Secure Sparse Gradient Aggregation in Distributed Architectures

van Rooij, M.; van Rooij, S.; Bouma, H.; Pimentel, A.

DOI
10.1109/IOTSMS58070.2022.10062180

Publication date
2022

Document Version
Final published version

Published in
2022 Ninth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)

License
Article 25fa Dutch Copyright Act (https://www.openaccess.nl/en/in-the-netherlands/you-share-we-take-care)

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 426, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)
Secure Sparse Gradient Aggregation in Distributed Architectures

Mario van Rooij
UvA + TNO
The Netherlands
mariovanrooij@hotmail.com

Sabina van Rooij
TNO
The Hague, The Netherlands
sabina.vanrooij@tno.nl

Henri Bouma
TNO
The Hague, The Netherlands
henri.bouma@tno.nl

Andy Pimentel
University of Amsterdam
Amsterdam, The Netherlands
a.d.pimentel@uva.nl

Abstract—Federated Learning allows multiple parties to train a model collaboratively while keeping data locally. Two main concerns when using Federated Learning are communication costs and privacy. A technique proposed to significantly reduce communication costs and increase privacy is Partial Weight Sharing (PWS). However, this method is insecure due to the possibility to reconstruct the original data from the partial gradients, called inversion attacks. In this paper, we propose a novel method to successfully combine these PWS and Secure Multi-Party Computation, a method for increasing privacy. This is done by making clients share the same part of their gradient, and adding noise to those entries, which are canceled on aggregation. We show that this method does not decrease the accuracy compared to existing methods while preserving privacy.

Index Terms—Federated Learning, Security, Privacy, Distributed systems, IoT, Big Data, Secure Multi-Party Computation

I. INTRODUCTION

More, and differently distributed data often results in a more general and accurate machine learning model, but the rise in cyber attacks and leaks [1] and European privacy laws [2] make it more difficult and irresponsible to freely centralize medical [3], personal [4], and other sensitive data. This problem can be dealt with by using Federated Learning (FL), a method in which multiple parties, e.g. Internet of Things (IoT) devices, send their gradient or weights to a central server to update a model over many iterations while keeping the data locally.

Initial methods [5]–[7] assumed gradient vectors were safe to send to the central server, but research has shown the original data can be recovered from them [8]–[11]. In image analysis, the recovery of images from gradients is called an inversion attack.

A mechanism that can be used to counter such inversion attacks is Homomorphic Encryption (HE) [12]. However, HE significantly increases computational-, and communication costs, and only one adversary among the clients is needed to reveal gradients, as a common private key is used [12].

Another method, called Partial Weight Sharing (PWS), is a way of updating the model by sending only parts of the gradient [7], [13], [14], which also reduces communication costs significantly, an important aspect for large models [15]. Further reduction can be reached by quantization in integer values [16], [17]. A popular PWS method is Top_k, where only the K entries of the largest magnitude of the gradient are shared. Deep Gradient Compression (DGC) [13] adds additional tricks but only shares 0.1% of the gradient while maintaining high accuracy. However, PWS by itself is not enough to protect against the server since a partially shared gradient can still be turned into the original image [12].

A different class of protection is the addition of noise to local gradients, a specific variant of which is Differential Privacy (DP) [18]. DP adds enough noise to a gradient to be protected against inversion attacks. DP has the disadvantage that it deteriorates accuracy [19].

Another method using noise by Bonawitz et al. [20] uses a secure sum to prevent inversion attacks. One client adds a noise vector to their gradient, and another client subtracts the same noise vector, and upon aggregation, these noises cancel out. This is a form of Secure Multi-Party Computation (S-MPC) since it reveals only the sum of the gradients and not the individual gradients. The benefit as compared to DP is that S-MPC does not deteriorate accuracy. However, it does not protect the global model against membership interference attacks [11]. Bonawitz et al. also found an efficient way of generating the noise vector by using a common seed, generated by an Elliptic Curve Diffie-Hellman (ECDH) key-exchange [21], for a pseudo-random number generator.

In this paper, we introduce two novel methods, called Secure Sparse Gradient Aggregation (SSGA) and a variant that deploys Deep Gradient Compression (SSGA-DGC).

Our novel Secure Sparse Gradient Aggregation (SSGA) and SSGA-Deep Gradient Compression (SSGA-DGC) methods make sparse PWS compatible with S-MPC to prevent direct inversion attacks and reduce communication costs. They do so by letting clients send the same part of their gradient and adding noise to those entries, which cancel on aggregation. Furthermore, we upgrade the security of the sparse method by only revealing the sum of the sparse gradients instead of the individual gradients. Lastly, we make an integer S-MPC scheme for safe aggregation suitable for real-valued numbers and analyze efficiency and total training time.

