Scaling Distributed Training with Adaptive Summation

Saeed Maleki [1] **Madanlal Musuvathi** [1] **Todd Mytkowicz** [1] **Olli Saarikivi** [1] **Emad Barsoum** [2] **Jaliya Ekanayake** [2] **Vadim Eksarevskiy** [2] **Tianju Xu** [2]

Abstract

Data parallelism is a common way to parallelize stochastic gradient descent (SGD). However, the loss of convergence at large minibatch sizes limits the scalability of data parallelism. This paper introduces a novel method to *combine* gradients called Adasum that significantly improves the convergence when using large minibatches. This paper provides the intuition and formal justification of Adasum along with a convergence proof. Additionally, the paper describes an efficient implementation of Adasum and its integration into the open-source toolkit Horovod for use in both TensorFlow and PyTorch.

The paper empirically shows that Adasum improves convergence when using large minibatch sizes for multiple optimizers (Momentum-SGD, Adam, and LAMB). For BERT-Large training with a minibatch size of 64K, using both Adasum and LAMB training converges in 20% fewer epochs than with LAMB alone. This combination also allows BERT-Large training to scale to a 128K minibatch size. While one of the motivations for LAMB was the inability of the Adam optimizer to scale beyond a minibatch size of 16K, we show that Adasum helps Adam scale BERT-Large training to a 64K minibatch size. Our implementation of Adasum in Horovod has already been adopted in several production environments.

1 Introduction

Recent trends in deep learning demonstrate that increasing model size, coupled with an increase in training data, results in improved model performance. This has led to progressively larger models, such as BERT (Devlin et al., 2019), GPT-2 (Radford et al., 2019), Megatron (Shoeybi et al., 2019), UniLM (Dong et al., 2019), and GPT-3 (Brown et al., 2020). This trend along with the end of Moore's law means that these large models have to be trained on distributed clusters of machines. This in turn means that stochastic gradient descent (SGD) — the dominant training algorithm — must run in parallel.

Data parallelism is a common approach to parallelize SGD. In data parallelism, a *minibatch* of data is distributed evenly among compute nodes and each node computes a gradient on its part before aggregating the gradients in a bulk synchronous parallel (BSP) fashion.[1] Finding the best minibatch size involves a tradeoff between *system efficiency* (number of dataset samples processed per second) and *convergence* (number of dataset samples needed to achieve the desired model accuracy) of SGD. A small minibatch size achieves better convergence by frequently updating the model, but suffers from smaller amount of parallelism. Conversely, a large minibatch size provides more parallelism but converges slower (Keskar et al., 2016; Hoffer et al., 2017; Goyal et al., 2017).

The main contribution of this paper is **Adasum**, an *adaptive summation* technique that improves the convergence when using a large minibatch size, thus enabling higher degrees of parallelism and better system efficiency. Adasum does this by being sensitive to the *orthogonality of gradients* when deciding how to combine them. It averages gradients when they are parallel, adds them when they are orthogonal and smoothly interpolates in between.

The intuition behind Adasum is that it approximates a large minibatch size update with multiple smaller minibatch size updates. This paper shows that Adasum significantly improves convergence of SGD with large minibatch sizes over gradient averaging. This remains true even when using various learning-rate *optimizers*, such as Momentum-SGD (Rumelhart et al., 1986), Adam (Kingma & Ba, 2015), and LAMB (You et al., 2019). When combined with LAMB, which is the state of the art optimizer for BERT-Large, Ada-

[1]Microsoft Research, Redmond, WA, USA [2]Microsoft, Redmond, WA, USA. Correspondence to: Saeed Maleki <saemal@microsoft.com>, Madanlal Musuvathi <madanm@microsoft.com>, Todd Mytkowicz <toddm@microsoft.com>, Olli Saarikivi <olsaarik@microsoft.com>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

[1]There are asynchronous SGD algorithms such as Hogwild! (Recht et al., 2011) and parameter server (Dean et al., 2012) but this paper only focuses on synchronous SGD.

sum converges in 20% fewer epochs than with LAMB alone when using a minibatch size of 64K, the largest minibatch size used in prior work (You et al., 2019). Additionally, we show that LAMB with Adasum converges with 128K minibatch size as well.

One of the motivations for LAMB was the inability to scale BERT-Large with the Adam optimizer to minibatch sizes larger than 16K (You et al., 2019). We show that Adasum together with Adam converges with a 64K minibatch size in this case. This is important as the Adam optimizer remains popular among users despite improvements such as LAMB.

A key benefit of Adasum is that it requires limited to no additional hyperparameter tuning to scale. When the learning rate has been properly tuned for a small minibatch size, Adasum dynamically adjusts the step size for a larger minibatch size based on the orthogonality of the gradients. In contrast, traditional gradient averaging method requires the learning rate to be increased to properly benefit from the increased parallelism while maintaining a similar convergence. Scaling learning rate up linearly with (Goyal et al., 2017) or by square root of (Hoffer et al., 2017; Krizhevsky, 2014) the minibatch size works sometimes, but in general searching for a hyperparameter is required to avoid divergence (Goyal et al., 2017). This is in practice a significant benefit, as the hyperparameter tuning of large models can be prohibitively expensive. Adasum has already been adopted in production environments for the training of state-of-the-art language models as well as in Microsoft Azure AutoML, where it serves customers who do not have the expertise or want to conduct hyperparameter tuning.

A high-performance implementation of Adasum is publicly available for both PyTorch and TensorFlow in the popular open-source distributed training framework Horovod[2]. Users can enable the Adasum feature by passing an extra argument to Horovod. Unlike the allreduce operation, Adasum is not an elementwise operation on the gradients — it requires two passes over the gradient, first to compute the dot product and norm of the gradients and then to use these scalars to do a weighted sum of the gradients. This paper describes a *vector-halving distance-doubling* algorithm for an efficient implementation. Further, the paper describes a mechanism to parallelize the Adasum computation and optimizer weight update that does not require code changes to and is thus agnostic to the underlying optimizer.

In summary, the contributions of this paper are:

- Adasum, a new way to combine gradients that improves the convergence of SGD with unprecedented minibatch sizes.
- A proof of convergence for SGD with Adasum.
- An optimizer agnostic parallelization strategy that significantly speeds up the Adasum computation and optimizer weight-update.
- A detailed discussion of how Adasum is implemented in Horovod, a popular distributed training framework for PyTorch and TensorFlow.
- An evaluation showing Adasum (1) scales minibatch sizes for existing optimizers well beyond what was possible in prior work, (2) converges faster even at minibatch sizes used in prior work, and (3) maintains these convergence benefits even under extensive hyperparameter tuning.

[2]https://github.com/horovod/horovod

2 BACKGROUND

2.1 Stochastic Gradient Descent

Machine learning involves learning a model parameterized by a set of weights w based on some training data. Given a set of training examples $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ where each x_i is a vector representing the input instance i and y_i is its output, training involves finding a w that minimizes the loss function $L = \sum_i Loss(w, x_i, y_i)$, the sum of individual loss function for the model w on input (x_i, y_i). $Loss$ function returns a non-negative scalar value determining how close are y_i and the prediction of the model w with input x_i. For simplicity, we denote $Loss(w, x_i, y_i)$ by $L_i(w)$.

Stochastic gradient descent (SGD) is the most commonly-used technique in training a machine learning model. SGD starts from an randomly initialized model w_0 and progressively updates the model at step i as $w_{i+1} = w_i - \alpha_i g_i(w_i)$. Here α_i is the learning rate at this step as determined by a learning rate schedule, and $g_i(w_i)$ is the gradient of the loss function at w_i computed on a *minibatch*. That is, $g_i(w_i) = \frac{1}{b}\sum_{j=1}^{b} \nabla L_{r_j}(w_i)$ is the sum of the gradients of individual loss functions for a randomly chosen $\{(x_{r_1}, y_{r_1}), \ldots, (x_{r_b}, y_{r_b})\}$ minibatch of size b.

2.2 Tradeoffs with Minibatch Size

In data parallelism, computing the gradient $g_i(w_i)$ for a minibatch is distributed among multiple compute nodes. We now discuss the tradeoffs involved in setting the minibatch size.

Suppose that there are P compute nodes. Each compute node takes b/P portion of the minibatch of size b. We call each portion a *microbatch* whose gradients are all added locally and then accumulated across all nodes. We model the total training time, in seconds, as:

$$train = steps(b) \cdot \Big(comp(\frac{b}{P}) + comm(P)\Big)$$

Here $steps(b)$ gives the number of iterations required to reach desired accuracy with minibatch size b. Additionally, $comp(\frac{b}{P})$ and $comm(P)$ give the time to compute the gradient for a microbatch of size $\frac{b}{P}$ and the time to perform allreduce between P compute nodes, respectively. The *steps* function is model specific, while *comp* and *comm* additionally depend on the hardware.

