SparseVFL: Communication-Efficient Vertical Federated Learning Based on Sparsification of Embeddings and Gradients

Yoshitaka Inoue
DOCOMO Innovations
Sunnyvale, CA, USA
yoshitaka.inoue@docomoinnovations.com

Hiroki Moriya
DOCOMO Innovations
Sunnyvale, CA, USA
hiroki.moriya@docomoinnovations.com

Qiong Zhang
Amazon Web Services
Seattle, WA, USA
zhangyqi@amazon.com

Kris Skrinak
Amazon Web Services
Seattle, WA, USA
skrinak@amazon.com

ABSTRACT

In Vertical Federated Learning, a server coordinates a group of clients to perform forward and backward propagation through a neural network. The server and clients exchange intermediate embedding and gradient data, which results in high communication cost. Traditional approaches trade off the amount of data exchanged with the model accuracy. In this work, we propose the SparseVFL algorithm in order to reduce the amount of exchanged data, while maintaining the model accuracy. In both the forward and backward propagation, SparseVFL makes sparse embeddings and gradients based on the combination of ReLU activation, L1-norm of embedding vectors, masked-gradient, and run-length coding. Our simulation results show that SparseVFL outperforms existing methods. SparseVFL can reduce the data size by 68–81% and the training time by 63% at a communication throughput of 10 Mbps between the server and clients.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Distributed algorithms**; • **Networks** → **Cloud computing**.

KEYWORDS

federated learning, vertical federated learning, communication cost

ACM Reference Format:
Yoshitaka Inoue, Hiroki Moriya, Qiong Zhang, and Kris Skrinak. 2023. SparseVFL: Communication-Efficient Vertical Federated Learning Based on Sparsification of Embeddings and Gradients. In *KDD FL4Data-Mining '23, August 7, 2023, Long Beach, CA, USA.*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD FL4Data-Mining '23, August 7, 2023, Long Beach, CA, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Federated Learning (FL) has attracted a lot of attention in recent years, for secure use of private data held on thousands of mobile devices or held across tens or hundreds of organizations [14, 25].

In Vertical Federated Learning (VFL) [23], clients can have distributed different features and a server can have labels as long as they have a common sample set. The server coordinates a group of the clients to perform forward and backward propagation through a neural network. In forward propagation, each client uploads the **embedding** vectors to the server. In backward propagation, the server returns the **gradient** vectors to each client. This process is repeated to minimize the loss.

A key challenge for VFL is to reduce the communication costs, which can be achieved by reducing the data size (i.e., embeddings and gradients) exchanged between the server and clients. Reducing the data size also leads to a reduction in the overall training time.

In this work, we focus on the sparsification of embedding and gradient vectors to reduce communication costs, and propose a communication-efficient VFL scheme called SparseVFL. We hypothesize that sparsification, i.e., increasing the number of zeros, would allow for significant data reduction through run-length coding. SparseVFL achieves data reduction by simply making partial modifications to the neural networks and loss function that make up the VFL.

2 RELATED WORKS

Data Size. Run-length coding [16] and Huffman coding [12] and basic approaches to data size reduction. Quantization also reduces data size and training time while maintaining model quality [6, 24], however, a shorter bit length causes quantization errors. Retaining only a limited number of gradient elements that are considered important, such as elements whose absolute magnitude is in the top-k, can reduce data size while maintaining model quality [15, 17, 18, 22]. These approaches discard small gradients and may degrade accuracy. Dimension reduction directly reduces the data size. Khan *et al.* [13] apply PCA or autoencoder to the client input data and sends the embedding to the server only once. Feng *et al.* [9] select important features by adding the L2,1-norm of the model parameters to the loss function, and this feature selection process is performed inside the client by using locally predicted labels. However, these approaches [9, 13] do not necessarily optimize the entire model

consisting of the client and server models. Another approach is to reduce the number of clients communicating with the server by evaluating the contribution of each client [8, 21], however, it doesn't reduce the data size for the clients with high contribution.

Reuse. Communication costs can be reduced by reusing old embeddings. Chen *et al.* [3] propose asynchronous VFL, where clients upload embeddings asynchronously, and the server uses the latest embeddings for active clients and the old embeddings for inactive clients. Fu *et al.* [10] propose CELU-VFL, where the server caches embeddings and gradients from the previous round and reuses such old data for some rounds. These approaches would slow down the convergence, however, they can be integrated with the data size reduction methods mentioned above, including our proposed method.

