FeGAN: Scaling Distributed GANs

Rachid Guerraoui, Arsany Guirguis,
Anne-Marie Kermarrec
EPFL
firstname.lastname@epfl.ch

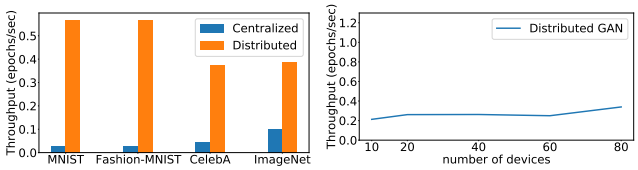
Erwan Le Merrer
Univ Rennes, Inria, CNRS, Irisa
erwan.le-merrer@inria.fr

Abstract

Existing approaches to distribute Generative Adversarial Networks (GANs) either *(i)* fail to scale for they typically put the two components of a GAN (the *generator* and the *discriminator*) on different machines, inducing significant communication overhead, or *(ii)* they face GAN training specific issues, exacerbated by distribution.

We propose FeGAN, the first middleware for distributing GANs over hundreds of devices addressing the issues of mode collapse and vanishing gradients. Essentially, we revisit the idea of *Federated Learning*, co-locating a generator with a discriminator on each device (addressing the scaling problem) and having a server aggregate the devices' models using *balanced sampling* and *Kullback-Leibler (KL) weighting*, mitigating training issues and boosting convergence.

Through extensive experiments, we show that FeGAN generates high–quality dataset samples in a scalable and devices' heterogeneity tolerant manner. In particular, FeGAN achieves up to 5× throughput gain with 1.5× less bandwidth compared to the state–of–the–art GAN distributed approach (named MD–GAN), while scaling to at least one order of magnitude more devices. We demonstrate that FeGAN boosts training by 2.6× w.r.t. a baseline application of Federated Learning to GANs, while preventing training issues.



(a) **(b)**

Figure 1. (a) A Parameter Server–based deployment [16] with one generator and distributed discriminators (on 21 machines) outperforms the centralized approach. Yet, (b) it does not scale, due to communication bottlenecks.

1 Introduction

GANs enable learning the statistical distribution of a target dataset and generating new samples from that dataset on demand. This feature can be used in a wide range of applications such as generating pictures from text descriptions [38], producing videos from still images [45], or increasing at will an image resolution [25]. Other applications, such as DeepFakes generation [27] are as impressive as they are critical for society.

At the core of a GAN lie two deep neural networks: the *generator* and the *discriminator*. They confront each other in a game: the generator aims at generating data that looks real (*i.e.*, coming from the real data distribution) to feed the discriminator. When the latter can no longer distinguish real data from the generated one, the training stops. The generator has then captured the data distribution and has reached the point where it can generate new samples.

Training GANs is resource and time consuming. For instance, it takes up to 48 hours to train a GAN to learn the distribution of a 512x512 image dataset on Google TPUs v3 devices [8]. Considering the number of potential applications, it is then of paramount importance to improve this training time. One obvious way to scale a system is to distribute the computing load on multiple devices. MD-GAN [16] has been recently proposed as a way to distribute GANs. MD-GAN leverages a single generator, at a central location, and distributes the discriminator across multiple devices (this architecture is also adopted by [52]). Such an architecture follows the celebrated *parameter server* model [26], where the server is the generator and the workers are discriminators. Figure 1a illustrates the benefit of such an architecture with 21 machines on four datasets (MNIST [3] and Fashion-MNIST [1], often used as baselines, CelebA [29] and ImageNet [11], two state-of-the art datasets for image-based applications). Clearly, distributing the training improves the system throughput, defined as the number of epochs (the processing of all data samples in the dataset) the system handles per second. Yet, due to the presence of a single centralized generator, **this architecture does not scale**, *i.e.*, adding devices does not improve the throughput, as illustrated in Figure 1b. This is due to the huge volumes of data to be transferred over the network to synchronize the generator and the multiple discriminators, making the central generator a bottleneck. Clearly there is a huge room for improvement.

Another architecture for distributed training is Federated Learning (FL) [32], in which training happens on the edge devices that own private data, assisted by a central server. Combining GAN training with this architecture can yield impressive applications on edge devices including text-to-image translation and generating new human poses. Yet, approaches that followed such an architecture for GAN training (for *e.g.*, running diagnostics [5] or for data privacy

goals [44]) also reported **difficulties due to learning divergence, vanishing gradients, and mode collapse problems**: *learning divergence* happens when neither the generator nor the discriminator reaches its goal, *i.e.*, Nash equilibrium is not reached [4]. A second salient problem is *vanishing gradients*, that occur when the discriminator is much more powerful than the generator, which, in this case, always fails to generate convincing samples to the discriminator, where the feedback from the discriminator does not help the generator to learn [33]. Last but not least, the problem of *mode collapse* happens when the generator learns to generate only a few classes of data input rather than learning the true distribution of data [9]. Such problems manifest clearly in the FL context. For instance, the server cannot communicate with all devices at every round for scalability reasons; some form of device sampling is needed. Yet, such sampling should not be uniform because of the data distribution skewness on the devices hosting the GAN model. Unfortunately, such data skewness can easily lead to mode collapse. Thus, the mere application of the FL approach [47, 48, 51] to GANs training is bound to failure [15] or to underperformance, as we shall confirm experimentally.

Contributions. We propose FeGAN, a distributed middleware that enables GANs to scale and cope with GAN specific issues such as mode collapse and vanishing gradients. Our main contributions are threefold.

1) We revisit the FL paradigm, normally dedicated to deep networks, to make it suitable to GANs. Essentially, we fully distribute both the generator and the discriminator so that a private GAN can be locally trained on each device. This helps scale the system and prevent the vanishing gradients problem, as we show in Sections 5 and 6.