Convergence is defined by $b \cdot steps(b)$ which is the total number of examples processed during training. It is known that convergence is faster with smaller minibatch size (Keskar et al., 2016). *System efficiency* is defined as $\frac{b}{comp(\frac{b}{P})+comm(P)}$, which is the number of examples processed in a second. If P is kept constant and b is scaled up, the system efficiency is improved but the convergence is negatively affected. Alternatively, if b is kept constant and P is scaled up, the convergence remains constant but the system efficiency up to a certain point is improved until $comm(P)$ dominates the benefit of $comp(\frac{b}{P})$. Therefore, there is a delicate balance on how b should be set for a given P so that the total training time, *train*, is minimized. (Yin et al., 2018) introduces a new measurement called *gradient diversity* which indicates how much b can be scaled without affecting convergence. We introduce a similar measurement to study amount of parallelism dynamically during training in Section 5.1.3.

In SGD there are steep diminishing returns to how much *steps* decreases as b is increased. This loss in convergence is what ultimately limits the scalability of synchronous SGD and is what Adasum addresses.

3 Adasum Algorithm

The main contribution of this paper is **adaptive summation** (or **Adasum** for short) operator that combines gradients. For a b minibatch size, P compute nodes and a $\frac{b}{P}$ microbatch per compute node, Adasum combines the gradients of the microbatches such that it approximates an SGD with a minibatch size of $\frac{b}{p}$. That means using Adasum provides the system efficiency of a b minibatch size SGD while it approximates convergence of a $\frac{b}{P}$ minibatch size SGD.

3.1 Approximating Large Minibatch SGD with Small Minibatch SGD

Consider a scenario with two steps of SGD running on a single compute node with minibatches b_1 and b_2 for the first and second steps, respectively. Starting from w_0, SGD computes:

$$\begin{aligned} w_1 &= w_0 - \alpha \cdot g_1(w_0) \\ w_2 &= w_1 - \alpha \cdot g_2(w_1) \\ \Rightarrow w_{1,2} &= w_0 - \alpha \cdot \big(g_1(w_0) + g_2(w_1)\big) \end{aligned} \tag{1}$$

where the last equation is just the combination of first two. We assumed the same α for both steps for convenience. Now consider a different scenario with one step of SGD with a minibatch of $b_1 \cup b_2$ running on two compute nodes where b_1 and b_2 are distributed between them. SGD computes:

$$w_1 = w_0 - \alpha \cdot \frac{\big(g_1(w_0) + g_2(w_0)\big)}{2} \tag{2}$$

where $g_1(w_0)$ and $g_2(w_0)$ are the gradients from w_0 for b_1 and b_2, respectively. Comparing Equations 1 and 2 shows why scaling up the learning rate is popular with larger minibatch sizes (Goyal et al., 2017). For example, by doubling the learning rate in Equation 2, the only difference between the updated models in these two scenarios is $g_2(w_1)$ and $g_2(w_0)$. The rest of this section describes how Adasum approximates $g_2(w_1)$, with readily available $g_2(w_0)$ and $g_1(w_0)$.

As previously observed (Zheng et al., 2016; Maleki et al., 2018), one can use second order reasoning to approximate $g_2(w_1)$ as follows:

$$\begin{aligned} g_2(w_1) &= g_2(w_0 - \alpha \cdot g_1(w_0)) \\ &= g_2(w_0) - \alpha \cdot \frac{\partial g_2}{\partial w}\Big|_{w_0} \cdot g_1(w_0) + O(\alpha^2 \left\|g_1(w_0)\right\|^2) \\ &\approx g_2(w_0) - \alpha \cdot H_2 \cdot g_1(w_0) \end{aligned} \tag{3}$$

where H_2 is the Hessian matrix of the loss function at w_0 for b_2 and $H_2 \cdot g_1(w_0)$ is a matrix-vector product. The approximation comes from neglecting the higher order terms as α and $\|g_1(w_0)\|$ tends to decay as the training progresses. Refer to Appendix A.1 for a detailed discussion.

Using a standard theorem (Hastie et al., 2001; Reid, 2012) for estimating the Hessian matrix for negative log-likelihood loss (details in Appendix A.1), H_2 can be estimated by $g_2(w_0) \cdot g_2^T(w_0)$, the outer product of $g_2(w_0)$ by itself, and therefore,

$$g_2(w_1) \approx g_2(w_0) - \alpha \cdot g_2(w_0) \cdot g_2(w_0)^T \cdot g_1(w_0) \tag{4}$$

Let's assume that α was chosen optimally in Equation 1 which means that $\alpha \sim \frac{1}{\|g_1(w_0)\|^2}, \frac{1}{\|g_2(w_0)\|^2}$ (refer to Appendix A.2 for details). We also assume that $\|g_1(w_0)\| \approx \|g_2(w_0)\|$. Combining all approximations produces:

$$\begin{aligned} g_2(w_1) &\approx g_2(w_0) - \frac{g_2(w_0) \cdot \big(g_2(w_0)^T \cdot g_1(w_0)\big)}{\|g_2(w_0)\|^2} \\ &= \Big(1 - \frac{g_2(w_0)^T \cdot g_1(w_0)}{\|g_2(w)\|^2}\Big) g_2(w_0) \end{aligned} \tag{5}$$

For the rest of this paper, we drop w_0 from $g_1(w_0)$ and $g_2(w_0)$ when the model is known from the context. Using Approximation 5 in Equation 1, $w_{1,2}$ is approximated by:

$$w_{1,2} \approx w_0 - \alpha \cdot \Big[g_1 + \Big(1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2}\Big) \cdot g_2\Big] \tag{6}$$

3.2 Sampling Multiple Paths

The approximation above provides an intriguing possibility. SGD is a stochastic process that samples a *path* defined by the order of the training data it processes. For instance, if SGD had processed minibatch b_2 before b_1, the final model could be approximated as

$$w_{2,1} \approx w_0 - \alpha \cdot \left[g_2 + \left(1 - \frac{g_2^T \cdot g_1}{\|g_1\|^2}\right) \cdot g_1 \right] \tag{7}$$

By combining Equation 6 and Equation 7 one can sample both paths $1, 2$ and $2, 1$ by averaging $w_{1,2}$ and $w_{2,1}$. This ability to sample multiple paths motivates the definition of the Adasum operation.

3.3 Adasum operation

We define the Adasum operation as

$$\mathbb{AS}(g_1, g_2) \triangleq \left(1 - \frac{g_2^T \cdot g_1}{2 \cdot \|g_1\|^2}\right) \cdot g_1 + \left(1 - \frac{g_2^T \cdot g_1}{2 \cdot \|g_2\|^2}\right) \cdot g_2 \tag{8}$$

From Equation 6, Equation 7, and Equation 8 we can see that we can average $w_{1,2}$ and $w_{2,1}$ by using the gradients combined with Adasum to update the model:

$$\frac{w_{1,2} + w_{2,1}}{2} = w_0 - \alpha \cdot \mathbb{AS}(g_1, g_2)$$

Before extending Adasum to more than two gradients below, we study two properties of the Adasum operation above. When g_1 and g_2 are orthogonal, their dot product $g_2^T \cdot g_1$ is zero. Therefore, Adasum simply adds the two gradients. When g_1 and g_2 are parallel, their dot product is the product of their norms and Adasum becomes the average of the two gradients. Intuitively, when the two gradients are pointing in orthogonal directions, Adasum behaves as if their loss functions are locally independent and aggressively sums the two gradients. Doing so when the two gradients are parallel has the danger of "overshooting" the minimum, particularly when the learning rate is also aggressive and therefore, Adasum safely averages the gradients. This adaptiveness becomes important as we later show that gradients tend to point in the same direction during the initial parts of the training. This is because the initial model is completely random and all gradients agree on the general direction model should progress. However, the gradients become progressively orthogonal in later parts of the training. Adasum automatically and adaptively interpolates between an aggressive sum and a safe average as training proceeds.

3.4 Combining More Than Two Gradients

We can extend Adasum to more than two gradients by recursively applying the operator. Let $g_{[i,j)}$ be the gradients corresponding to minibatches $b_i \ldots b_{j-1}$ (note that in $[i, j)$, i is included and j is excluded). Adasum operator combines $g_{[0,n)}$ in two ways and in both cases, similar to sequential SGD, the order in which samples and gradients are picked affects the output. The first way that Adasum operator combines gradients is as follows:

$$\mathbb{AS}(g_{[0,n)}) = \mathbb{AS}(\mathbb{AS}(g_{[0,n-1)}), g_n)$$

As explained above, as each application of the Adasum operation doubles the number of SGD paths samples, we achieve the effect of sampling exponentially many SGD paths.