Methods under Different Assumptions. C-VFL [2] assumes that embeddings, labels, and server model parameters are shared among all parties, including from one client to another client, so that each client model can be trained in the local iterations in each global round without additional communication costs. Xing *et al.* [4] reduces communication costs by using asynchronous training and quantization over the SplitNN model [11], where one client sends the latest client model parameters to the other client. C-VFL and SplitNN-based approaches are not fairly comparable to our proposed method because they make different information management assumptions than our work. Our assumptions are described in Section 3.1.

3 VERTICAL FEDERATED LEARNING

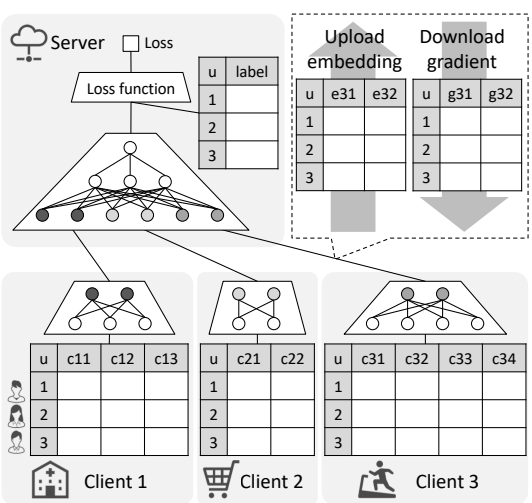


Figure 1: VFL architecture.

Figure 1 shows a typical VFL architecture. Let M be the number of clients, and each client $m \in [1, M]$ has N samples in the same order. N can be regarded as either the total number of samples or the batch size. Each sample $n \in [1, N]$ has a feature vector $\mathbf{x}_{n,m} \in \mathbb{R}^{D_{x,m}}$ stored in client m, where $D_{x,m}$ is a dimension of the feature vector.

In the forward-propagation, a client model h_m parameterized by θ_m maps the input feature vector $\mathbf{x}_{n,m}$ into an embedding vector $\mathbf{h}_{n,m} \in \mathbb{R}^{D_{h,m}}$, i.e., $\mathbf{h}_{n,m} := h_m(\theta_m; \mathbf{x}_{n,m})$, where $D_{h,m}$ is a dimension of the embedding vector. Meanwhile, a server model h_0 parameterized by θ_0 maps the set of embedding vectors $\{\mathbf{h}_{n,1}, ..., \mathbf{h}_{n,M}\}$ into a predicted vector $\tilde{\mathbf{y}}_n \in \mathbb{R}^{D_y}$, i.e., $\tilde{\mathbf{y}}_n := h_0(\theta_0; \mathbf{h}_{n,1}, ..., \mathbf{h}_{n,M})$. Then, using a label vector $\mathbf{y}_n \in \mathbb{R}^{D_y}$, the server minimizes a loss function $L := \frac{1}{N}\sum_{n=1}^{N} l(\theta_0, \mathbf{h}_{n,1}, ..., \mathbf{h}_{n,M}; \mathbf{y}_n)$.

In the back-propagation, the server sends the following gradient vector for $\mathbf{h}_{n,m}$ to client m.

$$\frac{\partial L}{\partial \mathbf{h}_{n,m}} = \left[\frac{\partial L}{\partial h_{n,m,1}}, ..., \frac{\partial L}{\partial h_{n,m,D_{h,m}}}\right] \in \mathbb{R}^{D_{h,m}} \tag{1}$$

Once client m gets the gradient $\partial L/\partial \mathbf{h}_{n,m}$, client m computes another gradient $\partial h_{n,m,k}/\partial \theta_m$ and combines the two gradients to compute the other gradient $\partial L/\partial \theta_m$ by using *chain rule*.

$$\frac{\partial L}{\partial \theta_m} = \sum_{n=1}^{N} \frac{\partial L}{\partial h_{n,m,k}} \frac{\partial h_{n,m,k}}{\partial \theta_m} \tag{2}$$

Each party, server or client, has its own optimizer, and the optimizer updates the model parameter θ_0 or θ_m based on its gradients.

3.1 Information Management Assumptions

To reduce the risk of privacy leakage among clients and to reduce the system complexity at the clients, this work assumes a server-centric star network as shown in Figure 1, where clients trust with servers, but clients do not trust with other clients because they may be competitors of each other. In this work, available information is defined as follows: (1) each party shares sample indices n (e.g., user id) and the number of samples N among all parties through the server, (2) each client m shares dimension $D_{h,m}$ and embeddings $\mathbf{h}_{n,m}$ only with the server, and (3) the server shares gradients $\frac{\partial L}{\partial \mathbf{h}_{n,m}}$ only with client m. Except for information necessary to synchronize training, the exchange of other information (e.g., labels, features, model parameters) and the use of other communication channels are prohibited.