2) We design mechanisms to make FeGAN resilient to GAN specific issues. In particular, we devise two techniques that are designed to resist the mode collapse and the learning divergence problems in a distributed setup. FeGAN prioritizes updates from certain nodes over some others, using *Kullback-Leibler (KL) weighting* scheme, and it carefully schedules the application of devices' updates on the global trained model, using the *balanced sampling* scheme. Both schemes not only boost the learning quality but also help save compute time and communication resources. FeGAN is tolerant to devices' heterogeneity in terms of memory and compute power and to server and network failures through periodic checkpointing of the learning state.

3) We conduct an extensive experimental evaluation of FeGAN and compare it to a state–of–the–art GAN distribution approach (MD-GAN), as well as with a centralized approach and a baseline application of Federated Learning to GANs.

2 Background

2.1 Generative Adversarial Networks

A GAN [14], made of generator $\mathcal{G}$ and discriminator $\mathcal{D}$, targets the learning of a dataset distribution in space X, where $\boldsymbol{x} \in X$ follows a distribution probability P_{data}. The generator is modeled by the function $\mathcal{G}_{\boldsymbol{w}} : R^{\ell} \longrightarrow X$, where $\boldsymbol{w}$ contains the parameters of its neural network $\mathcal{G}_{\boldsymbol{w}}$, and ℓ is fixed. Similarly, for the discriminator $\mathcal{D}_{\boldsymbol{\theta}} : X \longrightarrow [0,1]$ where $\mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})$ is the probability that $\boldsymbol{x}$ is a data from the training dataset, and $\boldsymbol{\theta}$ contains the parameters of the discriminator $\mathcal{D}_{\boldsymbol{\theta}}$. The objective consists in finding the parameters $\boldsymbol{w}^*$ for the generator: $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \max_{\boldsymbol{\theta}}(A_{\boldsymbol{\theta}} + B_{\boldsymbol{\theta},\boldsymbol{w}})$, with $A_{\boldsymbol{\theta}} = \mathbb{E}_{\boldsymbol{x}\sim P_{\text{data}}}[\log \mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})]$ and $B_{\boldsymbol{\theta},\boldsymbol{w}} = \mathbb{E}_{\boldsymbol{z}\sim \boldsymbol{\mathcal{N}}_{\ell}}[\log(1 - \mathcal{D}_{\boldsymbol{\theta}}(\mathcal{G}_{\boldsymbol{w}}(\boldsymbol{z}))]$, where $\boldsymbol{z} \sim \boldsymbol{\mathcal{N}}_{\ell}$ means that each entry of the ℓ-dimensional random vector $\boldsymbol{z}$ follows a normal distribution with fixed parameters. In this equation, $\mathcal{D}$ adjusts its parameters $\boldsymbol{\theta}$ to maximize $A_{\boldsymbol{\theta}}$, *i.e.*, the expected good classification on real data and $B_{\boldsymbol{\theta},\boldsymbol{w}}$, the expected good classification on generated data. $\mathcal{G}$ adjusts its parameters $\boldsymbol{w}$ to minimize $B_{\boldsymbol{\theta},\boldsymbol{w}}$ ($\boldsymbol{w}$ does not have impact on A), which means that it tries to minimize the expected good classification of $\mathcal{D}$ on generated data. This competitive scheme ends, if convergence occurs, to the learning of the dataset distribution P_{data}.

2.2 Distributed Machine Learning

The parameter server architecture [26] was a milestone for distributed ML, introduced to greatly accelerate the computation over a single centralized process. In this model, the *parameter server* allocates data (also called *batches*) to *workers* and orchestrates the learning process. Workers run computations locally and send back their gradient errors to the server that aggregates them in a global model. In turn, workers pull the up–to–date model from the server and iterate until the convergence of the global model is reached.

The Federated Learning (FL) [32] approach however seeks to leverage a large crowd of edge devices that each owns private data, without imposing any movement of the data (unlike with the parameter server). The devices collaboratively train a global model, which is hosted on a centralized server. The server decides which subset of the devices should participate in a learning round through random selection. Selected devices, in turn, send their updates to the server after a number of learning iterations over their private data. Typically, the server aggregates the devices' updates using *Federated Averaging* [32]. The accuracy of FL is shown in [53] to be negatively impacted by the imbalanced data distribution on the devices.

3 FeGAN Design and Implementation

In this section, we first provide the system setup and an overview of FeGAN. Then, we focus on our design choices, which make FeGAN scalable and resilient to GAN training issues, and the system aspects of FeGAN.

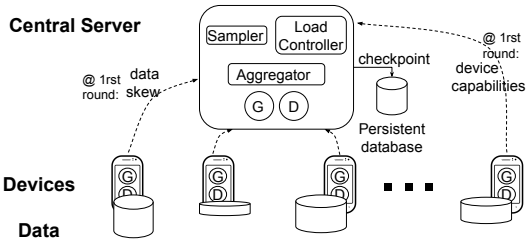


Figure 2. FeGAN architecture: $\mathcal{D}$ and $\mathcal{G}$ are co–located on all devices. Dashed lines denote the communication of metadata about devices' dataset and specifications. Data across devices might be unbalanced and skewed.

3.1 System Setup

FeGAN revisits the general Federated Learning (FL) paradigm [21] which consists in (1) a central server (hereafter, server) hosting the up-to-date global model and (2) a set of computing devices that can be mobile phones or computing nodes (hereafter called devices). Figure 2 depicts the FeGAN architecture. The training model is composed of two neural networks: the generator $\mathcal{G}$ and the discriminator $\mathcal{D}$. The server orchestrates the communication load by selecting which devices should contribute to updating the model at a given round. The server is also in charge of aggregating the computation from devices to update the global model. Each group of local iterations (on devices), ending with a global model update, is called an *FL round* (or simply *round*). On top of this, training and system specifics are designed to reach our scalability and resilience goals.

