 Incremental Updates in Adam

Investigating the effect of batch size and staleness further, we analyze why it makes a difference that gradients computed from the same parameters are applied one at a time (incurring staleness) instead of summed then applied once (as in synchronous SGD). As seen in Section 4.3, our artificial staleness was damaging to convergence even though gradients were synchronously computed with respect to the same parameters. In standard stochastic gradient descent there is no difference: gradients are multiplied by the learning rate then substracted from the parameters in either case. The Adam optimizer handles incremental updates and sums differently.

Adam is scale invariant. For example, suppose that two processors generate gradients 0.5 and 0.5 with respect to the same parameter in the first iteration. Incrementally updating with 0.5 and 0.5 is the same as updating with 1 and 1 due to scale invariance. Updating with the summed gradient, 1, will only move parameters half as far. This is the theory underlying the rule of thumb that learning rate should scale with batch size (Ott et al., 2018).

Time (t)		0	1	2	3	4	5	6
Constant	g_t		1	1	1	1	1	1
	m_t	0	0.1	0.19	0.271	0.344	0.41	0.469
	v_t	0	0.02	0.04	0.059	0.078	0.096	0.114
	$\hat{m}_t$	0	1	1	1	1	1	1
	$\hat{v}_t$	0	1	1	1	1	1	1
	θ	0	-0.001	-0.002	-0.003	-0.004	-0.005	-0.006
Scaled	g_t		0.5	1.5	0.5	1.5	0.5	1.5
	m_t	0	0.05	0.195	0.226	0.353	0.368	0.481
	v_t	0	0.005	0.05	0.054	0.098	0.101	0.144
	$\hat{m}_t$	0	0.5	1.026	0.832	1.026	0.898	1.026
	$\hat{v}_t$	0	0.25	1.26	0.917	1.26	1.05	1.26
	θ	0	-0.001	-0.002	-0.003	-0.004	-0.005	-0.005
Different sign	g_t		-1	2	-1	2	-1	2
	m_t	0	-0.1	0.11	-0.001	0.199	0.079	0.271
	v_t	0	0.02	0.1	0.118	0.195	0.211	0.287
	$\hat{m}_t$	0	-1	0.579	-0.004	0.579	0.193	0.579
	$\hat{v}_t$	0	1	2.515	2	2.515	2.2	2.515
	θ	0	0.001	0.001	0.001	0.000	0.000	-0.000

Table 2: The Adam optimizer slows down when gradients have larger variance even if they have the same average, in this case 1. When alternating between -1 and 2, Adam takes 6 steps before the parameter has the correct sign. Updates can even slow down if gradients point in the same direction but have different scales. The learning rate is $\alpha = 0.001$.

In practice, gradients reported by different processors are usually not the same: they are noisy estimates of the true gradient. In Table 2, we show examples where noise causes Adam to slow down. Summing gradients smooths out some of the noise. Next, we examine the formal basis for this effect.

Formally, Adam estimates the full gradient with an exponentially decaying average m_t of gradients g_t.

$$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

where β_1 is a decay hyperparameter. It also computes a decaying average v_t of second moments

$$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where β_2 is a separate decay hyperparameter. The squaring g_t^2 is taken element-wise. These estimates are biased because the decaying averages were initialized to zero. Adam corrects for the bias to obtain unbiased estimates $\hat{m}_t$ and $\hat{v}_t$.

$$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$$

$$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$$

These estimates are used to update parameters θ

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where α is the learning rate hyperparameter and ϵ prevents element-wise division by zero.

Replacing estimators in the update rule with statistics they estimate and ignoring the usually-minor ϵ

$$\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \approx \frac{Eg_t}{\sqrt{E(g_t^2)}}$$

which expands following the variance identity

$$\frac{Eg_t}{\sqrt{E(g_t^2)}} = \frac{Eg_t}{\sqrt{Var(g_t) + (Eg_t)^2}}$$

Dividing both the numerator and denominator by $|Eg_t|$, we obtain

$$= \frac{\text{sign}(Eg_t)}{\sqrt{Var(g_t)/(Eg_t)^2 + 1}}$$

The term $Var(g_t)/(Eg_t)^2$ is statistical efficiency, the square of coefficient of variation. In other words, Adam gives higher weight to gradients if historical samples have a lower coefficient of variation. The coefficient of variation of a sum of N independent[1] samples decreases as $1/\sqrt{N}$. Hence sums (despite having less frequent updates) may

[1] Batch selection takes compute time into account, so technically noise is not independent.

actually cause Adam to move faster because they have smaller coefficient of variation. An example appears in Table 2: updating with 1 moves faster than individually applying -1 and 2.