4 SPARSEVFL

In this section, we explain the SparseVFL algorithm[1], including embedding sparsification, gradient sparsification, and coding. Algorithm 1 shows the workflow of SparseVFL, including training, coding, and communication steps.

4.1 Embedding Sparsification

ReLU activation produces sparse embeddings by replacing all negative values with zeros. Other activation functions, e.g., SELU or ELU, retain negative values. Let $\sigma(\cdot) \geq 0$ be the ReLU activation, an embedding vector $\mathbf{h}_{n,m}$ is represented as follows:

$$\begin{cases} x > 0 & \Rightarrow \quad \sigma(x) = x, \quad \partial\sigma/\partial x = 1 \\ x \leq 0 & \Rightarrow \quad \sigma(x) = 0, \quad \partial\sigma/\partial x = 0 \end{cases} \tag{3}$$

$$\begin{aligned} \mathbf{h}_{n,m} =& [h_{n,m,1}, ..., h_{n,m,D_h}] \\ =& [\sigma(\theta_{m,1}(\mathbf{x}_{n,m})), ..., \sigma(\theta_{m,D_{h,m}}(\mathbf{x}_{n,m}))], \end{aligned} \tag{4}$$

where $\theta_{m,k}$ is a function. This ReLU activation is included in the fwdSparseEmb function at Line 7 in Algorithm 1. Client m sparsifies $\mathbf{h}_{n,m}$ and sends $\mathbf{E}'_m = \{h_{n,m,k} | h_{n,m,k} > 0\}$ to the server, instead of

[1] https://github.com/docomoinnovations/SparseVFL

Algorithm 1: SparseVFL

1 **Input**:$\mathbf{x}_1, ..., \mathbf{x}_M, \mathbf{y}$
2 **Parameter**: λ, Q
3 **Output**: server sv, list of clients cl
4 **while** not converged **do**
5 **for each** $cl_m \in cl$ **do in parallel**
6 `// Client`
7 $\mathbf{E}_m \leftarrow cl_m$.fwdSparseEmb($\mathbf{x}_m$)
8 $\mathbf{E}'_m, \mathbf{H}_m, \mathbf{T}_m \leftarrow cl_m$.encode($\mathbf{E}_m, Q$)
9 cl_m.storage.save($\mathbf{H}_m, \mathbf{T}_m$)
10 cl_m.send($m, \mathbf{E}'_m, \mathbf{H}_m, \mathbf{T}_m$)
11 `// Server`
12 $\mathbf{E}'_m, \mathbf{H}_m, \mathbf{T}_m \leftarrow sv$.receive($m$)
13 $\mathbf{E}_m \leftarrow sv$.decode($\mathbf{E}'_m, \mathbf{H}_m, \mathbf{T}_m, Q$)
14 sv.storage.save($m, \mathbf{E}_m, \mathbf{H}_m, \mathbf{T}_m$)
15 **end for**
16 `// Server`
17 $\tilde{\mathbf{y}} \leftarrow sv$.forward($sv$.storage)
18 $l \leftarrow sv$.loss($\tilde{\mathbf{y}}, \mathbf{y}$)+$\lambda \times sv$.normL1($sv$.storage)
19 **for each** $cl_m \in cl$ **do in parallel**
20 `// Server`
21 $\mathbf{G}_m \leftarrow sv$.backward($l, m$)
22 $\mathbf{H}_m, \mathbf{T}_m \leftarrow sv$.storage.load($m$)
23 $\mathbf{G}'_m \leftarrow sv$.encSparseGrad($\mathbf{G}_m, \mathbf{H}_m, \mathbf{T}_m, Q$)
24 sv.send($m, \mathbf{G}'_m$)
25 `// Client`
26 $\mathbf{G}'_m \leftarrow cl_m$.receive()
27 $\mathbf{H}_m, \mathbf{T}_m \leftarrow cl_m$.storage.load()
28 $\mathbf{G}_m \leftarrow cl_m$.decode($\mathbf{G}'_m, \mathbf{H}_m, \mathbf{T}_m, Q$)
29 cl_m.backward($\mathbf{G}_m$)
30 **end for**
31 **end while**

$\mathbf{E}_m = \{h_{n,m,k}\}$, where $|\mathbf{E}'_m| \leq |\mathbf{E}_m|$ holds, where $|\cdot|$ is the number of elements.