Each device owns a GAN locally, composed of a local generator and a local discriminator. Data stored on each device remains local: only the output of the local computation is sent to the server. Given its local nature, data might be unbalanced and possibly not identically nor independently distributed (*non-iid*) across devices. Yet, the server can gather metadata from devices regarding the number of local samples per class. Hence, we assume each device can label its data or at least can model its own data distribution, using generative or clustering schemes like Gaussian Mixture Models (GMM) [39, 46] or K-means [13].

Devices only communicate with the server. As they do not communicate with each other, they do not need to trust each other. Communication links between the devices and the server could be unreliable and asymmetric with limited bandwidth [21]. The server can keep the devices' updates secure using off–the–shelf *secure aggregation* protocols [6].

3.2 The FeGAN Algorithm

Figure 3 depicts a running example of FeGAN.

Initialization. Before starting the training process, each device informs the server of its local data distribution (typically how many classes and how many samples per class it owns locally). Such a step could be repeated should the dataset change during the training process. The goal of collecting metadata from all devices is for the server to account for the data imbalance across devices.

To this end, the server computes the *Kullback–Leibler (KL) divergence* [22, 23] scores, which reflect the degree of divergence of the devices' local data distribution from the global distribution. In particular, the KL divergence of device k is computed as follows: $D_{KL}(P_k||Q) = \sum_{x \in \mathcal{X}} P_k(x) \log(\frac{P_k(x)}{Q(x)})$, where P_k is the normalized vector of samples per class x at device k, Q is the normalized vector of the total number of samples per class (in the global dataset), and $\mathcal{X}$ is the collection of all classes. For instance, $P_k(x) = {}^{n_{k_x}}/_{n_k}$, where n_{k_x} is the number of samples of class x at device k, and n_k is the total number of samples at device k. The server then assigns a score s_k to each device k as follows: $s_k = \frac{n_k}{n} \times D_{KL}(P_k||Q)$, where n_k is the total number of samples per device k and n is the total number of samples in the dataset.

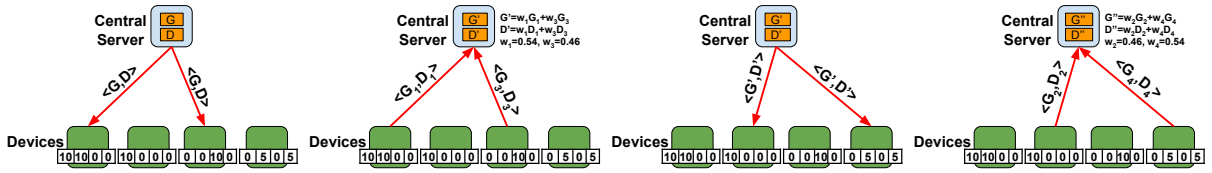
All devices also send to the server their capabilities, in terms of memory size and compute power available for training. The server then uses this information to choose a training load for each device. As for the current version of FeGAN, the server follows a linear regression model to compute this load, which we assume fixed throughout the training process.

Server operation. The server starts each FL round by choosing a group of devices, using balanced sampling (Section 3.3), to update collectively the model state. The server then asks the sampled devices to contribute to the current round and assigns to each device the pre–computed training load. At the end of the FL round, the server collects the updates of the devices and aggregates them using the KL weighting scheme (Section 3.3). The server also weighs such updates by the devices' respective assigned batch sizes (Section 3.4). The server repeats this procedure till convergence.

Device operation. Devices remain idle until they are contacted by the server to participate in one FL round. A device selected for a given round receives the updated model state (of $\mathcal{D}$ and $\mathcal{G}$) and the values for E, the number of local training iterations to be done by this device, and B, the training batch size. After running E local training iterations, each device sends back the updated model to the server.

3.3 FeGAN Design Choices

Co-locating networks. Training deep neural networks with backpropagation sometimes encounters the vanishing gradients problem, which happens when the computed gradients are extremely small, hindering the update of the network's parameters and hence, stopping the learning. In the context of GANs, such a problem happens usually due to having a very strong discriminator [33]. Given that training GANs is usually formulated as a *game* between two players/networks, typically a network gains *strength* with more *experience* (i.e., with more training iterations). This *strength* then reflects



(a) FL round 1 starts **(b)** Server updates G and D **(c)** FL round 2 starts **(d)** Server updates G and D

Figure 3. Running example of how FeGAN works. Assume a dataset of 50 samples in 4 classes, distributed on 4 devices with $C = 0.5$. In the first round, the server chooses device 1 (with the highest number of samples) and device 3 (with samples from a class not represented by device 1). When the devices reply, the server updates the model, applying *KL_weighting* on their replies. In the second round, the server chooses the unsampled devices to maintain fairness and the training continues as usual.

how good the network is (either the generator or the discriminator) in doing its task. In the situation of a powerful discriminator, the generator fails to learn as it cannot get much information from the discriminator's feedback. Hence, the discriminator wins in this case, *i.e.*, the generator always fails to generate *fake* samples that can trick the discriminator. In the centralized setup, this problem can be mitigated by choosing the *correct* set of hyper-parameters and loss functions [14, 30]. Yet, this is not sufficient in the distributed setup; it is also crucial to balance the strength of both networks. FeGAN solves this problem by co-locating the discriminator with the generator so that both networks can be trained simultaneously, increasing their strength at the same time.

Importantly, co–locating networks reduces the communication overhead, compared to other designs that centralize the generator in the server while distributing the discriminators on the devices, *e.g.*, [5, 16]. In the latter case, the distributed discriminators should periodically send their feedback to the centralized generator, whose size is the same as that of the input space that is desired to be learned, *e.g.*, pixels of an image. By co-locating networks, FeGAN drastically reduces such an overhead, as we show in Section 6.

Kullback-Leibler weighting. Data is more likely to follow different distributions on different devices and hence, using vanilla averaging of devices' local models might lead to learning divergence [28]. Vanilla *Federated Averaging (FedAvg)* [32] is also not sufficient for convergence in some cases [53] as it does not take data distribution discrepancies into consideration while weighting the devices' updates.