4 Ablation Study

We conduct ablation experiments to investigate the poor performance in asynchronous Transformer training for the neural machine translation task.

4.1 Experiment Setup

Our experiments use systems for the WMT 2017 English to German news translation task. The Transformer is standard with six encoder and six decoder layers. The RNN model (Barone et al., 2017) is based on the winning WMT17 submission (Sennrich et al., 2017) with 8 layers. Both models use back-translated monolingual corpora (Sennrich et al., 2016a) and byte-pair encoding (Sennrich et al., 2016b).

We follow the rest of the hyperparameter settings on both Transformer and RNN models as suggested in the papers (Vaswani et al., 2017; Sennrich et al., 2017). Both models were trained on four GPUs with a dynamic batch size of 10 GB per GPU using the Marian toolkit (Junczys-Dowmunt et al., 2018). Both models are trained for 8 epochs or until reaching five continuous validations without loss improvement. Quality is measured on newstest2016 using sacreBLEU (Post, 2018), preserving newstest2017 as test for later experiments. The Transformer's learning rate is linearly warmed up for 16k updates. We apply an inverse square root learning rate decay following Vaswani et al. (2017) for both models. All of these experiments use the Adam optimizer, which has shown to perform well on a variety of tasks (Kingma and Ba, 2014) and was used in the original Transformer paper (Vaswani et al., 2017).

For subsequent experiments, we will use a learning rate of 0.0003 for Transformers and 0.0006 for RNNs. These were near the top in both asynchronous and synchronous settings (Table 1).

4.2 Batch Size

We first explore the effect of batch size towards the model's quality. We use dynamic batching, in which the toolkit fits as many sentences as it can into a fixed amount of memory (so e.g. more sentences will be in a batch if all of them are short). Hence batch sizes are denominated in memory sizes. Our GPUs each have 10 GB available for batches which, on average, corresponds to 250 sentences.

With 4 GPUs, baseline synchronous SGD has an effective batch size of 40 GB, compared to 10 GB in asynchronous. We fill in the two missing scenarios: synchronous SGD with a total effective batch size of 10 GB and asynchronous SGD with a batch size of 40 GB. Because GPU memory is limited, we simulate a larger batch size in asynchronous SGD by locally accumulating gradients in each processor four times before sending the summed gradient to the parameter server (Ott et al., 2018; Bogoychev et al., 2018).

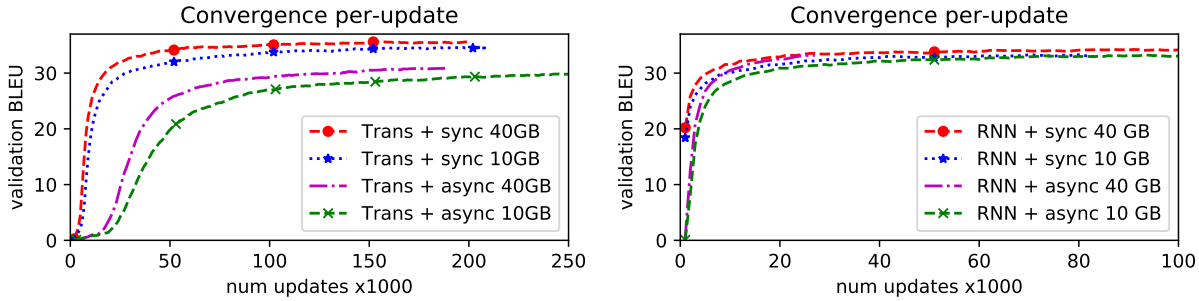
Models with a batch size of 40GB achieve better BLEU *per update*, compared with its 10GB variant as shown in Figure 1. However, synchronous SGD training still outperforms asynchronous SGD training, even with smaller batch size. From this experiment, we conclude that batch size is not the primary driver of poor performance of asynchronously trained Transformers, though it does have some lingering impact on final model quality. For RNNs, batch size and distributed training algorithm had little impact beyond the early stages of training, continuing the theme that Transformers are more sensitive to noisy gradients.