II. METHOD

This section first introduces the necessary preliminaries to the method. After this, the complete novel method is discussed. In Section II-A, the basics of FL and Top_k algorithm are
explained. Section II-B introduces how S-MPC can be used to hide gradient contributions of clients in FL. Lastly, Subsection II-C explains the proposed method, which is a combination of FL, S-MPC, and PWS.

A. PWS in FL

FL [5], [6] is an optimization technique, where multiple clients having their local data contribute to a model located on a server. The goal is to find \( w \), the weights of the model, such that a loss function \( F(w) \) has the smallest value. A common tactic to achieve a minimum is to perform stochastic gradient descent (SGD) on the data [22], which moves the weights \( w \) towards a minimum of the loss in an iterative process. In FL, clients produce a gradient for every iteration from a mini-batch. These gradients are then summed up and used to update the central model.

Let \( \nabla_{k,t} \) be the gradient of client \( k \) in iteration \( t \). To reach a minimum of \( F(w) \), weights are changed as follows:

\[
w_{t+1} = w_t - \eta(t)v_t,
\]

where \( \eta(t) \) is the learning rate, dependent on the iteration \( t \), and \( v_t \) is the momentum given by:

\[
v_{t+1} = \gamma v_t + \sum_{k=1}^{N} \nabla_{k,t},
\]

State-of-the-art optimizers like ADAM [23] introduce additional tricks, such as an adaptable learning rate, which can enhance the optimization process.

It is beneficial to compress the gradient using PWS, as communicating the full gradient tends to be a significant bottleneck in training a model. \( \text{Top}_k \) is one way of compressing the gradient, by replacing the term \( \nabla_{k,t} \) in Equation 2 with the \( K \) entries largest in magnitude in the gradient [24], denoted \( \text{Top}_k(\nabla_{k,t}) \). \( \text{Top}_k \) works because the relevant information of the gradient is predominantly contained in the components largest in magnitude.

Whereas the normal \( \text{Top}_k \) algorithm discards the entries of the gradient which are not sent, a residual can be introduced to keep track of the entries which are not sent [25]. The residual at the start of round \( t \) is defined by:

\[
R_{k,t} = R_{k,t-1} + \nabla_{k,t}.
\]

After communication of the \( \text{Top}_k(R_{k,t-1}) \), the communicated values are removed from the residual. So, in the proposed methods, instead of sending entries in the gradient which belong to the \( \text{Top}_k \) entries, the \( \text{Top}_k \) entries in the residual are chosen.

B. S-MPC and SGA

For the methods, parties require a shared, private key. A common key between clients-pairs to generate noise vectors efficiently can be generated using a Diffie-Hellman (DH) key exchange [21]. The seed for the pseudo-random number generator is refreshed every round by hashing. The key is then used to sample the noises \( \epsilon_{ki} \) between clients \( k \) and \( i \), sampled uniformly from \( \mathbb{Z}_R \). \( R \) is a parameter that determines the maximum integer number a client can contribute, \( R_U \):

\[
R_U = \left\lfloor \frac{R}{|C|} \right\rfloor - 1,
\]

where \( C \) is the pool of clients and \( |C| \) is the size of the client pool. The noises \( \epsilon_{ki} \) are used to create the uniformly distributed vector message [20]:

\[
P_{k,t} = \left( \nabla_{k,t} + \sum_{i \in C,k<i} \epsilon_{ki} - \sum_{i \in C,k>i} \epsilon_{ki} \right) \mod R,
\]

where \( \nabla_{k,t} \) is assumed to be of integer form. Integer conversion is addressed later on. \( P_{k,t} \) is now sent to the server and the aggregation equals:

\[
\sum_{k \in C} P_{k,t} \mod R = \sum_{k \in C} \nabla_{k,t} \mod R = \sum_{k \in C} \nabla_{k,t},
\]

We call this way of sending gradients Secure Gradient Aggregation (SGA). When we refer to SGA as a method, the difference with SGD is the use of the encrypted and integer-converted gradients, created with Equation 5.