FeGAN mitigates learning divergence by relying on Kullback-Leibler (KL) weighting, which is applied as follows. Upon receipt of the updated models from the devices, the server weights them by applying the *softmax* function on the *KL divergence score* of each device. More specifically, the model received from device k is given a weight w_k as follows: $w_k = \frac{\exp^{-s_k}}{\sum_{i \in \mathcal{I}} \exp^{-s_i}}$, where s_k is the pre–computed KL score of device k, and $\mathcal{I}$ is the set of devices contributing to the current FL round. The *softmax* function acts as a normalization[1] to the weights given to the devices' updates so as to make these weights sum to 1. In other words, the output of the *softmax* function (*i.e.*, the weights) can be seen as the probability of the contribution of each device to updating the global model state. The negative sign given to the score is to *penalize* the devices with more divergence from the global data distribution as their updates are less useful than the other devices' updates. Such a design of a weighting scheme is crucial to learn the *global distribution* of data.

Balanced sampling. Mode collapse [9] is arguably the hardest problem that one can encounter while training a GAN. Such a problem happens when the generator learns only partially the data distribution and hence, can only generate data from a limited number of classes. In our distributed, uncontrolled environment, the situation could be exacerbated due to the skewness of data distribution on the devices. Our *balanced sampling* scheme aims at solving this problem.

At each FL round, the server *samples* $C \times n_d$ devices to do the training procedure, where C denotes the fraction of devices doing the training in each round with $C \in]0, 1]$, and n_d is the total number of devices. Note that unlike vanilla Federated Learning [21], in FeGAN, the server starts the sampling procedure, by choosing devices with specific data distribution to avoid falling into mode collapse. The novelty of FeGAN's sampling is that it favors (1) devices with more samples, (2) devices with a balanced dataset (*i.e.*, almost equal number of samples per class), and (3) unsampled devices in the previous rounds. Thus, our sampling technique always ensures a balanced number of samples per class in the updates applied to the model ($\mathcal{G}$ and $\mathcal{D}$). Such a technique plays an important role in the robustness and the convergence speed, as we show in our evaluation.

Technically, FeGAN applies balanced sampling as follows: 1) The server maintains a priority queue with the number of accumulated samples per class from sampled devices (from previous rounds) stored in it. 2) At each round, the server picks the class with the least number of accumulated samples and chooses a device that declares to have this class to include it in the new round. 3) The server accounts for the samples

[1] We tried other normalization schemes (*e.g.*, linear averaging and step-wise averaging), and we found that *softmax* normalization gives the best results.

from other classes that will be included in the new round (due to including the chosen device) and updates the queue accordingly. 4) The server repeats this procedure until it samples $C \times n_d$ devices. It then sends the updated state of the model, *i.e.*, $\mathcal{D}$ and $\mathcal{G}$ networks, to the sampled devices.

3.4 FeGAN System Aspects

Handling heterogeneity. Devices in the wild are not equal in many respects, including their hardware specifications (*e.g.*, compute power and memory) and the available network bandwidth. Tolerating device heterogeneity in an optimal manner is an open problem with many heuristics [28]. Because we are targeting GANs training, it is then crucial to design a solution that handles device heterogeneity without falling into GAN–specific problems. For instance, a solution that prefers sampling strong devices over weak devices *e.g.*, [34] can lead to mode collapse if the weak devices owns data classes that are not represented by the strong devices.

FeGAN handles this issue by controlling the *training load* given to the contributing devices in one round. More specifically, the central server dictates the number of iterations and the batch size used for local training on devices. The server chooses less number of iterations and smaller data batches for weaker devices. This helps mitigate the straggler effect [26], which happens when some devices are significantly slow, degrading the whole system throughput. In the aggregation phase, devices' updates are also weighted based on the load given to each of them. Such weights are multiplied by the KL weights (w_k) explained in Section 3.3.

Fault tolerance. The central server represents a single point of failure: if it crashes, without any fault tolerance technique, the whole learning stops, as devices do not communicate nor trust each other. FeGAN uses periodic checkpointing for server's fault tolerance. Initially, FeGAN starts multiple servers, which number can be configured, yet, only one of them acts as a primary and the rest are backups. Only the primary communicates with the devices while other servers remain idle. After each FL round, the learning state is stored in a persistent database. This includes the state of both networks, $\mathcal{G}$ and $\mathcal{D}$, the optimizer, the number of epochs passed, the data distribution on the devices, the devices capabilities, and the history of chosen devices in all previous rounds.

In the case of the crash of the primary server, a backup server takes over. First, it loads the latest learning state from the persistent database and announces to the devices that it is the new primary. The new primary then starts a new round by choosing a new set of devices to do the training.

Network failure is detected through a standard timeout mechanism. Such a failure could be either due to slow communication, network partition, or a crashing device. In the event of a failure, the server calls for completion or abortion of the current round. Such a decision relies purely on the server configuration and the current learning state. The server then records the actual devices participated in this round rather than the intended set of devices to participate.

Implementation. We implemented FeGAN on PyTorch [36]. We integrated our implementation with already-existing public implementations of different GAN architectures implemented for training on one machine. We now describe the communication abstractions we implemented to allow the distribution of GAN via FeGAN. Our source code will be publicly available soon.

We rely on the notion of distributed groups as instructed by the PyTorch distributed backend. Any invoked communication abstraction takes as an input a *group* defining the set of nodes that will be involved in this communication. As we rely on a centralized architecture (see Figure 2), we allow only the centralized server to create groups to avoid any conflicts with the devices and hence, achieve strong consistency on the group formation.

We introduce two abstractions that can be used off–the–shelf directly by both the server and the devices: *multicast model* and the *average models*. *Multicast model* is used by the server at the beginning of any FL round to multicast the current state of the model to a group of devices (similar to *map* in MapReduce notation [10]). To comply with PyTorch distributed runtime, the chosen devices should also invoke this abstraction (*i.e.*, *multicast model*) to receive the models from the server. We use the NCCL backend of PyTorch [35] so that the models stored on a GPU need not to be copied to the CPU memory before being sent back to the devices and hence, saving time and memory[2].