Unlike the typical L1-regularization [9, 20], which adds the L1-norm of the model parameters $||\theta_m||_1$ to produce sparse model parameters, SparseVFL adds the L1-norm of the embedding vectors $||\mathbf{h}_{n,m}||_1$ to the loss function and produces sparse embedding vectors as output. Our loss function is written as

$$L := \frac{1}{N}\sum_{n=1}^{N} l(\theta_0, \mathbf{h}_{n,1}, ..., \mathbf{h}_{n,M}; \mathbf{y}_n) + \frac{\lambda}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} ||\mathbf{h}_{n,m}||_1, \quad (5)$$

where λ is a hyperparameter to control embedding sparsity. Equation 5 is at Line 18 in Algorithm 1.

4.2 Gradient Sparsification

In back-propagation, a gradient element $\partial L/\partial h_{n,m,k}$ corresponding to an embedding element $h_{n,m,k} = 0$ can be ignored. We call this **masked-gradient**. Masked-gradient is implemented in the encSparseGrad function at Line 23 in Algorithm 1. Typical optimizers such as Adam, SGD, and RMSprop [19] can use masked-gradient for the following reason.

Proof. According to Equations 3 and 4, under the condition that ReLU is used in the last layer of the client model, the gradient element corresponding to the embedding element $h_{n,m,k}$ at the parameter θ_m is represented as follows.

$$\theta_{m,k}(\mathbf{x}_{n,m}) > 0 \Rightarrow \quad h_{n,m,k} > 0, \qquad \frac{\partial h_{n,m,k}}{\partial \theta_m} \in \mathbb{R} \quad (6)$$

$$\theta_{m,k}(\mathbf{x}_{n,m}) \leq 0 \Rightarrow \quad h_{n,m,k} = 0, \qquad \frac{\partial h_{n,m,k}}{\partial \theta_m} = 0 \quad (7)$$

Therefore, Equation 2 sums up for the subset $\{n|h_{n,m,k} > 0\}$.

$$\frac{\partial L}{\partial \theta_m} = \sum_{n \in \{n|h_{n,m,k}>0\}} \frac{\partial L}{\partial h_{n,m,k}} \frac{\partial h_{n,m,k}}{\partial \theta_m} \quad (8)$$

In other words, instead of $\mathbf{G}_m = \{\partial L/\partial h_{n,m,k}\}$, the server can just return masked-gradient $\mathbf{G}'_m = \{\partial L/\partial h_{n,m,k} | h_{n,m,k} > 0\}$ to each client m, where $|\mathbf{G}'_m| \leq |\mathbf{G}_m|$ holds. □

4.3 Coding

We propose an efficient coding scheme for SparseVFL based on run-length coding [16]. Figure 2 shows the coding procedure as an example. First, we traverse the matrix of the embedding batch vertically (or horizontally), with indexing from 0 to $D_{h,m}N$, and reshape the matrix into a list. Then we extract the head indices, where non-zero values start, and the tail indices, where zero values start. The minimum information needed for decoding is as follows:

- Common in both forward and backward
  - N: #Samples of the batch (q_0-bit int scalar)
  - $D_{h,m}$: Dimension of the batch (q_1-bit int scalar)
- In forward
  - $\mathbf{E}'_m$: Non-zero embedding values (Q-bit float array)
  - $\mathbf{H}_m$: Head indices (q_2-bit int array)
  - $\mathbf{T}_m$: Tail indices (q_2-bit int array)
- In backward
  - $\mathbf{G}'_m$: Masked-gradient values (Q-bit float array)

Here, $q_0 = \lceil \log_2(N) \rceil$, $q_1 = \lceil \log_2(D_{h,m}) \rceil$, $q_2 = \lceil \log_2(D_{h,m}N) \rceil$, and Q is a hyperparameter. In the example of Figure 2, the original data ($\mathbf{E}_m$, $\mathbf{G}_m$) has 16 elements, while, the encoded data ($\mathbf{E}'_m$, $\mathbf{H}_m$, $\mathbf{T}_m$, $\mathbf{G}'_m$) has only 11 elements in total. In Algorithm 1, these coding methods are defined at Lines 8, 13, 23 and 28.