4.3 Gradient Staleness

To study the impact of gradient staleness, we introduce staleness into synchronous SGD. Workers only pull the latest parameter once every U updates, yielding an average staleness of $\frac{(U-1)}{2}$. Since asynchronous SGD has average staleness 3 with $N = 4$ GPUs, we set $U = 7$ to achieve the same average staleness of 3. Additionally, we also tried a lower average staleness of 2 by setting $U = 5$. We also see the effect of doubling the learning rate so the parameter moves twice as far, hence introduces staleness in terms of model distance.

In order to focus on the impact of the staleness, we set the batch size to 40 GB total RAM consumption, be they 4 GPUs with 10 GB each in synchronous SGD or emulated 40 GB batches on each GPU in asynchronous SGD.

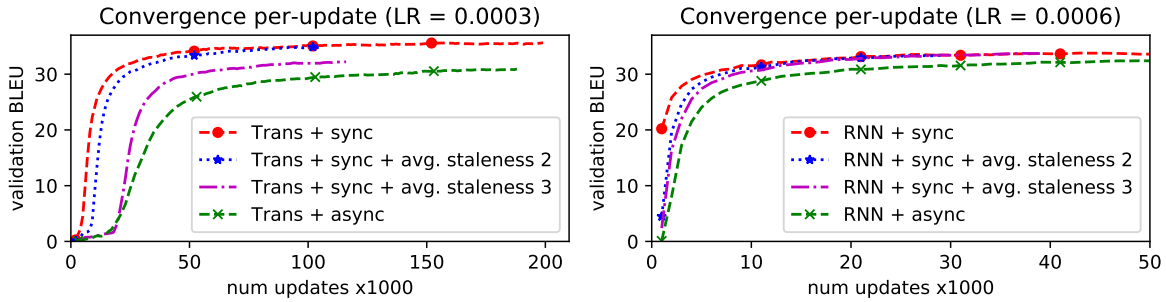
Results are shown in Figure 2. Staleness 3 substantially degrades Transformer convergence and final quality (Figure 2a). However, the impact of staleness 2 is relatively minor. We also continue to see that Transformers are more sensitive than



(a) Convergence over updates in Transformer model with various batch sizes

(b) Convergence over updates in RNN model with various batch sizes

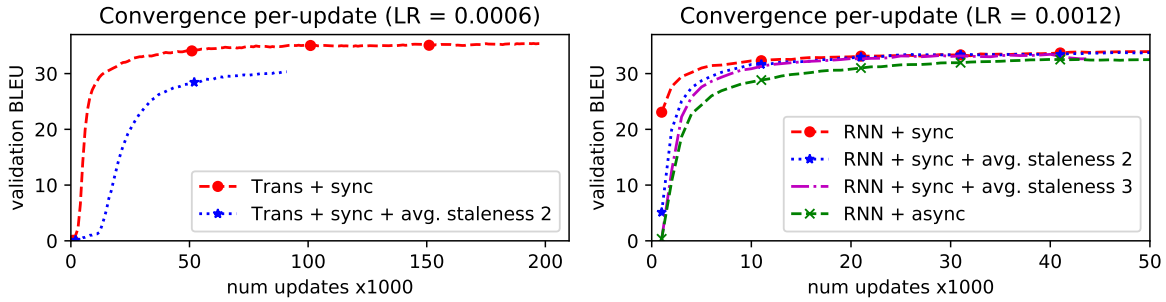
Figure 1: The effect of batch sizes on convergence of Transformer and RNN models.



(a) Transformer model with lr = 0.0003

(b) RNN model with lr = 0.0006

Figure 2: Artificial staleness in synchronous SGD compared to synchronous and asynchronous baselines, all with our usual learning rate for each model.



(a) Transformer model with lr = 0.0006

(b) RNN model with lr = 0.0012

Figure 3: Artificial staleness in synchronous SGD with doubled learning rates. Transformers with learning rate 0.0006 and staleness 3 (synchronous and asynchronous) did not rise above 0.