The second abstraction, *average models*, is used at the end of an FL round (similar to *reduce* in MapReduce notation). Typically, this function is invoked by the server to aggregate devices' updates. As our model is synchronous, the server waits for all contributing devices to send back their results before proceeding to the next round. If our weighting technique is not activated, we use the *reduce* operation from the distributed backend with *ReduceOp.SUM* as a reduce operator and then, divide by the number of contributing devices to this round to get the average of the updates. We use this operation as it is always faster than the other operations. Otherwise, we use the *gather* operation to first collect all the updates from the devices and then apply the weights calculated by the server. The main problem with the latter operation is that it does not work on GPUs[3] and hence, the updates need to be first copied to the main memory before sending them to the server. At the other end, the server collects all the updates, copy them to the GPU memory, and then calculates the weighted average to get the new model. We could not overcome this inherent problem of the PyTorch

[2]This is only true if the used abstraction is implemented by NCCL; otherwise, we use the GLOO [19] backend.

[3]https://pytorch.org/docs/stable/distributed.html.

distributed backend as *gather* is the only appropriate abstraction we can use to implement our *average model* abstraction while applying weighted averaging.

Finally, we report on the number of lines of code (LoC) required to port two publicly available[4] GAN implementations to FeGAN. Factoring out the common code for dataset partitioning, group initialization, and performance measurements (*e.g.*, *FID* calculation), it takes less than 70 LoC (which constitutes around 5% of the whole code) to port a GAN implementation to FeGAN.

4 Experimental Setup

In this section, we describe our experimental setup, baselines, and the configurations we used to evaluate FeGAN.

Testbed. We evaluate FeGAN on the Grid5000 platform [2], using machines from the same cluster. Each one has 1 CPU (Intel Xeon Gold 6126) with 2 cores, 16 GiB RAM and 2 Gbps Ethernet, and one GPU (Nvidia Tesla P100). Unless stated otherwise, experiments were run on 80 nodes. This number reaches up to 176 in some experiments.

Evaluation metrics. Although evaluating GANs has been an issue for ML practitioners, requiring human judgement to assess the quality of the generated data [7, 30, 31], robust metrics are now commonly used to assess the performance of GANs [16, 24, 52] such as *Frechet Inception Distance (FID)*. We evaluate FeGAN along two main metrics: *FID* to assess the GAN convergence (or quality of the learned distribution) and *throughput* to measure the system's efficiency. We chose FID as it is one of the most stable, robust, and widely-used metrics for GAN [7]; it was the one used by our predecessors [16, 52]. Besides, FID has an official implementation in the popular ML frameworks, including TensorFlow and PyTorch.
1. FID is a metric that computes the distance between feature vectors of real images and generated ones. The score reflects the statistical divergence between the generated images and the raw images using the *Inception-v3* model [43] used for image classification. The lower the score, the better (a 0.0 score indicates that the two groups of images are identical).
2. Throughput defines the number of updates the server can process per second. Such a metric reflects the scalability of the system as well as the efficiency of the communication.

Note that we observe training convergence with time, the number of training epochs, and FL rounds. However, in our experiments, we maintain a one-to-one mapping between the number of epochs and the number of FL rounds so, we always show only one of them for space constraints. Also, note that the shown results are the average of 6 runs per experiment; we omit error-bars for better readability.

Datasets. We evaluate FeGAN in the context of three datasets: *MNIST* [3], *Fashion-MNIST* [1], and *ImageNet* [11]. In our evaluation, we focus on image generation not only because it is very challenging and resource-demanding but also due to its multiple applications in the real world. *MNIST* and *Fashion-MNIST* both describe grey-scale images of 10 classes representing handwritten digits and clothes respectively. Each dataset has 28x28 60,000 training images and 10,000 testing images; we use the latter for computing *FID*. *ImageNet* [11] is a large dataset with around 14M images of a 256x256 resolution, dispatched into more than 21, 000 classes. ImageNet classification challenges usually use a subset of this dataset [40, 42]. We use a subset of ImageNet with 100K images, distributed among 200 classes with 500 samples per class. Reducing the dataset only allows for faster convergence. Unless otherwise stated, we use Fashion-MNIST as a default dataset throughout our evaluation.

Non-iidness. In all the considered datasets, the data is balanced: the average number of samples per class is almost the same in all classes. To our knowledge, there is no publicly available dataset with inherent *non-iidness* or imbalanced data. We designed a distribution engine to emulate imbalanced and skewed distributions of data among devices.

Our engine accepts two input parameters: *max_class* and *max_samples*. The former defines the maximum number of classes any device can have, and the latter defines the maximum number of samples per class on any device. For each device, the engine generates a random number $r\_c \in [1, \frac{\text{max\_class} \times i}{n}]$, where i is the index of the device, *i.e.*, $i \in [1, n]$ and n is the total number of devices. r_c defines the number of classes this device will have. Then, the engine randomly samples r_c classes from the dataset. Similarly, the engine (at each device) generates a random number r_s for each selected class with $r\_s \in [1, \min(i^2, \frac{\text{max\_samples} \times i}{n})]$. Finally, the engine randomly samples r_s samples for each selected class from the dataset. Note that the engine generates a different value for r_s per class. Using this engine, we managed to emulate several cases of data imbalance and *non-iid* distribution of data among devices, including the typical *non-iid* workloads reported before in the literature [16, 32, 41, 52].

GAN architecture. Without loss of generality, we evaluate FeGAN with two popular GAN architectures. We believe our work can be ported to any GAN architecture, as FeGAN is GAN-internals agnostic. The two experimented architectures are: Least Squares Generative Adversarial Networks (LSGAN) [31] for the MNIST and Fashion-MNIST datasets, and a Deep Convolutional Generative Adversarial Network (DCGAN) [37] for the ImageNet dataset. The generator network comprises of one fully-connected layer followed by three convolutional layers with a *Leaky ReLU* activation function each and a *tanh* activation function for the output. The discriminator network is composed of four convolutional layers with *Leaky ReLU* activation function followed by one fully-connected layer with a *Sigmoid* activation function at

[4] https://github.com/eriklindernoren/PyTorch-GAN.

the output. The number of input, output, and hidden neurons per layer depends on the input image size. Our experiments show that both networks do not require more than 3 MB of memory for the storage of their architecture and weights.