4.4 Communication Costs

Let I be the number of epochs, the data size S_m [byte] transferred between the server and client m for I epochs is represented by the following equations. For the original VFL (Algorithm α),

$$S_m^{(\alpha)} = \begin{cases} 2|\mathbf{E}_m|IQ/8 & (train) \\ |\mathbf{E}_m|IQ/8 & (valid) \\ |\mathbf{E}_m|Q/8 & (test) \end{cases} \quad (9)$$

and for SparseVFL (Algorithm β),

$$S_m^{(\beta)} = \begin{cases} (2|\mathbf{E}'_m| + |\mathbf{H}_m| + |\mathbf{T}_m|)IQ/8 & (train) \\ (|\mathbf{E}'_m| + |\mathbf{H}_m| + |\mathbf{T}_m|)IQ/8 & (valid) \\ (|\mathbf{E}'_m| + |\mathbf{H}_m| + |\mathbf{T}_m|)Q/8 & (test) \end{cases} \quad (10)$$

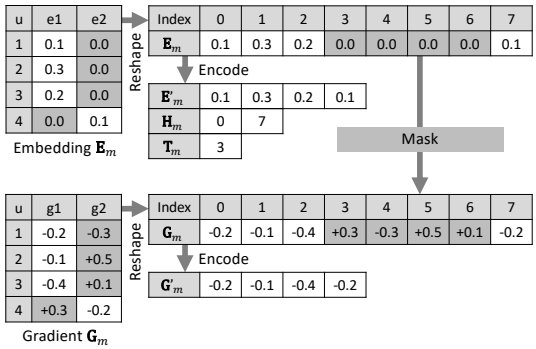


Figure 2: Coding scheme ($N = 4$ and $D_{h,m} = 2$).

Table 1: Dataset size and label setting.

Dataset	Training	Validation	Test	Label
Adult	29,304	3,257	16,281	Income>50K
Wine Quality	5,197	650	650	Quality>6
Covertype	464,809	58,101	58,102	7 classes

where *Valid* mode is used for validation steps that involve only forward-propagation for each epoch, and *Test* mode is used for the inference step with the trained models. The worst case of $S_m^{(\beta)}$ is when zeros and non-zeros alternate in an embedding batch, i.e., $|\mathbf{E}'_m| = |\mathbf{H}_m| = |\mathbf{T}_m| = |\mathbf{E}_m|/2$. In this case, Equation 10 would be

$$\hat{S}_m^{(\beta)} = \begin{cases} 2|\mathbf{E}_m|IQ/8 & (train) \\ 3/2 \times |\mathbf{E}_m|IQ/8 & (valid) \\ 3/2 \times |\mathbf{E}_m|Q/8 & (test) \end{cases} \tag{11}$$

Therefore, $S_m^{(\beta)} \leq \hat{S}_m^{(\beta)} = S_m^{(\alpha)}$ always holds for *train* mode. However, for the *valid* and *test* modes, $\hat{S}_m^{(\beta)}$ is 3/2 times larger than $S_m^{(\alpha)}$. In this case, the coding step should be disabled for the *valid* and *test* modes.

Unlike the asynchronous VFL [3], the synchronous VFL requires the server to wait for the slowest client. Let T_0 [s] be the computation time in the server, T_m [s] be the computation time in each client m, and B [bps] be the throughput of the communication line, the training time τ_B [s] is represented as $\tau_B = T_0 + T_{\hat{m}} + 8S_{\hat{m}}/B$, where $\hat{m} = \arg\max_{m \in [1,\ldots,M]} \{T_m + 8S_m/B\}$.

5 EXPERIMENTS

In this section, we evaluate the performance of our proposed SparseVFL in terms of model accuracy and communication costs. We use three real-world datasets described in Table 1, which are publicly available [1, 5, 7]. Assuming that the three clients have exclusive sets of columns, the columns are split into three sets to make them as balanced as possible. See Appendix A for details. SparseVFL does not care how the columns are divided. In this experiment, each of the client models has a single linear layer with ReLU activation. The server model has a 2-layer MLP. We use four Adam optimizers for each model and a cross-entropy loss function for $l(\cdot)$ in Equation 5.

In this experiment, we run the algorithms (including both server and client) in a single process in a single machine. The server procedure and the client procedure exchange intermediate data by saving and loading the data on the SSD. The machine specifications are as follows: Ubuntu 18.04.6 LTS, Intel Xeon CPU E5-2620 v4 @ 2.10 GHz, 32 CPUs, RAM 251 GB, NVIDIA GeForce GTX 1080 (GPU 8 GB), CUDA 11.6. One process uses one CPU and one GPU.

Table 2: Ablation experiments for SparseVFL-16.