One can reuse the standard ring algorithm for allreduce to implement the Adasum operation. One complexity is that Adasum operator is not a point-wise computation due to the inner product and norm computations. Therefore, it cannot be performed in a pipelining manner as it requires two passes of the data. In the second way, we implement a more efficient algorithm that uses the following recursive application:

$$\mathbb{AS}(g_{[0,n)}) = \mathbb{AS}\big(\mathbb{AS}(g_{[0,n/2)}), \mathbb{AS}(g_{[n/2,n)})\big) \tag{9}$$

The resulting ADASUMRVHDD algorithm is shown in Algorithm 1. ADASUMRVHDD uses a modified recursive vector-halving distance-doubling (RVHDD) algorithm for allreduce (Vandegeijn, 1994; Chan et al., 2007), which is both latency and bandwidth optimal in a switch based networks.

ADASUMRVHDD in Algorithm 1 is called in a single program multiple data (SPMD) fashion where *size* number of compute nodes call the same function with $rank \in \{0, \ldots, size - 1\}$ used as their identification. The function is called with x set to each compute node's gradient and $d = 1$. ADASUMRVHDD algorithm performs the Adasum operation in two phases, a reduce-scatter phase (lines 1-1) followed by an allgather phase (lines 1-1).

Algorithm 1 starts with ranks exchanging half of their vector x with a neighbor at distance $d = 1$ (lines 1-1). Here the two halves a and b are assigned such the left neighbor's half is in a and the right neighbor's half is in b.

Lines 1-1 of Algorithm 1 represent the main modification to baseline RVH algorithm. First, on line 1, each rank calculates a dot product and squared norms for a and b, which are slices of a larger logical vector shared across exactly the ranks in *group* (line 1) which in the first iteration is of size 2. Line 1 then sums the products among the ranks in *group* to produce the complete results in v. The reduction is finally applied locally using the values in v (line 1). Now each group logically has the result of an Adasum operator scattered among the members.

Algorithm 1 Recursive vector-halving distance-doubling with Adasum

```
Require: $size > 2$ is a power-of-two.
function AdasumRVHDD($x, d$)
    $mid = \lfloor |x|/2 \rfloor$
    if $\lfloor rank/d \rfloor$ is even then
        $nghr = rank + d$ {Left neighbor}
        Send($x_{mid:|x|}, nghr$) {Send right half}
        $a = x_{0:mid}$
        $b = $ Recv($nghr$) {Receive left half}
    else
        $nghr = rank - d$ {Right neighbor}
        Send($x_{0:mid}, nghr$) {Send left half}
        $a = $ Recv($nghr$) {Receive right half}
        $b = x_{mid:|x|}$
    end if
    $d' = 2 \cdot d$
    $v = [a \cdot b, a \cdot a, b \cdot b]$ {Partial dot products}
    $group = [\lfloor \frac{rank}{d'} \rfloor \cdot d' + i$ for $i = 0 \,..\, d' - 1]$
    $v = $ AllReduce($v, +, group$) {Finish dot products}
    $x' = a \cdot (1 - \frac{v_1}{2v_2}) + b \cdot (1 - \frac{v_1}{2v_3})$ {Apply Adasum }
    if $d' < size$ then
        $x' = $ AdasumRVHDD($x', d'$)
    end if
    Send($x', nghr$) {Send my half}
    $y = $ Recv($nghr$) {Receive neighbor's half}
    $x = x' \mathbin{+\!\!+} y$ if $\lfloor rank/d \rfloor$ is even else $y \mathbin{+\!\!+} x'$
end function
```

The function is then recursively is called by doubling the distance on line 1. Now every 2 consecutive groups work together to reduce another pair of vectors using Adasum operator. This continues until one logical vector is scattered among the group of all compute nodes.

The second phase of the algorithm recursively exchanges slices of the final vector so that each compute node has a copy of full vector on lines 1-1.

3.5 Layer-Wise Adasum

In practice, ML models have multiple layers. We apply Adasum *per layer*, as opposed to the whole gradient. This ensures Adasum adaptively interpolates between an average and sum, based on per-layer orthogonality. Section 5.1.3 empirically validates this decision on both ResNet-50 and BERT-Large.

3.6 Convergence Proof

In this Section, we provide the skeleton of Adasum convergence proof and we defer most of the details to Appendix. SGD uses a minibatch of size b which is usually much smaller than n, the total number of training examples, to estimate the *true gradient*, the gradient of all training examples. Let's denote the true gradient with $G(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla L_i(w)$ and the gradient of a randomly sample minibatch of size b with $g_b(w) = \frac{1}{b}\sum_{i=1}^{b} \nabla L_{r_i}(w)$. Because of the uniformly randomly selected minibatch, in expectation, they are equal: $\mathbb{E}(g_b(w)) = G(w)$. (Polyak & Tsypkin, 1973) argues that any SGD-like algorithm requires the update rule to follow the *pseudogradient* property. The details of Theorem is discussed in Appendix A.3. However, the most important properties that Adasum needs to satisfy to qualify to be a pseudogradient are: (1) $G(w)^T \cdot \mathbb{E}(\mathbb{AS}(g_{b_1}(w), g_{b_2}(w))) > 0$ for independently randomly chosen b_1 and b_2 and non-optimal ws, and (2) $\|\mathbb{E}(\mathbb{AS}(g_{b_1}(w), g_{b_2}(w)))\| \le c\,\|G(w)\|$ for some constant c.

We prove the first required property using Lemma A.2 in Appendix A.3. Lemma A.2 claims That the angle between $\mathbb{E}(\mathbb{AS}(g_{b_1}(w), g_{b_2}(w)))$ and $G(w)$ is smaller than 0.108π. Any two vectors with an angle smaller than $\frac{\pi}{2}$ have a positive inner product which satisfies the first requirement. Adasum operator is applied on $\log P$ steps on P gradients as shown in Algorithm 1 where P is the number of compute nodes. In Appendix A.3, we argue that under safe assumptions the first property of positive inner product is still true for arbitrary number of Adasum operators.

We prove the second required property using Lemma A.3 in Appendix A.3. Lemma A.3 claims that $\mathbb{E}(\|\mathbb{AS}(g_{b_1}(w), g_{b_2}(w))\|) \le 2\,\|G(w)\|$. Since Adasum operator is applied in $\log P$ levels, expected value of the norm of the final vector must be smaller that $2^{\log P}\,\|G(w)\| = P\,\|G(w)\|$ which satisfies the second property of pseudogradient.

4 Horovod Integration

Adasum is implemented in Horovod (hor) and is publicly available in the main branch of its open-source repository. Horovod integrates with multiple machine learning frameworks, such as TensorFlow and PyTorch, and targets multiple backend transports, including Ethernet/IB and NVLink.

Adasum can be enabled by specifying an option to the `DistributedOptimizer` API of Horovod as follows.

```
opt = hvd.DistributedOptimizer(opt, op=hvd.Adasum)
```

For more fine grained control, we also expose the Adasum operator through Horovod's allreduce as follows.

```
  hvd.allreduce(opt, op=hvd.Adasum)
```

This is useful when users want to perform additional operations such as gradient clipping beyond those implemented in a `DistributedOptimizer`.

One subtlety is that the Adasum operation should be performed on the model update *after* the optimizer has been applied. Intuitively, this is because Adasum emulates the behavior of smaller minibatch sizes and the logic of optimizers should apply to these smaller minibatches before the allreduce. See Appendix A.6 for more details. This is handled transparently through the `DistributedOptimizer` API, but must be manually implemented when using `hvd.allreduce`.

Parallelizing Adasum Computation For large models such as BERT-Large, memory available in a GPU only fits a small microbatch size. In such cases, to increase the effective microbatch size, we use the GPUs available in a single node to accumulate local gradients and use the Adasum operation across nodes. This scenario also allows the following parallelization of the Adasum computation across the local GPUs.

Our approach is inspired by the optimizer-state partitioning algorithm pioneered by Marian (mar) where the insight is that optimizer parameters are identical for all GPUs and thus it is not necessary to replicate them. We use this same insight to parallelize the Adasum computation as follows. For each layer its gradient is reduced inside a node onto a single GPU, on which the optimizer is applied. ADASUMRVHDD is then applied across GPUs holding the same layer on different nodes. The model update is finally broadcast to all other GPUs inside the same node. A key difference between the Marian approach and ours is that rather than distributing this state uniformly, our partitioning never splits layers, which greatly simplifies the implementation and works without requiring changes to the underlying optimizer.

Low Precision Support Our implementation of Adasum integrates with the low-precision support in Horovod. There are two subtleties in our implementation. First, Adasum computes the dot product and norms of the combined gradients using a `double` precision accumulator for numerical stability. Second, we use *dynamic scaling* (Micikevicius et al., 2017) to scale gradient calculations into the dynamic range of the low-precision format.