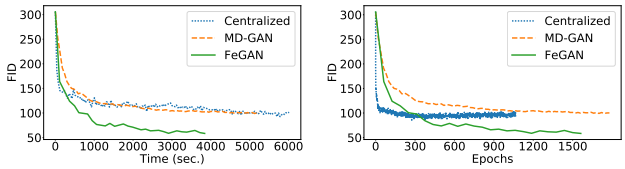
Hyper-parameters. Unless otherwise stated, we use the following hyper-parameter values in our experiments. We set E, the local number of iterations per each device, to 30, C, the fraction of devices chosen each round (by the server) for the training, to 0.025, B, the batch size, to 50, and *FID* batch size, the number of samples used to calculate the *FID*, to 10,000. We discuss the effect of changing the values of E and C in Section 6. We monitor the progress of *FID* every 1000 training iterations (not to confuse with *epochs* nor with *FL rounds*). We use the *Adam* optimizer [20] for both the generator and the discriminator with an initial learning rate of 0.0002 and values for betas 0.5 and 0.999. As instructed in their original papers, we use Mean Square Error (MSE) loss function with LSGAN [31] and Binary Cross-Entropy (BCE) with DCGAN [37] for both networks.

Baselines. We compare FeGAN against three competitors:
1. Centralized GAN. We use a centralized GAN as a baseline to be able to compare the resulting *FID* with FeGAN as well as to illustrate the throughput gain of distributing the computation. We train a GAN on a single machine with two GPUs. The hyper-parameters are set to the same value as FeGAN, including the batch size for fairness.
2. Parameter Server-based GAN. Existing approaches to distribute GANs rely on the parameter server architecture [26] to handle the distributed training [16, 52]. Note that the evaluation of such systems (in their original papers) has been conducted by emulation. Since none of these systems is open–sourced yet and rely on the same architecture (with minor nuances), we picked **MD-GAN** [16] as a representative of this class of systems. Like FeGAN, MD-GAN assumes that data on devices is never shared with any other machine. We implemented MD-GAN in our distributed framework using the same networking abstractions, models and datasets that we used in FeGAN. For the sake of fairness, we put Tensors on GPUs and handle communication using the same GPU-to-GPU abstractions. We also use the best values for the hyper-parameters as instructed by the authors in their original paper [16].
3. Federated Averaging (FL-vanilla). We compare to a strawman FL setup, where discriminators and generators remain on devices. Such devices send error gradient updates to a central server that hosts the up-to-date generator and discriminator. This constitutes the FL baseline, inspired by the *FedAvg* algorithm [32], we coin FL-vanilla.

FeGAN configurations. We report on all the variants of our FeGAN system with all combinations of *balanced sampling* and *KL weighting* techniques being used or not. We abbreviate *sampling* with s and *weighting* with w in the figures with 0 means disabled and 1 means enabled. For example, $s = 1, w = 0$ denotes FeGAN deployment that uses our *balanced sampling* scheme but not the *weighting* one. This is completely independent of the distribution of data among devices. Note that setting $s = 0, w = 0$ is equivalent to employing FL-vanilla.

5 FeGAN Convergence

In this section, we show the quality of the data generated by FeGAN, measured by *FID*, as well as the convergence behavior compared to other baselines in both *iid* data and *non-iid* data contexts.



(a) Time **(b)** Training epochs

Figure 4. Convergence of FeGAN compared to the *Centralized* approach and MD-GAN. We distributed the Fashion-MNIST dataset identically and independently (*iid*) on 80 devices for the distributed deployments. *FID* metric: the lower, the better the generated samples.

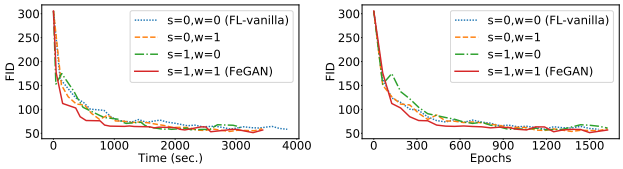
5.1 Convergence of FeGAN in *iid* Settings

Comparison with competitors. We run a head-to-head comparison with a centralized GAN and MD-GAN w.r.t. convergence performance over time and the number of training epochs. In this experiment, we distribute data identically and independently (*iid*) over devices in the distributed deployments, namely MD-GAN and FeGAN. As we wanted to focus on the benefit obtained from FeGAN architecture, namely the fact that both the generator and the discriminator are distributed, we voluntarily disable the weighting and sampling schemes in this experiment ($s = 0, w = 0$). Indeed, distribution leads to communication overhead, which FeGAN minimizes (by the co-location of the generator with the discriminator on all devices in addition to sampling a limited number of devices for training in each round).

Results displayed on Figure 4 clearly show that FeGAN significantly outperforms both competitors. In both figures, we observe that FeGAN converges to an *FID* value of around 50 where the other approaches converge to an *FID* value of around 100. Compared to the centralized deployment, FeGAN achieves a better *FID* in less time (Figure 4a). At the first glance, this result looks counter-intuitive. Yet, this is due to the fact that in FeGAN, the server aggregates updates based on more data samples (*i.e.*, collected from more devices), highlighting the advantage of distributing the training

process.[5] This also gives FeGAN a more diverse view of the dataset and ensures a better convergence than in the centralized case [49, 54]. This is consistent with the observations reported in [12, 17] while comparing a centralized approach to a distributed one. Since MD-GAN relies on a single generator at the server, it fails to benefit from the data diversity on multiple devices and hence, achieves a performance similar to the centralized approach. FeGAN achieves a much faster convergence because the server communicates only with a fraction of devices in each round as opposed to MD-GAN where it communicates with all of them; this enables FeGAN to achieve faster iterations.