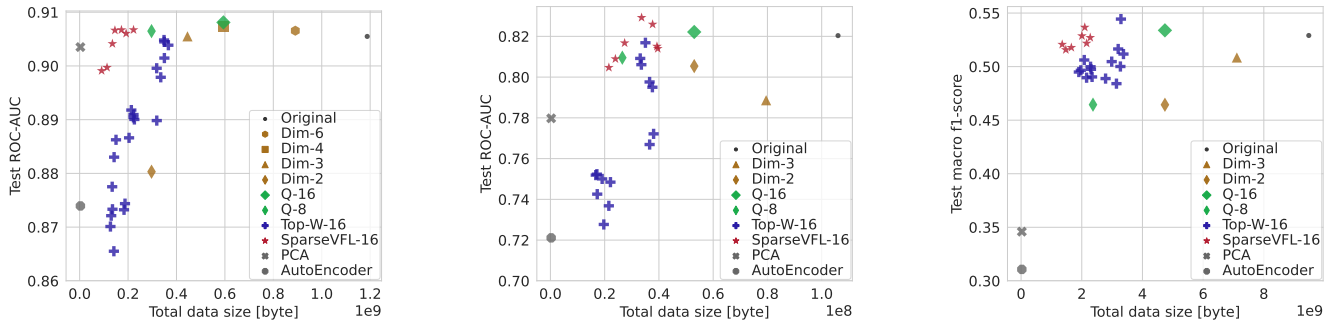
Activation	Norm	Traversal	Size	ROC-AUC
ReLU	**L1**	**Vertical**	**223**	0.9067
SeLU	**L1**	**Vertical**	594	0.9063
eLU	**L1**	**Vertical**	594	0.9067
ReLU	L2	**Vertical**	302	0.9056
ReLU	-	**Vertical**	351	0.9077
ReLU	**L1**	Horizontal	277	0.9067

5.1 Results

Table 2 shows the result of the ablation experiments for SparseVFL with quantization bits $Q = 16$ (SparseVFL-16) on Adult dataset. We can see that the data size more than doubles when ReLU is replaced by SeLU. ReLU has the largest contribution (−371) in terms of data reduction, followed by L1-norm (−128), and vertical traversal used in run-length coding (−54), while maintaining almost the same ROC-AUC. Figure 4 shows the embeddings of the first 32 samples from client 1. In this example, all samples have non-zero values in one or more dimensions. In particular, all but one sample have zero in the 4th dimension. For this reason, vertical traversal is more efficient than horizontal traversal. Although order randomness is necessary for learning stability, compression efficiency increases as the same input features become more consecutive.

We compare the performance of SparseVFL-16 with several existing algorithms. Original is the VFL algorithm described in Section 3 with $Q = 32$ and $D_{h,m} = 8$ (Adult) or $D_{h,m} = 4$ (the other datasets). Q-16 and Q-8 quantizes Original to 16-bit and 8-bit, respectively. Dim-$D_{h,m}$ is the same as Original but takes $D_{h,m} \in \{2, 3, 4, 6\}$. PCA and AutoEncoder use the methods proposed in [13] with $Q = 32$, where AutoEncoder has two linear layers for encoder and decoder, respectively. Top-W-16 uses our proposed techniques described in Sections 4.1 and 4.3 for embeddings. While, for gradients, Top-W-16 uses the approach proposed in [15, 17, 22], which keeps the $W \in \{1024, 2048, 4096\}$ elements with the largest absolute magnitudes in a gradient batch and sends the W elements and their indices. SparseVFL-16 is our proposed approach. For Top-W-16 and SparseVFL-16, $Q = 16$ and $\lambda \in \{0.01, 0.0158, 0.0251, 0.0398, 0.0631, 0.1, 0.1585\}$. Other hyperparameters are set as follows: learning rate = 0.01, batch size = 1024, and the number of epochs = 200. These hyperparameters are the same for all the parties.

Figure 3 shows the experimental results. The x-axis represents the total data size [byte] transferred between the server and the clients for 200 epochs, and the y-axis is the accuracy (ROC-AUC or macro f1-score) on the test data, i.e., the upper left corner indicates an efficient algorithm. For all the three datasets, SparseVFL-16 outperforms the other algorithms in terms of (1) more accurate



(a) Adult **(b) Wine Quality** **(c) Covertype**

Figure 3: Comparison of data reduction algorithms.