CPU and GPU Vectorization Adasum runs on both CPU and GPU hardware in `fp16`, `fp32`, and `fp64`. For CPU hardware, we manually vectorize loop bodies that perform both dot products and summations. When Horovod is compiled with CUDA aware MPI, we implement these same loops as GPU kernel calls that operate directly on GPU memory and thus save on the transfer from GPU to CPU. This is particularly important on hardware that supports GPUDirect RDMA as GPU memory need not be copied to the CPU for the Adasum operator.

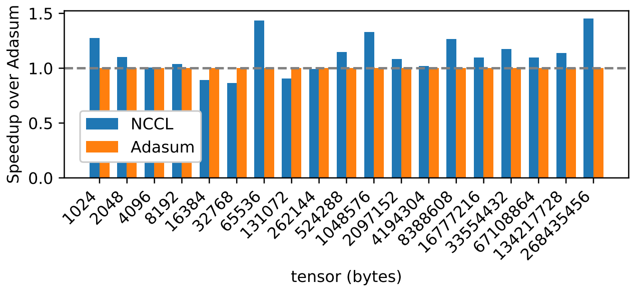


Figure 1. Latency of ADASUMRVHDD vs. NCCL for various message sizes.

	Without	With
Microbatch size fitting on a GPU	22	36
#Samples/s without model update	618.8	674.0
Model update time (s)	1.82	0.97

Table 1. Performance improvement of Adasum parallelization with 4 GPUs.

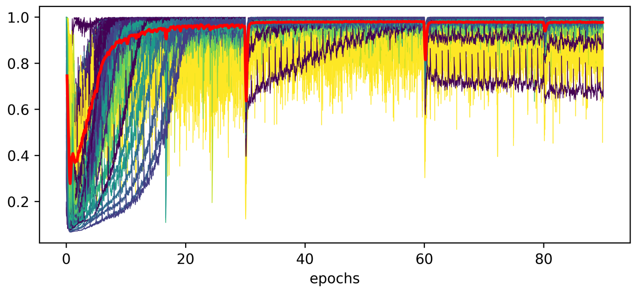
5 RESULTS

This section validates the design and our implementation of Adasum and evaluates it on a variety of real-world training scenarios. Section 5.1 describes standalone experiments that investigate the algorithmic and performance aspects of Adasum. Section 5.2 then presents a sequence of case studies that demonstrate the convergence benefits of Adasum and evaluate its sensitivity to hyperparameter tuning.

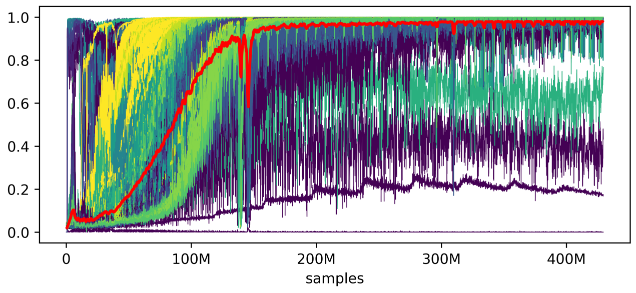
5.1 Standalone Experiments

5.1.1 ADASUMRVHDD *Performance*

We first measure the performance of the core Adasum computation. Unlike standard allreduce that requires one pass over the gradient buffers, Adasum requires two passes to compute the dot product and norm of gradients before doing the adaptive sum. Nevertheless, we show that Adasum can be efficiently implemented. Figure 1 shows the latency of ADASUMRVHDD when compared to NCCL allreduce evaluated on 16 Azure `Standard_NC24rs_v3` nodes with 4 V100s per node (PCIe interconnect) connected by 100 Gb/s Infiniband connection. The x axis is the size of the input tensor (measured in bytes), which is equally distributed across 64 GPUs. The figure shows Adasum performance is competitive with NCCL allreduce (for sizes up to 256 MB) with overhead less than 45% but sometimes outperforming by up to 15%. Note, Horovod fuses gradients from layers in a buffer of size 64 MB before performing allreduce. As communication is only a part of gradient computations during training, this overhead is masked when comparing end to end system efficiency. Section 5.2 delves deeper into these aspects.



(a) ResNet-50



(b) BERT-Large

Figure 2. Orthogonality of gradients for ResNet-50 (a) and BERT-Large (b). A value of 1 in the y-axis means the gradients are orthogonal. The bold red lines show orthogonality averaged across all layers. Other lines show the orthogonality of individual layers.

5.1.2 *Parallelization Performance*

Another crucial optimization is the parallelization of Adasum operation described in Section 4. Table 1 above shows the performance improvement given by this optimization for a PyTorch implementation of BERT-Large on an Azure `Standard_NC24rs_v3` node (4 GPUs) evaluated during the first phase of BERT-Large (128 max sequence length). Since this optimization reduces the memory usage and improves the model update time, we broke down the benefit of it in two folds: number of samples processed without model update and model update time by itself. First, we increased the microbatch size that fits in a GPU by 60% as shown in the first row using the extra memory. We disabled the model update and measured the total number of examples that the 4 GPUs can compute a gradient for per second which is shown in the second row. As it can be seen, this provides 10% improvement for pure gradient computation. Finally, we disabled the gradient computation and only measured the model update time in the last row of Table 1 which shows that a nearly $1.87\times$ improvement. The overall system efficiency benefits from both of these improvements and it depends on the minibatch size used.

5.1.3 *Per-Layer Orthogonality*

The core intuition of Adasum is to adaptively combine the gradients based on their orthogonality — when gradients are orthogonal to each other, Adasum adds the gradients and when gradients are parallel, Adasum averages the gradients. (Yin et al., 2018) introduces *gradient diversity* and shows that there is a direct correlation between it and the largest minibatch size that can be used with SGD without negatively affecting convergence. Gradient diversity is defined by $\frac{\sum_{i=1}^{n}\|g_i\|^2}{\left\|\sum_{i=1}^{n} g_i\right\|^2}$. (Yin et al., 2018) claims that when the gradient diversity is high (for example when all gradients are orthogonal, diversity is 1), the convergence rate of SGD with minibatch size n is on a par with a minibatch size of 1. On the other hand, when all gradients are the same, diversity is $\frac{1}{n^2}$ and convergence of SGD is poor. Instead of gradient diversity, we introduce a similar fraction but for gradients from gradients from each layer which computes $\frac{\left\|Adasum(g_{[1,n]})\right\|^2}{\sum_{i=1}^{n}\|g_i\|^2}$. This measure is in $[0, 1]$ and the closer it is to 1, the closer Adasum is to summing the gradients instead of averaging. The two measurements represent the same property at the two extreme cases.

Our hypothesis is that as the training proceeds, the gradients become orthogonal to each other, allowing Adasum to aggressively sum the gradients instead of average, thus enabling better convergence. To investigate this hypothesis, we periodically measure our fraction for each layer during training. Figure 2 plots this measure during the training of ResNet-50 (a) and BERT-Large (b) with 64 GPUs. The bold red line shows the average of the measure for all layers. Clearly, this lines show that gradients on average tend to become orthogonal as the training proceeds both for ResNet-50 and BERT-Large. Other colors plot the measure for individual layers. With many layers, one cannot discern the behavior of each layer individually. However, general trends show that most layers become orthogonal as the training proceeds, but do so at different rates. This suggests the effect of Adasum cannot be easily achieved with appropriately designed learning rate schedulers. This discrepancy is more visible for BERT-Large, where some layers have low orthogonality throughout the training process. Note there are clear drops in the orthogonality during the training for both benchmarks. These drops happens exactly at boundaries of learning rate schedule change.

5.2 Convergence Results

This section demonstrates the improvement in convergence results for BERT-Large and ResNet-50. First, we describe the key convergence results:

Does Adasum enable faster convergence? For BERT-Large, Adasum with LAMB converges in 20% fewer epochs than with LAMB alone when using a batch size of 64K (see Section 5.2.1). Likewise, for ResNet-50, which uses the Momentum optimizer, Adasum converges with a 16K minibatch size without hyperparameter tuning while allreduce

Algorithm	Number of steps Phase 1	Phase 2
Baseline-Adam	-	-
Baseline-LAMB (You et al., 2019)	7039	1563
Adasum-Adam	7039	1563
Adasum-LAMB - 20%	5639	1250
Adasum-LAMB - 30%	5039	1563
Adasum-LAMB - 128K	4574	1563

Table 2. Convergence results on BERT-Large. Table shows the number of steps required for Phase 1 and Phase 2 to achieve target SQuAD score of 90.5, when using the minibatch size of 64K for Phase 1 (with the exception of last row) and 32K for Phase 2.