C. Secure Sparse Gradient Aggregation (SSGA)

To make S-MPC and PWS compatible, all clients have to send the same entries. This can be done by producing a mask for the gradient, which all clients have access to. Let \( M \) be such a mask, which is a vector with the same length as the gradient. Clients only send the gradient values where the mask has a value of 1. Every client applies it to its gradient and obtains the following quantity:

\[
P_{k,i}[M] = \left( \nabla_{k,i}[M] + \sum_{i \in C,k<i} \epsilon_{ki}[M] \right. - \sum_{i \in C,k>i} \epsilon_{ki}[M] \left. \right) \mod R,
\]

where \( \nabla_{k,i} \) is the gradient of client \( k \) in round \( t \), \( \epsilon_{ki} \) is the noise vector generated from a common private key between two clients, and \( \nabla_{k,i}[M], \epsilon_{ki}[M], P_{k,i}[M] \) indicate the masked versions of \( \nabla_{k,i} \), \( \epsilon_{ki} \), and \( P_{k,i} \), respectively. We can now calculate the sum of \( P_{k,i}[M] \) over all clients:

\[
\sum_{k \in C} P_{k,i}[M] \mod R = \sum_{k \in C} \nabla_{k,i}[M],
\]

which can now be used to update the global model using Equation 1. The only difference is that the gradient, \( \nabla_{k,i} \), is replaced with a sparse gradient, \( \nabla_{k,i}[M] \).

Having combined S-MPC and PWS, a mask \( M \) has to be constructed such that applying the mask to all clients’ gradients still makes the masked gradient contain large entries. It is constructed using a union \( I = \cup_{k \in C} i_k \), where \( i_k \) is the set of indices of gradient entries which exceed a threshold.
Having a shared mask provides a communication benefit because all clients send the same $\approx K$ entries. In the normal Top$_k$ algorithm the download size increases approximately linearly with the number of clients [26], [27]. In SSGA, the download size is a constant maximum as a function of the number of clients determined by how many indices are selected per client.

Algorithm 1 Secure Sparse Gradient Aggregation (SSGA) and SSGA-DGC algorithm. Lines containing \( \square \) are specific to SSGA, while lines containing \( \blacksquare \) are specific to SSGA-DGC. During a number of iterations, a server receives updates from clients that it uses to update a central model. After every iteration, the server broadcasts the updated model to all the clients.

1: Clients $C$
2: Establish common keys $s_{k,i}$
3: for $t = 1, 2, \ldots, N_{\text{iterations}}$ do
4: \hspace{1em} for $k = 1, 2, \ldots, N_{\text{clients}}$ in parallel do
5: \hspace{2em} Calculate local gradient $\nabla \ell_k = \nabla w_t (x_{k,t})$
6: \hspace{2em} Update residual $R_{k,t} = R_{k,t-1} + \nabla \ell_k$
7: \hspace{2em} Update local momentum $\eta_k \leftarrow \gamma \eta_k - \nabla \ell_k$
8: \hspace{2em} Update residual $R_{k,t} = R_{k,t-1} + \eta_k$
9: \hspace{2em} $i_{k,t} = \text{Top}_k$ indices of $R_{k,t}$ to server
10: \hspace{1em} Server broadcast of $I_{k,t} = \cup_{i \in C} I_{k,t}$
11: \hspace{1em} Clients $M[i] = 1$ if $i \in I_{k,t}$ else 0
12: \hspace{1em} Generate noises $\epsilon_{k,i}$ for $k; i \in C$ with keys
13: \hspace{2em} $P_{k,t} = R_{k,t} [M] + \sum_{i \in C, k \neq i} \epsilon_{k,i} - \sum_{i \in C, k > i} \epsilon_{k,i}$
14: \hspace{2em} $P_{k,t} \rightarrow$ server
15: \hspace{2em} Calculate new residual $R_{k,t} = R_{k,t} [-M]$ \( \square \)
16: end for
17: Server momentum $m_{t+1} = \gamma m_t + \sum_{k \in C} P_{k,t}$ \( \square \)
18: Server changes weights $w_{t+1} = w_t - \eta m_{t+1}$, \( \blacksquare \)
19: Server aggregates $w_{t+1} = w_t - \eta \sum_{k \in C} P_{k,t}$ \( \blacksquare \)
20: $w_{t+1} \rightarrow$ clients
21: end for

III. EXPERIMENTS AND RESULTS

This section describes the soft-, and hardware used in the project, what benchmarks were performed, and what the outcomes were.