RNNs to training conditions.

Results for Transformer worsen when we double the learning rate (Figure 3). With staleness 3, the model stayed at 0 BLEU for both synchronous or asynchronous SGD, consistent with our earlier result (Table 1).

We conclude that staleness is primary, but not wholly, responsible for the poor performance of asynchronous SGD in training Transformers. However, asynchronous SGD still underperforms synchronous SGD with artificial staleness of 3 and the same batch size (40 GB). Our synchronous SGD training has consistent parameters across processors, whereas processors might have different parameters in asynchronous training. The staleness distribution might also play a role because staleness in asynchronous SGD follows a normal distribution (Zhang et al., 2016) while our synthetic staleness in synchronous SGD follows a uniform distribution.

5 Asynchronous Transformer Training

5.1 Accumulated Asynchronous SGD

Previous experiments have shown that increasing the batch size and reducing staleness improves the final quality of asynchronous training. Increasing the batch size can be achieved by accumulating gradients before updating. We experiment with variations on three ways to accumulate gradients:

Local Accumulation: Gradients can be accumulated locally in each processor before sending it to the parameter server (Ott et al., 2018; Bogoychev et al., 2018). This approach scales the effective batch size and reduces communication costs as the workers communicate less often. However, this approach does not reduce staleness as the parameter server updates immediately after receiving a gradient. We experiment with accumulating four gradients locally, resulting in 40 GB effective batch size.

Global Accumulation: Each processor sends the computed gradient to the parameter server normally. However, the parameter server holds the gradient and only updates the model after it receives multiple gradients (Dean et al., 2012; Lian et al., 2015). This approach scales the effective batch size. On top of that, it decreases staleness as the parameter server updates less often. However, it does not reduce communication costs. We experiment with accumulating four gradients globally, resulting in 40 GB effective batch size and 0.75 average staleness.

Combined Accumulation: Local and global accumulation can be combined to gain the benefits of both: reduced communication cost and reduced average staleness. In this approach, gradients are accumulated locally in each processor before being sent. The parameter server also waits and accumulates gradients before running an optimizer. We accumulate two gradients both locally and globally. This yields in 40 GB effective batch size and 1.5 average staleness.

We tested the three gradient accumulation flavors on the English-to-German task with both Transformer and RNN models. Synchronous SGD also appears as a baseline. To compare results, we report best BLEU, raw training speed, and time needed to reach several BLEU checkpoints. Results are shown in Table 3.

Asynchronous SGD with global accumulation actually improves the final quality of the model over synchronous SGD, albeit not meaningfully. This one change, accumulating every 4 gradients (the number of GPUs), restores quality in asynchronous methods. It also achieves the fastest time to reach near-convergence BLEU in both Transformer and RNN.

While using local accumulation provides even faster raw speed, the model produces the worst quality among the other accumulation techniques. Asynchronous SGD with 4x local accumulation is essentially just ordinary asynchronous SGD with 4x larger batch size and 4x less update frequency. In particular, gradient staleness is still the same, therefore this does not help the convergence per-update.

Combined accumulation performs somewhat in the middle. It does not converge as fast as asynchronous SGD with full global accumulation but not as poor as asynchronous SGD with full local accumulation. Its speed is also in between, reflecting communication costs.

5.2 Generalization Across Learning Rates

Earlier in Table 1 we show that asynchronous Transformer learning is very sensitive towards the learning rate. In this experiment, we use an asynchronous SGD with global gradient accumulation to train English-to-German on different learning rates. We compare our result with vanilla synchronous and vanilla asynchronous SGD.

Our finding empirically show that asynchronous Transformer training while globally accumulating the gradients is significantly more robust. As shown in Table 5, the model is now capable to learn on higher learning rate and yield comparable results compared to its synchronous variant.

5.3 Generalization Across Languages

To test whether our findings on English-to-German generalize, we train two more translation systems using globally accumulated gradients. Specifically, we train English to Finnish (EN $\rightarrow$ FI) and English to Russian (EN $\rightarrow$ RU) models for the WMT 2018 task (Bojar et al., 2018). We validate our model on newstest2015 for EN $\rightarrow$ FI and newstest2017 for EN $\rightarrow$ RU. Then, we test our model on newstest2017 for EN $\rightarrow$ DE and newstest2018 for both EN $\rightarrow$ FI and EN $\rightarrow$ RU. The same network structures and hyperparameters are used as before.