	PH1 speedup		PH2 speedup		Time (minutes)	
GPUs	Allreduce	Adasum	Allreduce	Adasum	Sum	Adasum
64	1	0.98	1	0.99	997	809
256	3.79	3.61	3.89	3.92	260	214
512	7.47	6.48	7.24	7.28	135	118

Table 3. System efficiency on BERT-Large for an minibatch size of 64K and 32K for phase 1 and phase 2, respectively. The speedup numbers are relative to the throughput of Baseline-LAMB with 64 GPUs, which is 12.2K examples per second for Phase 1 and 4.6K examples per second for Phase 2. The improved convergence time of Adasum is a result of the 20% improvement in convergence as shown in Table 2.

does not.

Does Adasum enable larger minibatch sizes than prior art? For BERT-Large we show Adasum scales the Adam optimizer to a 64K minibatch size while the inability of Adam to scale beyond 16K was the primary motivation for optimizers like LAMB (You et al., 2019). Likewise, we show that Adasum converges with a 128K minibatch size for LAMB (See Section 5.2.1).

How sensitive is the benefit from Adasum to hyperparameter tuning? For LeNet-5, we show that even with extensive hyperparameter search over the learning rate, Adasum still converges faster than allreduce (see Section 5.2.3).

5.2.1 BERT-Large

This section shows Adasum scales both Adam (Kingma & Ba, 2014) and LAMB (You et al., 2019) for the PyTorch NVIDIA implementation of BERT-Large (nvi). For the Adasum implementation, we replaced its use of `torch.distributed` with the Adasum operator in Horovod. We trained to a target F1 score of SQuAD 1.1 of 90.5 averaged over 5 tries with different seeds (You et al., 2019; Devlin et al., 2019).

The system is a cluster of DGX-2 nodes where each node has 16 V100 GPUs with 32GB of memory per GPU connected by NVSwitch. Each node has 8 NICs with Infiniband support capable of delivering a throughput of 800GB/s per node.

Convergence Table 2 describes the convergence of Adasum over the Adam and LAMB optimizer when using a minibatch size of 64K for Phase 1 and 32K for Phase 2. As reported in prior work, the Adam optimizer does not scale to a minibatch size beyond 16K(You et al., 2019). This motivated the study of more sophisticated optimizers such as LARS and LAMB. For instance, our runs of the LAMB optimizer (without Adasum) achieve the target SQuAD score with 7039 steps of phase 1 and 1563 steps of phase 2, as shown in second row of Table 2.

The next two rows of Table 2 show the performance of Adasum. In contrast to Adam baseline, the Adasum-Adam optimizer converges with 64K when run with the same number steps for Phase 1 and Phase 2 as the LAMB baseline. Despite the advances of optimizers such as LAMB, the Adam optimizer continues to be popular for some models. When compared to prior work (You et al., 2019), it is important to note that Adasum adds no additional hyperparameters and simply uses the baseline parameters of the Adam optimizer. On the other hand, improvements provided by Adasum are orthogonal to improvements in optimizers. As shown in Table 2, Adasum-LAMB provides 20% faster convergence compared to the LAMB baseline, requiring 5639 steps for phase 1 and 1250 steps phase 2.

We also performed two variations of our Adasum-LAMB results. First, we aggressively reduce the number of Phase 1 steps by 30%. With an equivalent aggressive reduction on Phase 2, we slightly missed the target SQuAD score by 0.5. However, we did achieve the target accuracy with the full 1563 steps in Phase 2, which is what we report in the table. With a more fine grained search for Phase 2 steps, we believe we can achieve the target SQuAD score with fewer Phase 2 steps. This is a great result given Phase 1 takes a larger percentage of training time than Phase 2.

For the second variation, we increased the minibatch size of Phase 1 to 128K. We were able to achieve the target SQuAD score with 4574 steps in Phase 1, while using the standard 1563 steps of Phase 2 with 32K minibatch size. Note that this convergence of Phase 1 with 128K minibatch size requires $128K \times 4574 = 585,472K$ samples ($steps(b) \cdot b$ for convergence definition in Section 2), while the default LAMB requires $64K \times 7039 = 450,496K$ samples. This 30% increase could be compensated for by the potential $2\times$ increase in the amount of parallelism available with a larger number of GPUs. To the best of our knowledge, this is the largest reported minibatch size for BERT-Large. In contrast, despite our best efforts, LAMB alone with 128K minibatch size requires around 7000 steps for Phase 1, which negates any potential speedup from the $2\times$ larger minibatch size.

System Efficiency Table 3 shows the speedup of Adasum-LAMB when compared to Baseline-LAMB for a minibatch size of 64K. On our GPU cluster, the Baseline-LAMB processes 12.2K examples per second during Phase 1 and 4.6K examples per second during Phase 2. This reduction in throughput arises because Phase 2 has more computation due to increases sequence length. The speedup numbers in the table are relative to this baseline. For instance, at 256 GPUs, the baseline scales to a speedup of 3.789 (a perfect scaling would be 4). Just as a comparison with published NVIDIA numbers (nvi), the baseline finishes BERT-Large in 260 minutes which is slower than 236 minutes reported by NVIDIA. This difference is due to the NVIDA cluster having a slightly more performant DGX-2H configuration using a higher clock speed compared to our DGX-2 cluster.

Though Adasum performs more computation during allreduce, the reduction in throughput for 64 GPUs is less than 2% for Phase 1 and less than 1% for Phase 2. As we increase the number of GPUs, the additional computation results in lower scaling efficiency for Phase 1. For instance, Adasum incurs roughly a 5 % reduction (13% reduction) in throughput when compared to the baseline for Phase 1 on 256 (512) GPUs. On the other hand, the throughput for Phase 2 with Adasum shows similar scalability as we increase the number of GPUs, with Adasum being faster sometimes. This is due to the fact that Phase 2 does more computation.

The slight reduction in system efficiency is more than compensated by the 20 % reduction in convergence. As such, Adasum achieves faster time to accuracy than the baseline. In particular, Adasum completes BERT-Large in 214 minutes, which is faster than NVIDA reported numbers on 256 GPUs (nvi) and 20% faster than our own baseline run on the same hardware. On 512 GPUs, Adasum reaches the desired accuracy in 118 minutes. We observed some of the overhead is due to CUDA aware MPI (OpenMpi + UCX) was not as fast as NCCL. We are in the process of porting Adasum's allreduce to NCCL as a consequence.

5.2.2 *ResNet-50*

This section evaluates Adasum on PyTorch's ResNet-50 (He et al., 2016) on Imagenet (Russakovsky et al., 2015) using the Momentum-SGD optimizer. We modified PyTorch's model to run with Horovod and compared the performance of Adasum with Horovod's `Sum` operator. We ran experiments on Azure's `Standard_NC24rs_v3` virtual machines, each of which has 4 NVIDIA Tesla V100 GPUs connected with PCIe with 16GiB of memory for each, dual-socket Intel Xeon E5-2690 v4 CPUs, 448 GiB of memory, and connected via Infiniband. We train on 64 V100s with 2K and 16K examples per allreduce and use the default hyperparameters that ship with the benchmark for its momentum based SGD optimizer.

	Allreduce		Adasum	
	2K	16K	2K	16K
Number of epochs	62	-	62	69
Time per epoch (min)	5.61	2.12	5.72	2.23

Table 4. Convergence and system efficiency for Resnet50.

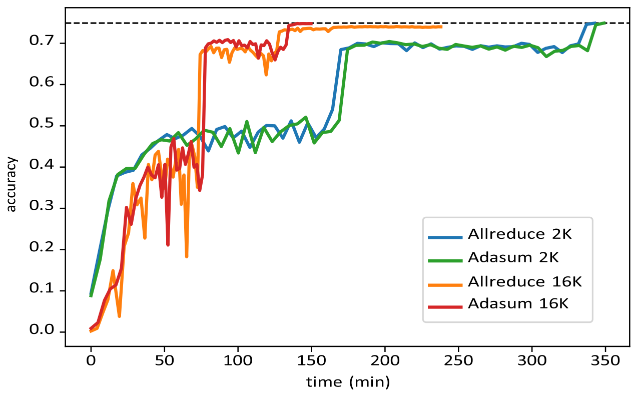


Figure 3. Time-to-accuracy chart for ResNet-50 with 64 GPUs on 16 Standard_NC24rs_v3 VMs.

Table 4 and Figure 3 show the results. For 2K batch size, both allreduce and Adasum reach the target accuracy of 74.9% in 62 epochs. Adasum's time per epoch is 2% slower due to the overheads of the Adasum operation. On the other hand, allreduce with 16K minibatch size *never* reaches 74.9% validation accuracy as it plateaus below (we let it run for 120 epochs). Adasum converges in 69 epochs with this larger minibatch size with the same hyperparameters as the 2K minibatch size. Of course, it might be possible for allreduce to converge with 16K minibatch size with better hyperparameter tuning. One of the advantages of Adasum is that this additional effort is not necessary. Section 5.2.3 investigates this further with an extensive hyperparameter search. Note that increasing the minibatch size from 2K to 16K results in a 61% and 62% improvement in system efficiency for Adasum and allreduce, respectively. Moreover, Adasum 16K achieves the target accuracy of 74.9% and is 2.3× faster in time to accuracy than Adasum 2K, *while using the same number of GPUs*.