A. Experimental Setup

Experiments measuring time were performed on a single machine, where all clients were simulated with one NVIDIA GTX 1080 ti, having one Intel® Xeon® Bronze 3104 core to perform CPU operations. To implement the novel method, we used Horovod [28], Pytorch [29], and TinyEC1. For the key establishment, we used the brainpoolP256r1 elliptic curve [30]. Experiments used the CIFAR-10 dataset [31] with a Resnet-110 model architecture training 200 epochs unless indicated that the TinyImagenet dataset [32] was used, which used a Resnet-50 model with 50 epochs of training. Datasets were always randomly split among 4 clients while there was

---

1https://github.com/alexmg/rinyc
Fig. 2. Validation loss curves for the SGD, SGA, DGC, and ADAM algorithms on CIFAR-10. The curve was taken from a single client in a single run of the experiment.

a common validation-, and test set. The batch size used was 128. The learning rate for SGD and SSGA was set to 0.02 and for SSGA-DGC 0.1. For TinyImageNet the learning rate was 0.0125. The learning rate decayed over the epochs to 0 using a cosine scheduler. For CIFAR-10 a weight decay of 0.9 was used wherever momentum was involved. The floating point numbers are projected onto 32-bit integers.

B. DGC

DGC serves as a good baseline for comparison to the novel methods, since it preserves accuracy while significantly compressing the gradient. Yet it was unclear what baseline DGC is compared to in the original paper. It is important to establish what method was used to see if DGC actually performs as well as other state-of-the-art gradient-descent algorithms. We hypothesize that DGC does not have higher accuracy than state-of-the-art gradient-descent algorithms on the CIFAR-10 dataset. We reproduce the results in DGC and compare them to the SGD, SGA, and ADAM optimizer [33]. To verify the results in the original paper on DGC, we train a model using the baselines and DGC and compare it using accuracy.

Figure 2 shows the validation loss curve as a function of the number of epochs for the CIFAR-10 dataset. Initially, DGC converges faster than SGD and ADAM, but in the final epochs, SGD has a lower value for the loss function.

To further assess the quality of the different optimizers we looked at the accuracy. Figure 3 shows the accuracy with the respective standard deviation obtained over 5 experiments for ADAM, SGD, SGA, and DGC in the FL setting. We can observe that DGC outperforms ADAM by around 2%, but is outperformed by SGD by 0.6%. Furthermore, SGA shows no significant decline as compared to SGD. To prove statistical significance the Welch-t-test is used. The mean of DGC significantly differed from all other methods ($p < 0.01$). When comparing the results to the DGC paper [13] it can be observed that a similar accuracy is reached as compared to the paper (93.87% in the paper). In every case a lower test-loss results in higher accuracy. This establishes that the optimizer used in the baseline of DGC was a state-of-the-art optimizer, but that SGA and SGD outperform DGC.

C. Top-$k$ and Random Masks

In SSGA and SSGA-DGC masks are used to ensure the same entries are shared. These masks have to be computed and communicated by the clients. Random masks, therefore, reduce overhead and computation. We hypothesize that random masks will diverge, and reach a lower accuracy than the use of Top-$k$ masks. To assess the added value of Top-$k$ masks, we let one group use random masks, and another Top-$k$ masks (selection of highest magnitude entries). Using the Top-$k$ mechanism, both SSGA and SSGA-DGC converge, while the mechanism using random masks diverges. When the learning rate was lowered the accuracy was stuck around 10%, the amount that would be correctly guessed randomly. This confirms that for SSGA and SSGA-DGC, the Top-$k$ masks are needed.

D. Fixed global maximum

In SSGA, as well as SSGA-DGC clients share the maximum value of their residual entries every iteration. By doing this, the quantization error is minimized. However, this conversion might potentially reveal something about the distribution of the clients residual. Therefore, it may be desirable if the clients had a fixed maximum value that can be contributed every round. This can be achieved by setting a maximum value before the algorithm starts and letting clients send values no larger than this value, by use of clipping.

2Note that the only difference between SGD and SGA is the secure aggregation
One problem with a fixed maximum is that the distribution of residuals changes over the training process. At the start of training, residuals often have a larger standard deviation (see Figure 4), whereas later in the process the standard deviation becomes smaller. Furthermore, it is known that for different models gradient distributions are also different, and therefore it can be difficult to set a maximum value by hand. Lastly, the compression impacts the distribution of the residual. The higher the compression, the more the values will add up while not being sent, making the distribution wider.

To assess the impact of using a fixed value for the maximum residual value, different maximum values were used and the accuracy was measured. It is expected that the accuracy reached using a fixed value is significantly lower than the original SSGA, and SSGA-DGC with adaptive maximum values.