The results shown in Table 4 empirically confirm that accumulating the gradient to obtain a

Transformer

Communication	accumulation		batch	avg.	speed	best	hours to X BLEU		
	local	global	size	staleness	(wps)	BLEU	33	34	35
synchronous	1	4	40 GB	0	36029	35.66	5.3	7.6	15.6
asynchronous	1	1	10 GB	3	39883	30.72	-	-	-
asynchronous	4	1	40 GB	3	45177	30.98	-	-	-
asynchronous	2	2	40 GB	1.5	43115	35.68	4.9	6.8	15.4
asynchronous	1	4	40 GB	0.75	39514	35.84	4.6	6.7	11.4

RNN

Communication	accumulation		batch	avg.	speed	best	hours to X BLEU		
	local	global	size	staleness	(wps)	BLEU	32	33	34
synchronous	1	4	40 GB	0	23054	34.30	3.6	6.2	18.8
asynchronous	1	1	10 GB	3	24683	33.76	2.7	5.1	-
asynchronous	4	1	40 GB	3	27090	33.83	4.1	6.1	-
asynchronous	2	2	40 GB	1.5	25578	34.20	3.2	5.9	13.7
asynchronous	1	4	40 GB	0.75	24312	34.48	3.1	5.4	14.5

Table 3: Quality and convergence of asynchronous SGD with accumulated gradients on English to German dataset. Dashes indicate that model never reach the target BLEU.

Model	EN → DE		EN → FI		EN → RU	
newstest	2016	2017	2017	2018	2015	2018
Trans. + synchronous SGD	35.66	28.81	18.47	14.03	29.31	25.49
Trans. + asynchronous SGD	30.72	24.68	11.63	8.73	21.12	17.78
Trans. + asynchronous SGD + 4x global accum.	35.84	28.66	18.47	13.78	29.12	25.25
RNN + synchronous SGD	34.30	27.43	16.94	12.75	26.96	23.11
RNN + asynchronous SGD	33.76	26.84	14.94	10.96	26.39	22.48
RNN. + asynchronous SGD + 4x global accum.	34.48	27.56	17.05	12.76	27.15	23.41

Table 4: The effect of global accumulation on translation quality for different language pairs on development and test set, measured with BLEU score.

	Communication		
	Sync.	Async.	Async
Learn Rate			+ 4x GA
0.0003	35.66	30.72	35.84
0.0006	35.42	0.00	35.81
0.0012	33.96	0.00	33.62
0.0024	29.35	0.00	1.20

Table 5: Performance of the asynchronous Transformer on English to German with 4x Global accumulations (GA) across different learning rates on development set measured with BLEU score.

larger batch size and a lower staleness in Transformer massively improves the result, compared to basic asynchronous SGD (+6 BLEU on average). The improvement is smaller in RNN experiment, but still substantial (+1 BLEU on average). We also have further confirmation that training a Transformer model with normal asynchronous SGD is impractical.

6 Related Work

6.1 Gradient Summing

Several papers wait and sum P gradients from different workers as a way to reduce staleness. In Chen et al. (2016), gradients are accumulated from different processors, and whenever the P gradients have been pushed, other processors cancel their process and restart from the beginning. This is relatively wasteful since some computation is thrown out and $P-1$ processors still idle for synchronization. Gupta et al. (2016) suggest that restarting is not necessary but processors still idle waiting for P to finish. Our proposed method follows Lian et al. (2015) in which an update happens every time P gradients have arrived and processors con-

tinually generate gradients without synchronization.

Another direction to overcome stale gradient is to reduce its effect towards the model update. McMahan and Streeter (2014) dynamically adjust the learning rate depending on the staleness. Dutta et al. (2018) suggests completely ignoring stale gradient pushes.

6.2 Increasing Staleness

In the opposite direction, some work has added noise to gradients or increased staleness, typically to cut computational costs. Recht et al. (2011) propose a lock-free asynchronous gradient update. Lossy gradient compression by bit quantization (Seide et al., 2014; Alistarh et al., 2017) or threshold based sparsification (Aji and Heafield, 2017; Lin et al., 2017) also introduce noisy gradient updates. On top of that, these techniques store unsent gradients to be added into the next gradient, increasing staleness for small gradients.