5.2.3 *Extensive Hyperparamer Search*

Now we study Adasum on LeNet-5, a relatively small model for which we can do extensive hyperparameter search. These experiments validate that the convergence benefit of Adasum is not sensitive to hyperparameter tuning, i.e., even with highly tuned hyperparameters Adasum improves convergence with large minibatch sizes.

We used the PyTorch version of LeNet-5 found in Horovod's examples with a momentum based SGD optimizer with a minibatch size of 32 and use 99.3% as our target accu-

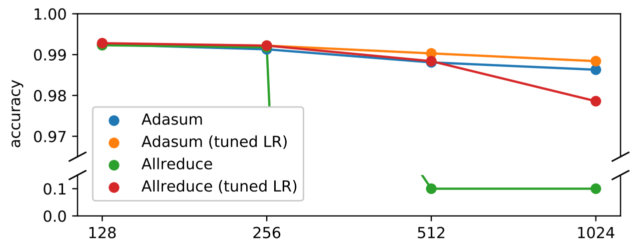


Figure 4. LeNet-5 accuracies under an aggressive learning rate schedule for a small minibatch size scaled up to various large minibatch sizes.

racy. We used an Azure cluster of NC-series VMs with 333 NVIDIA K80 GPUs for these experiments. While the original example has a fixed learning rate schedule of 10 epochs, we were able to bring this down to 2 epochs using a linear warmup and decay to zero. We found a configuration of max learning rate of 0.00328 and a warmup of 17%.

In the following experiments, we scale up the minibatch size with the number of GPUs (i.e., keep the microbatch size constant). Since MNIST has 60000 images, one GPU will take 1875 steps per epoch, while 32 GPUs would take only 58 steps per epoch.

We evaluated Adasum and allreduce with minibatch sizes of 128, 256, 512 and 1024 (4, 8, 16 and 32 GPUs respectively) with both an unmodified learning rate as well as an optimized one, which we searched for separately for each combination of method and minibatch size. For allreduce when we don't tune the learning rate it is scaled up linearly. Figure 4 shows the accuracies reached by each configuration under the aggressive learning rate schedule we found for a the small minibatch size of 32.

Without learning rate tuning allreduce fails to converge at more than 256 minibatch size, while Adasum still converges at 1024 *without any hyperparameter search*. This highlights the easy scalability that Adasum enables. Furthermore, even with a tuned learning rate allreduce is far below even untuned Adasum at a minibatch size of 1024, and is still beat by Adasum with a tuned learning rate at 512 minibatch size.

Consider the tuned learning rates for each configuration:

Method	128	256	512	1024
Adasum	0.033	0.015	0.012	0.02
Allreduce	0.027	0.017	0.009	0.004

For allreduce going from 512 to 1024 minibatch size is coupled with a halving of the learning rate, which means that the per iteration step size stays the same even though twice as many GPUs are participating in each iteration. In contrast, Adasum can maintain much higher learning rates at minibatch sizes of 512 and 1024.

6 Related Work

Previous works for enabling large-batch training have focused on the problem of adapting the learning rate appropriately. Adam (Kingma & Ba, 2015) adjust the step size based on the variance of gradients, taking smaller steps when variance is high. LARS (You et al., 2017) adapts learning rates per-layer using a *trust ratio* calculated as the ratio between the norm of the layer weights and the norm of gradients, the intuition being that divergence happens when steps are large in relation to the parameters being updated. LAMB (You et al., 2019) can be seen as LARS applied to Adam instead of vanilla SGD. These approaches that use statistical measures to adapt learning rate are qualitatively different from Adasum, which exploits a specific property (orthogonality) of gradients to take bigger steps when appropriate. Adasum and learning rate adaption methods are in many cases complementary, as we have shown in our experiments successfully combining Adasum with Adam and LAMB.

Asynchronous SGD (Dean et al., 2012; Chilimbi et al., 2014) approaches can address two issues in distributed synchronous SGD: synchronization overhead of faster nodes having to wait for stragglers to finish the iteration, and non-overlapping of compute and communication. However, stale gradients present another potential source of degraded convergence. Specifically, the DC-ASGD algorithm (Zheng et al., 2016) addressed this staleness using an approximation of the Hessian as used in Adasum. They only use the diagonal elements of the $g \cdot g^T$ approximation of the Hessian and require an additional hyperparameter which requires a careful tuning over time. It was also only evaluated for SGD and Momentum-SGD. Our approach was motivated to be a drop-in replacement of the allreduce operation and thus we eliminate all hyperparameters in our combination and it is optimizer agnostic. Similarly, Maleki et al. (Maleki et al., 2018) use the Hessian to reduce staleness and use a Johnson-Lindenstrauss projection to get a low rank approximation of a semantics-preserving model combiner. Their approach only works with exact Hessian computation and is unlikely to scale to DNNs.

While large-batch training methods decrease the amount of communication needed, *gradient compression* approaches reduce the cost of each communication round. In gradient quantization approaches gradients are cast to a lower bit-width datatype for communication, with bit-widths ranging all the way down to 1 bit (Seide et al., 2014). Low-rank compression methods communicate the most important dimensions of gradients (Vogels et al., 2019).

References

Horovod: Distributed training framework for tensorflow, keras, pytorch, and apache mxnet. https://github.com/horovod/horovod. Accessed: 2020-05-22.

Marian: Fast neural machine translation in c++ https://marian-nmt.github.io. Accessed: 2020-05-22.

NVIDIA deep learning examples for tensor cores. https://github.com/NVIDIA/DeepLearningExamples. Accessed: 2020-05-22.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020.

Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007. doi: 10.1002/cpe.1206. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1206.

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.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1223–1231. Curran Associates, Inc., 2012.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423.

Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H.-W. Unified language model pre-training for natural language understanding and generation. In *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, December 2019.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL http://arxiv.org/abs/1706.02677.

Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Hoffer, E., Hubara, I., and Soudry, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 1731–1741. Curran Associates, Inc., 2017.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima, 2016. http://arxiv.org/abs/1609.04836.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2014. http://arxiv.org/abs/1412.6980.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Maleki, S., Musuvathi, M., and Mytkowicz, T. Semantics-preserving parallelization of stochastic gradient descent. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 224–233, 2018.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2017. http://arxiv.org/abs/1710.03740.

Polyak, B. and Tsypkin, Y. Pseudogradient adaptation and training algorithms. *Automation and Remote Control*, 34: 377–397, 01 1973.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 24*, pp. 693–701. Curran Associates, Inc., 2011.

Reid, M. Fisher information and the hessian of log likelihood. http://mark.reid.name/blog/fisher-information-and-log-likelihood.html, 2012. Accessed: 2020-05-22.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986. doi: 10.1038/323533a0. URL https://doi.org/10.1038/323533a0.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In Li, H., Meng, H. M., Ma, B., Chng, E., and Xie, L. (eds.), *INTERSPEECH*, pp. 1058–1062. ISCA, 2014.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019. http://arxiv.org/abs/1909.08053.

Vandegeijn, R. On global combine operations. *Journal of Parallel and Distributed Computing*, 22 (2):324 – 328, 1994. ISSN 0743-7315. doi: https://doi.org/10.1006/jpdc.1994.1091. URL http://www.sciencedirect.com/science/article/pii/S0743731584710914.

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 14259–14268. Curran Associates, Inc., 2019.

Yin, D., Pananjady, A., Lam, M., Papailiopoulos, D., Ramchandran, K., and Bartlett, P. Gradient diversity: a key ingredient for scalable distributed learning. In Storkey, A. and Perez-Cruz, F. (eds.), *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pp. 1998–2007. PMLR, 09–11 Apr 2018. URL http://proceedings.mlr.press/v84/yin18a.html.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., and Hsieh, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. Technical Report UCB/EECS-2019-103, EECS Department, University of California, Berkeley, Jun 2019. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-103.html.

Zheng, S., Meng, Q., Wang, T., Chen, W., Yu, N., Ma, Z., and Liu, T. Asynchronous stochastic gradient descent with delay compensation for distributed deep learning. *CoRR*, abs/1609.08326, 2016. URL http://arxiv.org/abs/1609.08326.