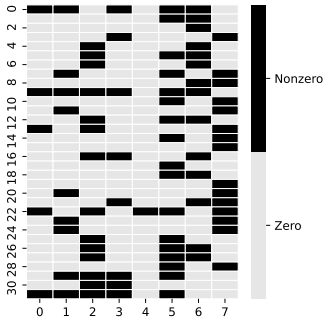


Figure 4: Embeddings by SparseVFL-16 on Adult dataset.

Table 3: Computation time T_m, data size S_m, and training time τ_B for the Adult dataset.

Algorithm	m	T_m [s]	S_m [MB]	τ_{10M} [s]	τ_{100M} [s]
Original	0	34.9	–	365.0	80.0
	1	12.4	396		
	2	13.4	396		
	3	12.4	396		
SparseVFL-16 ($\lambda = 0.01$)	0	50.9	–	135.5	77.1
	1	16.6	85		
	2	19.6	80		
	3	17.9	58		

with the same data size or (2) same accuracy with less data size. For example, in the Adult dataset, SparseVFL-16 with $\lambda = 0.01$ could produce a model with ROC-AUC 0.9067 with 223 MB, while Top-W-16 with $W = 2048$ and $\lambda = 0.0251$ has the lower score 0.8900 with almost the same size 226 MB, and Q-8 has the lower score 0.9065 with the larger size 342 MB. For each dataset, SparseVFL-16 with the highest score could reduce 68–81% of the data size of Original. Therefore, we can see that SparseVFL is effective in reducing the communication data size. PCA and AutoEncoder require very little data because they send embeddings only once and no gradients between the server and clients, however, the accuracy is worse than the other algorithms because the client models are not optimized.

Table 3 shows the comparison of training time for Original and SparseVFL-16 for the Adult dataset. SparseVFL-16 can reduce the training time τ_B by 63% when $B = 10$ Mbps and by 4% when $B = 100$ Mbps.

6 CONCLUSION AND FUTURE WORK

In this work, we proposed a communication-efficient VFL called SparseVFL, which exploits the properties of ReLU, L1 norm for embedding vectors, and masked gradient to produce sparse embeddings and gradients. SparseVFL doesn't depend on the neural network architecture, as long as it meets the SparseVFL requirements. We expect that SparseVFL will facilitate data sharing between organizations and contribute to social value.

In subsequent research, we will deploy the parties on the distributed machines to evaluate the communication costs in our real-world environment. In addition, the current SparseVFL uses synchronous FL, where the slowest client affects the overall training time. In order to further reduce the training time, we will work on the asynchronous SparseVFL.

REFERENCES

[1] Jock A Blackard and Denis J Dean. 1999. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and electronics in agriculture* 24, 3 (1999), 131–151.

[2] Timothy J Castiglia, Anirban Das, Shiqiang Wang, and Stacy Patterson. 2022. Compressed-VFL: Communication-Efficient Learning with Vertically Partitioned Data. In *International Conference on Machine Learning*. PMLR, 2738–2766.

[3] Tianyi Chen, Xiao Jin, Yuejiao Sun, and Wotao Yin. 2020. Vafl: a method of vertical asynchronous federated learning. *arXiv preprint arXiv:2007.06081* (2020).

[4] Xing Chen, Jingtao Li, and Chaitali Chakrabarti. 2021. Communication and Computation Reduction for Split Learning using Asynchronous Training. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. https://doi.org/10.1109/sips52927.2021.00022

[5] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. 2009. Modeling wine preferences by data mining from physicochemical properties. *Decision support systems* 47, 4 (2009), 547–553.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[7] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[8] Zhenan Fan, Huang Fang, Zirui Zhou, Jian Pei, Michael P Friedlander, and Yong Zhang. 2022. Fair and efficient contribution valuation for vertical federated learning. *arXiv preprint arXiv:2201.02658* (2022).

[9] Siwei Feng and Han Yu. 2020. Multi-participant multi-class vertical federated learning. *arXiv preprint arXiv:2001.11154* (2020).

[10] Fangcheng Fu, Xupeng Miao, Jiawei Jiang, Huanran Xue, and Bin Cui. 2022. Towards communication-efficient vertical federated learning training via cache-enabled local updates. *arXiv preprint arXiv:2207.14628* (2022).

[11] Otkrist Gupta and Ramesh Raskar. 2018. Distributed learning of deep neural network over multiple agents. arXiv:1810.06060 [cs.LG]

[12] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[13] Afsana Khan, Marijn ten Thij, and Anna Wilbik. 2022. Communication-Efficient Vertical Federated Learning. *Algorithms* 15, 8 (2022). https://doi.org/10.3390/a15080273

[14] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated Learning: Strategies for Improving Communication Efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*. https://arxiv.org/abs/1610.05492

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

[16] A Harry Robinson and Colin Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proc. IEEE* 55, 3 (1967), 356–364.