The results depicted on Figure 4b show that FeGAN consistently outperforms MD-GAN. However, the centralized approach converges faster than the distributed approaches in the first few hundreds of epochs. After that, FeGAN achieves a better *FID* value. This is due to the fact that in the first few epochs, FeGAN contacts different devices per round and hence, the server aggregates updates from different data, trying to adapt both networks (the generator and the discriminator) to this data, which leads to the observed slow start. In the subsequent iterations, as the weights of both networks become more stable, FeGAN leads to better-generated data (and hence, a better *FID*).



(a) Time **(b)** Training epochs

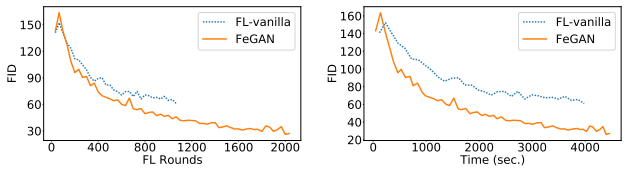
Figure 5. Convergence of a few FeGAN configurations in a distributed setup with the Fashion-MNIST dataset distributed identically and independently (*iid*) on 80 devices.

FeGAN configurations. Results previously demonstrated the relevance of FeGAN (with $s = 0, w = 0$) in an *iid* setup over existing distributed and centralized approaches. We now assess the impact of the core mechanisms of FeGAN, namely the impact of s and w in an *iid* setting.

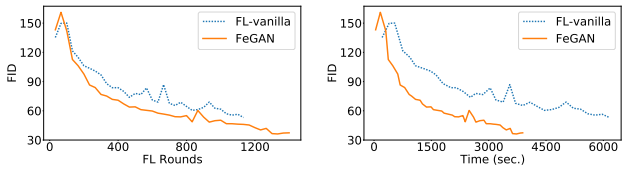
Figure 5 displays the results of different FeGAN deployments with all combinations of enabling and disabling the *balanced sampling* and the *KL weighting* schemes. We observe that in terms of convergence speed (both in terms of time and training epochs) they perform approximately the same. FeGAN (with $s = 1, w = 1$) performs slightly better and provides quickly a smaller *FID*.

Those results demonstrate that, while we aimed at sustaining *non-iid* data distributions when designing our system, FeGAN is equally good in *iid* setups. This makes FeGAN an excellent system across data distributions.

[5]Note that we could not arbitrarily increase the training batch size on the centralized setup, that is prone to memory constraints.

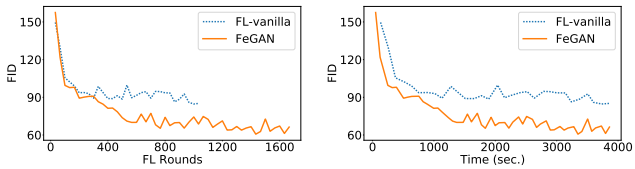


(a) Convergence with FL rounds **(b)** Convergence with time
max_class=20 - max_samples=3000

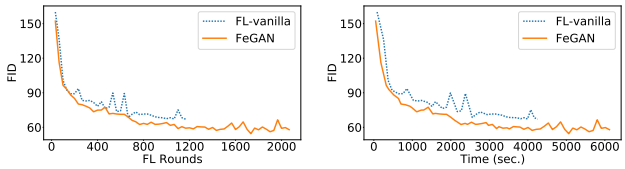


(c) Convergence with FL rounds **(d)** Convergence with time
max_class=40 - max_samples=3000

Figure 6. Convergence of FeGAN in a distributed setup with MNIST; data not distributed identically and independently (*non-iid*) on devices.



(a) Convergence with FL rounds **(b)** Convergence with time
max_class=10 - max_samples=2000

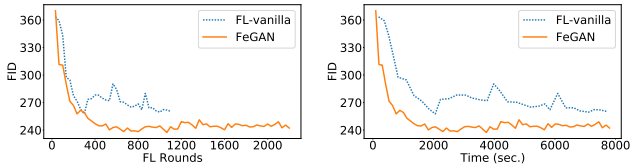


(c) Convergence with FL rounds **(d)** Convergence with time
max_class=40 - max_samples=4000

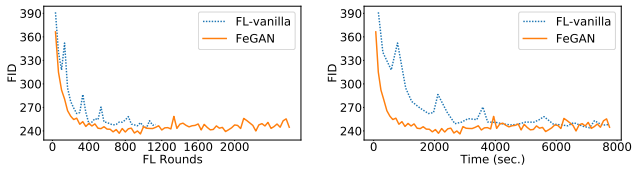
Figure 7. Convergence of FeGAN in a distributed setup with Fashion-MNIST; data not distributed identically and independently (*non-iid*) on devices.

5.2 Convergence in a *Non-iid* Context

In this section, we extensively evaluate the performance of FeGAN in a *non-iid* setup. To this end, we compare FeGAN in its full-fledged version (with $s = 1, w = 1$) to the FeGAN version where the sampling and the weighting schemes have been disabled (*i.e.*, the FL-VANILLA competitor). We run these experiments in three different datasets and different data distributions. We control the data distribution (and hence, the degree of *non-iidness* and data imbalance) using the data distribution engine described in Section 4. Figures 6, 7, and 8 show the results on the MNIST, Fashion-MNIST, and ImageNet datasets respectively, each in two settings where we vary the maximum number of classes and samples per device. The first two datasets are used to show the effectiveness



(a) Convergence with FL rounds **(b)** Convergence with time
max_class=200 - max_samples=500



(c) Convergence with FL rounds **(d)** Convergence with time
max_class=200 - max_samples=50

Figure 8. Convergence of FeGAN in a distributed setup with ImageNet; data not distributed identically and independently (*non-iid*) on devices.

of FeGAN on well-known baseline datasets, while the ImageNet experiment shows the performance of FeGAN on a large-scale dataset. For space constraints and as they exhibit similar shapes, we discuss the three figures collectively, highlighting the main performance gains of FeGAN.