Figure 5 displays the final accuracy for the baselines using the adaptive maximum (left), SSGA using different maximum values (middle), and SSGA-DGC using different maximum values (right). We can see that in cases where an unsuited maximum value is chosen, there is a significant decline in accuracy and that there are two areas of decline. One where the maximum integer is too large, and one where it is too small.

The graph shows that there is an optimal global maximum that reaches a similar accuracy as compared to the adaptive maximum strategy. We can conclude that fixing the global maximum harms accuracy when the wrong maximum value is chosen. The security risk introduced by the adaptive minimum is minimal, as a single value cannot possibly reveal a lot of information about the data.

E. SSGA Accuracy

To assess the quality of the SSGA, and SSG-DGC optimizers we train on CIFAR-10 for different compression ratios. To evaluate the accuracy, the methods are compared to the baselines established in Sec. III-B. We hypothesize that SSGA and SSGA-DGC have similar accuracy to DGC with a comparable compression ratio and that a higher compression ratio decreases the accuracy of both SSGA and SSGA-DGC. SSGA and SSGA-DGC with a 200× compression ratio are compared to DGC because the communication costs are similar. Figure 6 shows the accuracy for the baselines and novel methods. To indicate compression of 200×, we add the suffix 200 to the method, e.g. SSGA-200. First, there is no significant decline in accuracy for both SSGA-200 and SSGA-DGC-200 compared to DGC. Furthermore, we can see that both methods show a decline as a function of the compression rate. At 400× compression, SSGA seems to decline but, although SSGA-
DGC-400 is much more efficient than DGC, its accuracy does not decline significantly.

To further investigate the methods, the accuracy is assessed on TinyImageNet. The model is only finetuned, meaning that only the last fully connected layer is trained and the rest of the layers are frozen. This last layer consists of \((512 \times 4) \times 200\) weights. Figure 7 shows that SSGA and SSGA-DGC reach a similar accuracy to each other at every compression ratio. A Welch-t-test shows that SSGA-DGC outperforms SSGA at 50, 100, and 400 × compression. It also shows that there is no significant decline as compared to SGD at 100 ×, and 1,000 × compression. As compared to DGC, SSGA and SSGA-DGC perform significantly better at 1,000 × compression, and worst at 100,000 × compression. From this first experiment in Sec. III-E we can conclude that SSGA and SSGA-DGC perform similar to or better than DGC and that the accuracy declines as the compression ratio increases. Furthermore, SSGA and SSGA-DGC can withstand higher compression ratios than DGC before a significant decline in accuracy is observed.

### F. Effect of Residual

The next experiment measures the effect of the residual. We compare SSGA to SSGA without a residual, measuring the effect of the residual on the accuracy. We hypothesize that a residual is important to maintain accuracy. To assess the effect of the residual on the accuracy in SSGA an experiment is performed without the residual. The experiment uses CIFAR-10 with 200 × compression. The result is shown in Table I under the name SSGA-200-no-residual. Removing the residual from SSGA-200 results in a negative effect on the accuracy. There was a 12.2% decline in accuracy as compared to SSGA-200. Therefore, we conclude that the residual mechanism plays an important part in the preservation of accuracy.

### G. Quantization on Different Data-Types

As SSGA and SSGA-DGC use integers to project the real numbers onto, different integer types can be used. In the standard SSGA and SSGA-DGC signed int-32 is used. Using integer types with more bits results in higher accuracy in calculating the sum of the residual, but also increases communication costs. Therefore, we are interested in establishing the effect of using smaller integer values on the accuracy of a model. The value that changes is \(R_U\) in Equation 4 since \(R\) is determined by the number of bits in the integer. We expect unsigned int-16, and unsigned int-8 to result in a lower accuracy when training a model. Different data types are used on CIFAR-10 with 400 × compression. Note that the outcome of this experiment may be different when using more than 4 clients. As unsigned int-16, and unsigned int-8 are not supported by Nvidia NCCL operations, we simply use signed int-32, but restrict the maximum value to simulate integers having a smaller number of bits.