A Approximating Large Minibatch SGD with Small Minibatch SGD

Suppose $L_1(w)$ and $L_2(w)$ are two loss functions corresponding to two different examples. Starting from model w_0, sequential SGD, calculates $w_1 = w_0 - \alpha \nabla L_1(w_0)$ followed by $w_2 = w_1 - \alpha \nabla L_2(w_1)$ where α is a properly set learning rate for both iteration. With forward substitution, $w_2 = w_0 - \alpha(\nabla L_1(w_0) + \nabla L_2(w_1))$. Alternatively, $\nabla L_1(w_0)$ and $\nabla L_2(w_0)$ (note that gradients are both at w_0) are computed in parallel and w is updated with $w_2' = w_0 - \alpha(\nabla L_1(w_0) + \nabla L_2(w_0))$. Clearly w_2' and w_2 are different because ∇L_2 was computed at a different point.

A.1 Using Taylor Expansion

Adasum uses an estimate for $\nabla L_2(w_1)$ using the Taylor expansion to capture the effect of the first update on the second. Note that $\nabla L_2(w_1)$ is a convenient notation for the first order derivative of L_2 and can be re-written with $\frac{\partial L_2}{\partial w}\Big|_{w_1}$. Therefore:

$$\begin{aligned}\nabla L_2(w_1) &= \frac{\partial L_2}{\partial w}\Big|_{w_1} = \frac{\partial L_2}{\partial w}\Big|_{w_0+(w_1-w_0)} = \frac{\partial L_2}{\partial w}\Big|_{w_0} \\ &+ \frac{\partial^2 L_2}{(\partial w)^2}\Big|_{w_0} \cdot (w_1 - w_0) + O(\|w_1 - w_0\|^2) \\ &= \frac{\partial L_2}{\partial w}\Big|_{w_0} - \alpha \frac{\partial^2 L_2}{(\partial w)^2}\Big|_{w_0} \cdot \nabla L_1(w_0) \\ &+ \alpha^2 O(\|\nabla L_1(w_0)\|^2) \\ &\approx \nabla L_2(w_0) - \alpha H_2(w_0) \cdot \nabla L_1(w_0)\end{aligned} \tag{10}$$

where the error in the approximation is $\alpha^2 O(\|\nabla L_1(w_0)\|^2)$ and $H_2(w_0)$ is the Hessian matrix of loss function L_2. The quadratic relationship between the error in Formula 10 and the learning rate α helps this approximation as in most training practices, learning rate decays as training progresses. However, computing $H_2(w_0)$ requires significant computational power as the size of this matrix is the number of model parameters squared. With millions of parameters, even storing the matrix is infeasible.

(Hastie et al., 2001) shows that for models with negative log likelihood loss functions (which is the case for all models studied in this paper), the Hessian matrix can be approximated by the outer product of the gradients. By using Equation 10 and this approximation, $\nabla L_2(w_1)$ can be rewritten by:

$$\nabla L_2(w_1) \approx \nabla L_2(w_0) - \alpha \nabla L_2(w_0) \cdot \nabla L_2(w_0)^T \cdot \nabla L_1(w_0) \tag{11}$$

A.2 Choosing optimal learning rate

For this discussion, we assumed that α was properly chosen for the sequential SGD algorithm. We use Taylor expansion for the loss function $L_2(w_0 - \alpha \nabla L_2(w_0))$ and we will take its derivative with respect to α to find the optimal value:

$$\begin{aligned}&L_2(w_0 - \alpha \nabla L_2(w_0)) \approx L_2(w_0) - \alpha \nabla L_2(w_0)^T \cdot L_2(w_0) \\ &+ \frac{\alpha^2}{2} \nabla L_2(w_0)^T \cdot H_2(w_0) \cdot \nabla L_2(w_0) \\ &\implies \frac{\partial L_2(w_0 - \alpha \nabla L_2(w_0))}{\partial \alpha} = -\nabla L_2(w_0)^T \cdot L_2(w_0) \\ &+ \alpha \nabla L_2(w_0)^T \cdot H_2(w_0) \cdot \nabla L_2(w_0) = 0 \\ &\implies \alpha \|\nabla L_2(w_0)\|^4 = \|\nabla L_2(w_0)\|^2 \\ &\implies \alpha = \frac{1}{\|\nabla L_2(w_0)\|^2}\end{aligned} \tag{12}$$

where the last line is derived from the approximating for the Hessian matrix. By putting together Equation 12 and 11, $\nabla L_2(w_1)$ can be approximated by:

$$\nabla L_2(w_1) \approx \nabla L_2(w_0) - \frac{\nabla L_2(w_0) \cdot \nabla L_2(w_0)^T}{\|\nabla L_2(w_0)\|^2} \nabla L_1(w_0) \tag{13}$$

Therefore, to approximate the sequential SGD semantics in a parallel setting, Adasum uses:

$$\begin{aligned}w_2 &= w_0 - \alpha(\nabla L_1(w_0) + \nabla L_2(w_1)) \approx w_0 \\ &- \alpha(\nabla L_1(w_0) + \nabla L_2(w_0) \\ &- \frac{\nabla L_2(w_0) \cdot \nabla L_2(w_0)^T}{\|\nabla L_2(w_0)\|^2} \nabla L_1(w_0)) \\ &= w_0 - \alpha(g_1 + g_2 - \frac{g_2 \cdot g_2^T}{\|g_2\|^2} g_1)\end{aligned} \tag{14}$$

where in the last equality, $\nabla L_1(w_0)$ was replaced by g_1 and $\nabla L_2(w_0)$ by g_2 for simplicity (note that the gradients in the last equality are all from w_0).

The Adasum operation is the *symmetric* version of the model update in Equation 14 that samples both paths as described in Section 3.3.

A.3 Convergence Proof for Adasum

(Polyak & Tsypkin, 1973) discusses the requirements for a training algorithm to converge to its optimal answer. Here we will present a simplified version of Theorem 1 and Corollary 1 from (Polyak & Tsypkin, 1973).

Suppose that there are N training examples for a model with loss functions $L_1(w), \ldots, L_N(w)$ where w is the model parameter and w_0 is the initial model. Define $L(w) = \frac{1}{N}\sum_i L_i(w)$. Also assume that w^* is the optimal model where $L(w^*) \leq L(w)$ for all ws. A training algorithm is *pseudogradient* if:

- It is an iterative algorithm where $w_{i+1} = w_i - \alpha_i h_i$ where h_i is a random vector and α_i is a scalar.
- $\forall \epsilon \exists \delta : \mathbb{E}(h_i)^T \cdot \nabla L(w) \geq \delta > 0$ where $L(w) \geq L(w^*) + \epsilon$ and w^* is the optimal model.
- $\mathbb{E}(\|h_i\|^2) < C$ where C is a constant.
- $\forall i : \alpha_i \geq 0$, $\sum_i \alpha_i = \inf$, and $\sum_i \alpha_i^2 < \inf$.

The following Theorem is taken from (Polyak & Tsypkin, 1973).

Theorem A.1. *A pseudogradient training algorithm converges to the optimal model w^*.*

In this section, we assume that the true gradient, ∇L is bounded at any point. As a reminder, $\mathbb{AS}(g_1, g_2) = (1 - \frac{g_1 \cdot g_1^T}{2 \cdot \|g_1\|^2}) \cdot g_2 + (1 - \frac{g_2 \cdot g_2^T}{2 \cdot \|g_2\|^2}) \cdot g_1$. As discussed in Section 3.4 Adasum operator reduces N gradients in a binary tree manner. We will prove that the final gradient has all necessary requirements of pseudogradient. First, we discuss the inner product of Adasum final vector with $\nabla L(w)$:

Lemma A.2. *Suppose $X = \{x_1, \ldots, x_N\}$ is a random variable distribution. For all a and b independently chosen from X, let's define $Y = \mathbb{AS}(a, b)$. Assume that θ is the angle between $\mathbb{E}(X)$ and $\mathbb{E}(Y)$. $\cos\theta > 0.942$.*

Proof.

$$\begin{aligned}
&\mathbb{E}(Y) = \mathbb{E}\left(\mathbb{AS}(a,b)\right) = \mathbb{E}\Big((1 - \frac{a \cdot a^T}{2 \cdot \|a\|^2}) \cdot b \\
&+ (1 - \frac{b \cdot b^T}{2 \cdot \|b\|^2}) \cdot a\Big) = \mathbb{E}(a) + \mathbb{E}(b) \\
&- \mathbb{E}\left(\frac{a \cdot a^T}{2 \cdot \|a\|^2}\right) \cdot \mathbb{E}(b) - \mathbb{E}\left(\frac{b \cdot b^T}{2 \cdot \|b\|^2}\right) \cdot \mathbb{E}(a) = 2\,\mathbb{E}(X) \\
&- \mathbb{E}\left(\frac{a \cdot a^T}{\cdot \|a\|^2}\right) \cdot \mathbb{E}(X)
\end{aligned} \tag{15}$$

where the last equation comes from the independence of a and b. Next we will calculate, η, the angle between $2\,\mathbb{E}(X) - \frac{a \cdot a^T}{\cdot \|a\|^2} \cdot \mathbb{E}(X)$ for some arbitrary a. First let's denote $\mathbb{E}(X)$ with r and assume the angle between r and a is γ. By using the property of inner product, we have:

$$\begin{aligned}
\cos\eta &= \frac{r^T \cdot (2r - \frac{a \cdot a^T}{\cdot \|a\|^2} \cdot r)}{\|r\| \cdot \left\|2r - \frac{a \cdot a^T}{\cdot \|a\|^2} \cdot r\right\|} \\
&= \frac{2\|r\|^2 - \|r\|^2 (\cos\gamma)^2}{\|r\| \cdot \sqrt{4\|r\|^2 + \|r\|^2 (\cos\gamma)^2 - 4\|r\|^2 (\cos\gamma)^2}} \\
&= \frac{2 - (\cos\gamma)^2}{\sqrt{4 - 3(\cos\gamma)^2}}
\end{aligned} \tag{16}$$