Dean et al. (2012) mention that communication overload can be reduced by reducing gradient pushes and parameter synchronization frequency. In McMahan et al. (2017), each processor independently updates its own local model and periodically synchronize the parameter by averaging across other processors. Ott et al. (2018) accumulates gradients locally, before sending it to the parameter server. Bogoychev et al. (2018) also locally accumulates the gradient, but also updates local parameters in between.

7 Conclusion

We evaluated the behavior of Transformer and RNN models under asynchronous training. We divide our analysis based on two main different aspects in asynchronous training: batch size and stale gradient. Our experimental results show that:

- In general, asynchronous training damages the final BLEU of the NMT model. However, we found that the damage with the Transformer is significantly more severe. In addition, asynchronous training also requires a smaller learning rate to perform well.

- With the same number of processors, asynchronous SGD has a smaller effective batch size. We empirically show that training under a larger batch size setting can slightly improves the convergence. However, the improvement is very minimal. The result in asynchronous Transformer model is subpar, even with a larger batch size.

- Stale gradients play a bigger role in the training performance of asynchronous Transformer. We have shown that the Transformer model's performed poorly by adding a synthetic stale gradient.

Based on these findings, we suggest applying a modification in asynchronous training by accumulating a few gradients (for example for the number of processors) in the server before applying an update. This approach increases the batch size while also reducing the average staleness. We empirically show that this approach combine the high quality training of synchronous SGD and high training speed of asynchronous SGD.

Future works should extend those experiments to different hyper-parameter configurations. One direction is to investigate wether vanilla asynchronous Trasnformer can be trained under different optimizers. Another direction is to experiment with more workers where gradients in asynchronous SGD are more stale.

8 Acknowledgements

Alham Fikri Aji is funded by the Indonesia Endowment Fund for Education scholarship scheme. This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (http://www.csd3.cam.ac.uk/), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

References

Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445.

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720.

Antonio Valerio Miceli Barone, Jindřich Helcl, Rico Sennrich, Barry Haddow, and Alexandra Birch. 2017. Deep architectures for neural machine translation. In *Proceedings of the Second Conference on Machine Translation*, pages 99–107.

Nikolay Bogoychev, Kenneth Heafield, Alham Fikri Aji, and Marcin Junczys-Dowmunt. 2018. Accelerating asynchronous stochastic gradient descent for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2991–2996.

Ondej Bojar, Christian Federmann, Mark Fishel, Yvette Graham, Barry Haddow, Matthias Huck, Philipp Koehn, and Christof Monz. 2018. Findings of the 2018 conference on machine translation (wmt18). In *Proceedings of the Third Conference on Machine Translation (WMT), Volume 2: Shared Task Papers*, pages 272–307. Association for Computational Linguistics.

Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*.

Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, et al. 2018. The best of both worlds: Combining recent advances in neural machine translation. *arXiv preprint arXiv:1804.09849*.

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

Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. 2018. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. *arXiv preprint arXiv:1803.01113*.

Suyog Gupta, Wei Zhang, and Fei Wang. 2016. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 171–180. IEEE.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations*, pages 116–121, Melbourne, Australia. Association for Computational Linguistics.

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

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282.

Brendan McMahan and Matthew Streeter. 2014. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9.

Martin Popel and Ondřej Bojar. 2018. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70.

Matt Post. 2018. A call for clarity in reporting bleu scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191.

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

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. 2017. The University of Edinburgh's neural mt systems for WMT17. In *Proceedings of the Second Conference on Machine Translation*, pages 389–399.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016a. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, Berlin, Germany. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016b. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725.

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

Anand Srinivasan, Ajay Jain, and Parnian Barekatain. 2018. An analysis of the delayed gradients problem in asynchronous SGD.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Chong Wang, Xi Chen, Alexander J Smola, and Eric P Xing. 2013. Variance reduction for stochastic gradient optimization. In *Advances in Neural Information Processing Systems*, pages 181–189.

Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-aware async-sgd for distributed deep learning. In *IJCAI*.