[17] Shaohuai Shi, Kaiyong Zhao, Qiang Wang, Zhenheng Tang, and Xiaowen Chu. 2019. A Convergence Analysis of Distributed SGD with Communication-Efficient Gradient Sparsification.. In *IJCAI*. 3411–3417.

[18] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth annual conference of the international speech communication association*.

[19] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. 2019. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics* 50, 8 (2019), 3668–3681.

[20] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[21] Guan Wang, Charlie Xiaoqian Dang, and Ziye Zhou. 2019. Measure contribution of participants in federated learning. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2597–2604.

[22] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems* 31 (2018).

[23] Kang Wei, Jun Li, Chuan Ma, Ming Ding, Sha Wei, Fan Wu, Guihai Chen, and Thilina Ranbaduge. 2022. Vertical Federated Learning: Challenges, Methodologies and Experiments. *arXiv preprint arXiv:2202.04309* (2022).

[24] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. 2019. Qpytorch: A low-precision arithmetic simulation framework. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE, 10–13.

[25] Hangyu Zhu, Haoyu Zhang, and Yaochu Jin. 2021. From federated learning to federated neural architecture search: a survey. *Complex & Intelligent Systems* 7, 2 (2021), 639–657.

Table A1: Definition of feature columns.

Adult	
Client 1	age, workclass <9>, fnlwgt, education <16>, education_num
Client 2	marital_status <7>, occupation <15>, relationship <6>, race <5>, sex <2>
Client 3	capital_gain, capital_loss, hours_per_week, native_country <42>
Wine Quality	
Client 1	fixed_acidity, volatile_acidity, citric_acid, residual_sugar
Client 2	chlorides, free_sulfur_dioxide, total_sulfur_dioxide, density
Client 3	ph, sulphates, alcohol, color
Covertype	
Client 1	elevation, aspect, slope, horizontal_distance_to_hydrology, horizontal_distance_to_roadways, horizontal_distance_to_fire_points, vertical_distance_to_hydrology
Client 2	hillshade_9am, hillshade_noon, hillshade_3pm, wilderness_area <4>
Client 3	soil_type <40>

A APPENDIX: DEFINITION OF FEATURE COLUMNS

The feature columns $\mathbf{x}_{n,m}$ of each client $m \in \{1, 2, 3\}$ are defined in Table A1. The column names with angle brackets represent categorical columns and its number of classes, otherwise numerical columns. For example, Column "workclass" has 9 classes. In this work, the categorical columns and numerical columns are preprocessed using one-hot encoding and min-max encoding, respectively.

B APPENDIX: MATH NOTATIONS

Tables A2 and A3 show summary of math notations.

Received 18 May 2023; revised 18 May 2023; accepted 18 May 2023

Table A2: Sample-level math notations.

Symbol	Meaning
M	The number of clients
m	Client id
N	The number of samples
n	Sample id
$\mathbf{x}_{n,m}$	Feature vector
$D_{x,m}$	Dimension of $\mathbf{x}_{n,m}$
h_m	Model of client m
θ_m	Parameter of client model h_m
$\mathbf{h}_{n,m}$	Embedding vector
$D_{h,m}$	Dimension of $\mathbf{h}_{n,m}$
h_0	Server model
θ_0	Parameter of server model h_0
$\tilde{\mathbf{y}}_n$	Predicted vector
$\mathbf{y}_n$	Label vector
L, l	Loss function
σ	ReLU activation
λ	Sparsification parameter

Table A3: Batch-level math notations.

Symbol	Meaning
$\mathbf{E}_m$	Batch of $h_{n,m,k}$
$\mathbf{E}'_m$	Subset of $\mathbf{E}_m$ where $h_{n,m,k} > 0$
$\mathbf{G}_m$	Batch of $\partial L / \partial h_{n,m,k}$
$\mathbf{G}'_m$	Subset of $\mathbf{G}_m$ where $h_{n,m,k} > 0$
q_0, q_1, q_2	Bit length
Q	Quantization parameter
$\mathbf{H}_m$	Head indices
$\mathbf{T}_m$	Tail indices
I	The number of epochs
S_m	Data size transferred between the server and client m
T_0	Computation time in the server
T_m	Computation time in client m
B	Throughput of the communication line
τ_B	Training time
W	The number of remaining gradients