By taking a derivative of γ from the last equation, we find the minimum value of $\cos\eta$ to be ≈ 0.9428 which concludes that eta is at most 0.108π. Since in Formula 15, $\mathbb{E}(Y)$ is calculated over an average of all possible a vectors, we can still guarantee that $\mathbb{E}(Y)$ and $\mathbb{E}(X)$ have at most an angle of 0.108π since we derived this value for the worst case scenario. □

Lemma A.3. *With same assumptions as in Lemma A.2, $\|\mathbb{E}(X)\| \leq \|\mathbb{E}(Y)\|$ and $\mathbb{E}(\|Y\|) \leq 2\,\mathbb{E}(\|X\|)$.*

Proof. As discussed in Lemma A.2, $\mathbb{E}(Y) = 2\,\mathbb{E}(X) - \mathbb{E}(\frac{a \cdot a^T}{\cdot \|a\|^2}) \cdot \mathbb{E}(X) = (2I - \mathbb{E}(\frac{a \cdot a^T}{\cdot \|a\|^2}) \cdot \mathbb{E}(X)$. It is trivial to check that the matrix $(2I - \mathbb{E}(\frac{a \cdot a^T}{\cdot \|a\|^2})$ is symmetric with eigenvalues between 1 and 2. Therefore, $\|\mathbb{E}(X)\| \leq \|\mathbb{E}(Y)\|$.

Now suppose that a and b are uniformly randomly chosen from Y.

$$\begin{aligned}
&\mathbb{E}(\|\mathbb{AS}(a,b)\|^2) \leq \mathbb{E}\left(\left\|(1 - \frac{a \cdot a^T}{2 \cdot \|a\|^2}) \cdot b\right\|^2\right. \\
&+ \left.\left\|(1 - \frac{b \cdot b^T}{2 \cdot \|b\|^2}) \cdot a\right\|^2\right) \leq \mathbb{E}\left(2\|b\|^2 + 2\|a\|^2\right) \\
&\leq 4\,\mathbb{E}(\|X\|^2) \implies \mathbb{E}(\|Y\|) \leq 2\,\mathbb{E}(\|X\|)
\end{aligned} \tag{17}$$

□

Assumption: Lemma A.2 showed that in the worst case $\mathbb{E}(Y)$ can rotate at most 0.108π with respect to $\mathbb{E}(X)$. Even meeting this worst case requires carefully crafted x_is. If Adasum was applied recursively on X ($Y = \mathbb{AS}(X, X), Z = \mathbb{AS}(Y, Y), \ldots$) for k times, the expected value of final distribution will at most have an angle of $0.108k\pi$ which is only possible if each Adasum meets the worst case scenario and each worst case is stacked over the previous one. As one can imagine, this is an extremely unlikely scenario. In case x_is are gradients, we assume that Adasum recursively always keeps the angle with $\mathbb{E}(X)$ to at most σ where $\cos\sigma > 0$. Using this assumption and

Lemma A.3, we can prove that Adasum algorithm is a pseudogradient training algorithm.

Theorem A.4. *Adasum algorithm applied in an iterative manner using a proper learning rate on a set of N gradients, $G = \{g_1, \ldots, g_N\}$ computed in parallel, is a pseudogradient training algorithm.*

Proof. Given that Adasum follows the iterative method of SGD, the first assumption of a pseudogradient training algorithm is met. Also, since we use the learning rate schedule from the converging sequential SGD, the requirement for the learning rate is trivially met. Section 3.4 discussed how Adasum reduces all gradients in a binary tree manner which has $\log N$ steps assuming that N is a power of 2. The distribution of the leaf level in this binary tree is G and the next level's distribution is $G_1 = \mathbb{AS}(G, G)$. Level i's distribution is $G_{i+1} = \mathbb{AS}(G_i, G_i)$. At the top of the tree, we have $G_{\log N}$ as the distribution. Using Lemma A.3, $\mathbb{E}(\|G_{\log N}\|) \leq 2^{\log N}\, \mathbb{E}(\|G\|) = N\, \mathbb{E}(\|G\|)$. Since $\mathbb{E}(\|G\|)$ is bounded based on the assumption of vanilla SGD, $\mathbb{E}(\|G_{\log N}\|)$ is bounded as well. This meets the requirement for the norm of the h_is. Finally, Lemma A.3 proves that $\mathbb{E}(G) \leq \|\mathbb{E}(G_{\log N})\|$ and therefore $\mathbb{E}(G_{\log N})^T \dot{\mathbb{E}}(G) \geq \|\mathbb{E}(G_{\log N})\|\, \|\mathbb{E}(G)\| \cos\sigma \geq \|\mathbb{E}(G)\|^2 \cos\sigma$. For any w_i which is not w^*, $\|\mathbb{E}(G)\| > 0$ and based on the assumption, $\cos\sigma > 0$. Therefore, the positive inner product assumption is also met which concludes that Adasum is a pseudogradient training algorithm and it converges. □

A.4 Adasum Convergence Rate

Convergence rate of Adasum is highly dependent on orthogonality of the gradients. In the worst case scenario if all gradients are parallel, the algorithm converges in $1/N$ rate of the sequential SGD where N is the number of processors and in the best case where all of the gradients are orthogonal, we expect Adasum to converge as fast the sequential SGD.

A.5 Convergence of Adasum with per Layer Application

Let's define G^l to be the gradient distribution of layer l. Similar argument from Theorem A.4 can be applied to prove that $\mathbb{E}(G^l_{\log N})^T \cdot \mathbb{E}(G^l) > 0$ where $G^l_{\log N}$ is the distribution of $G_{\log N}$ projected on only layer l. It is straightforward to follow that $\mathbb{E}(G_{\log N})^T \cdot \nabla\mathbb{E}(G) > 0$ as $\mathbb{E}(G_{\log N})^T \cdot \nabla\mathbb{E}(G) = \sum_l \mathbb{E}(G^l_{\log N})^T \cdot \nabla\mathbb{E}(G^l) > 0$. This satisfies the second requirement of Theorem A.1 when Adasum is applied per layer. The third property can be similarly satisfied as $\mathbb{E}(\|G_{\log N}\|^2) = \sum_l \mathbb{E}\left(\left\|G^l_{\log N}\right\|^2\right) \leq \sum_l N\,\mathbb{E}(\|G^l\|^2) = N\,\mathbb{E}\left(\|G\|^2\right)$ and $\mathbb{E}\left(\|G\|^2\right)$ is bounded based on the assumption of vanilla SGD. The other two requirements of Theorem A.1 are satisfied similar to Theorem A.4.

A.6 Convergence of Adasum with Other Optimizers than SGD

Recent works (Kingma & Ba, 2015; You et al., 2019; 2017) have introduced other optimizing methods such as Adam, LAMB or LARS that significantly improves the convergence. In all of these optimizations, for a computed gradient g, $f(g)$ is calculated which returns a vector with the same shape as g and each element of $f(g)[i] = c_i \cdot g[i]$ where c_i is a positive and bounded scalar. These optimizers differ by having different learning rate mechanisms for each parameter. For each of these optimizers, $f(g)$ is only a slightly biased estimation of the gradient. Therefore, we can assume that in expectation, $\mathbb{E}(f(G^l)) \approx c_l \cdot E(G^l)$ for each layer l where c_l is a positive bounded scalar. If Adasum is applied to $f(g^l_1), \ldots, f(g^l_N)$ for each layer, similar positive inner product argument in Theorem A.4 can be applied and conclude that the inner product of the final vector is only multiplied by c_l which is positive and bounded. Similarly, because $f(g)$ only multiplies the elements of g by a bounded value, $\mathbb{E}(\left\|f(G^l)\right\|^2)$ as well as expectation of the norm of the final generated vector by Adasum are bounded. Therefore, Adasum converges for other optimizers where in expectation, $f(G)$ is a slightly biased G.