Figure 8 shows the results for the two methods for different integer types (int-32, uint-16, uint-8). A Welch-t-test revealed that there was no significant impact on the accuracy of using different integers. Therefore, it would be beneficial to work with uint-16 or uint-8, when the NCCL allreduce operations are available in NCCL.
H. Energy- and Time-Efficiency

Lastly, we assess the communication and computational costs of the novel methods in two experiments. This is relevant because the algorithm should either be more efficient computationally, more accurate, or more secure (while preserving accuracy as compared to other methods) than other optimizers. In the first experiment the training time and GPU energy usage are measured for a full training cycle for SGD, SSGA, SSGA-DGC, and DGC, to determine the efficiency of the different methods. As communication time will be negligible in the following experiment due to an extremely fast network, our hypothesis of energy use and computation has to be based on the computational side. The computational efficiency is measured by training the different methods on CIFAR-10 on an 82 Gbps network, measured using the OSU microbenchmark (OSU-Micro-Benchmarks/5.7.1-gompi-2021a-CUDA-11.3.1), taking the highest average result over 100 runs as the network speed. During the training process, the GPU usage is sampled using WandB [34] to measure how active the GPUs are during training. The curve is then integrated to obtain energy usage. The total training time is the complete runtime of the algorithm, including key establishment, training, test, and validation. Table II shows the total training time and GPU energy usage for the different methods. The most efficient method is SGD, which uses significantly less energy and takes significantly less time to train the model. Lower training time is caused by an extremely low communication time, as a single machine was used to simulate all 4 clients. When comparing the energy use of the sparse methods, we can observe that DGC uses the most energy, SSGA-200× comes after that with a 7% decrease as compared to DGC, and SSGA-DGC is the most efficient with a 14% decrease as compared to DGC. Therefore we can conclude that SSGA-200×, and SSGA-DGC-200× use significantly less GPU energy than DGC. When comparing total training times, SSGA-DGC-200× is also significantly faster than DGC. On the other hand, SSGA-200× has a larger training time than DGC.

The second experiment models the training times for SSGA, SSGA-DGC, and SGD for different network speeds. In this case, we expect SSGA, and SSGA-DGC to outperform SGD in total training time on slow networks. The reason for this is that in slower networks communication takes longer, and since SSGA and SSGA-DGC compress the communicated information, communication takes significantly less time on slow networks for the methods as compared to methods that do not compress the gradient, like SGD. To determine on what kind of networks SSGA and SSGA-DGC start getting useful we model the time it takes to complete training on different networks. This is done using two assumptions. First of all, we assume that all clients have the same computational power. Secondly, we assume only the NCCL_Allgather and NCCL_Allreduce operations in the Horovod logging files scale linearly with the network speed (e.g. a 2× slower network takes 2× as long). We performed a measurement on SSGA and SSGA-DGC on CIFAR-10 with an 82 Gbps network. The modeled training time as a function of the network speed is displayed in Figure 9. SGD has a lower computation time than SSGA, but SSGA and SSGA-DGC have a lower communication time. In slow networks, this means that SSGA and SSGA-DGC outperform SGD in total training time. The total training time is lower for SSGA and SSGA-DGC in networks having a speed of 1Gbps. We can conclude that for network bandwidth smaller than 1 Gbps, SSGA-DGC and SSGA both outperform SGD in training time using this specific setting.

TABLE II ENERGY USE AND TRAINING TIME OF DIFFERENT METHODS ON CIFAR-10. THE ERROR ON THE GPU ENERGY USAGE AND TOTAL TRAINING TIME IS GIVEN BY THE STANDARD DEVIATION OVER 3 RUNS.

<table>
<thead>
<tr>
<th>Method</th>
<th>GPU Energy Usage (kWh)</th>
<th>Total Training time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>0.8433 ± 0.0022</td>
<td>4502 ± 10</td>
</tr>
<tr>
<td>DGC</td>
<td>1.108 ± 0.029</td>
<td>5405 ± 19</td>
</tr>
<tr>
<td>SSGA-200</td>
<td>1.027 ± 0.003</td>
<td>6002 ± 16</td>
</tr>
<tr>
<td>SSGA-DGC-200</td>
<td>0.9502 ± 0.0011</td>
<td>5338 ± 3</td>
</tr>
</tbody>
</table>

Fig. 9. modeled training time for the two different methods using different network speeds. SGD has a lower training time for high network speeds, whereas SSSGA has a lower training time for slower networks. The gradient was compressed 200×. The error in the communication time was negligible.
compression without a significant decline in accuracy for finetuning. The method is useful compared to SGD on 1Gbps networks or slower, when the increased computation time of SSSA and SSSA-DGC is overtaken by the increased communication time. The proposed method has a constant download size as a function of the number of clients, whereas other methods scale approximately linearly with the number of clients. Future work could include experimentally determining whether SSSA and SSSA-DGC hold up against inversion attacks, and benchmarking the methods on different datasets. It would also be helpful to further improve the current approach with the possibility of dropouts, by additional secret sharing [20].

REFERENCES