Faster model updates. Our first observation is that FeGAN runs epochs *faster*, *i.e.*, leading to a faster convergence, than FL-vanilla[6]. The *balanced sampling* (as compared to random sampling in FL-vanilla) of FeGAN favors devices that were not visited in previous rounds and hence, it has a higher probability of sweeping over the whole dataset (distributed among devices) faster than FL-vanilla. We quantify this throughput gain to $1.27-1.8\times$ with MNIST, $1.57-1.7\times$ with Fashion-MNIST and $1.9-2.13\times$ with ImageNet. We discuss this observation in detail in Section 6.

***Better* FID.** Not only FeGAN achieves faster epochs in all datasets but we also observe that it converges to a better *FID* value, compared to FL-vanilla, after training for the same number of epochs. This is due to both our *balanced sampling* and *KL weighting* schemes. The fact that the server adjusts the weight of each device depending on its data distribution helps achieve a better generation of data (and hence, a lower value for *FID*), as opposed to FL-vanilla that considers devices equal regardless of the number of classes and the number of samples per class they hold. This, in turn, enables the server of FeGAN to consider faster a diverse set of data with a higher probability, contributing to learning faster the global data distribution.

Convergence gain. To demonstrate the superiority of FeGAN over FL-vanilla, we compare the time required to achieve a given *FID* value (usually the best value achieved by FL-vanilla) and call it *convergence gain*. Results show that the convergence gain of FeGAN over FL-vanilla is $1.27-1.57\times$, $1.71-3\times$ and $2.75-3.67\times$, when training MNIST, Fashion-MNIST, and ImageNet respectively. Beyond the ultimate *FID* value, a system can achieve eventually, the convergence gain is an important end–metric for ML practitioners. Such a metric combines the system's performance aspect (*e.g.*, throughput or latency) and the ML algorithm performance aspect (*i.e.*, convergence over training epochs).

The less uniform, the more effective FeGAN is. When comparing the first and second rows of Figures 6 and 7, we observe that the performance gap between FeGAN and FL-vanilla is larger for lower values of *max_class*. In other words, the effectiveness of FeGAN is even clearer when each device owns only a few classes from the whole dataset. This demonstrates the ability of FeGAN to account for a higher degree of data imbalance over devices; we believe the latter is a realistic scenario as we do not expect that all classes would be represented on distributed devices. Figures 8c and 8d show a real–world scenario, where each device has at most 50 samples from any of the 200 classes of ImageNet. Results from both figures show that FeGAN converges faster than FL-vanilla in terms of both the number of rounds and time.

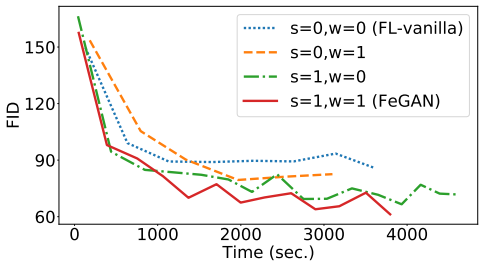


Figure 9. Convergence of FeGAN with *non-iid* Fashion-MNIST data distributed on 80 devices, showing all cases of enabling and disabling weighting and sampling schemes.

5.3 Relative Importance of Sampling & Weighting

We evaluate the relative impact of activating each mechanism in all datasets. The results for Fashion-MNIST are depicted in Figure 9. We observe that while using only one mechanism is enough for FeGAN to achieve a lower *FID* than FL-vanilla (where using *sampling* alone is slightly better than using *weighting* alone), using both mechanisms achieves the lowest *FID*.

Preventing mode collapse. Figure 10 shows the KL-divergence of the data distribution seen by the server (over FL rounds) compared to the real data distribution. We compare *balanced sampling* (used by FeGAN) to random sampling (used by FL with *FedAvg* [21]) and the *iid* case (when data is distributed identically on all devices). On the one hand, the figure shows

[6]Note that we run both systems for the same amount of time per experiment; that is why the lines in all sub-figures b are balanced, unlike sub-figures a.

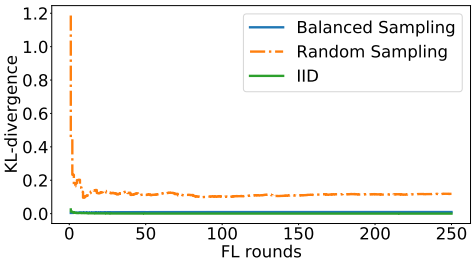
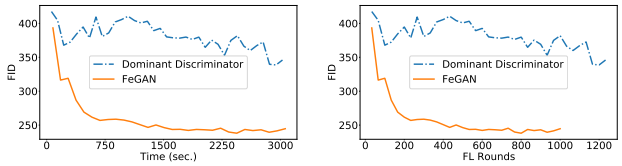


Figure 10. KL–divergence of the real distribution from the distribution seen by the centralized server. The smaller the KL-divergence, the lower the probability of mode collapse.

that using random sampling with *non–iid* data will let the server see a diverging distribution compared to the real distribution of data. On the other hand, balanced sampling of *non–iid* data allows the server to see almost–exact distribution of data (compared to the optimal case, *i.e.*, with *iid* data). As demonstrated by previous work [28], the smaller the KL-divergence, the less the probability of experiencing mode collapse; FeGAN then shows resilience in that respect.



(a) Time **(b)** Training epochs

Figure 11. The co-location of both networks on all nodes avoids yielding a powerful discriminator (as in MD-GAN), preventing the problem of vanishing gradients to happen.

Preventing vanishing gradients. FeGAN co–locates the discriminator with the generator on all devices so that the training of both networks can happen simultaneously in all training iterations, allowing both networks to have the same power in confronting each other. To demonstrate the efficiency of this approach, we run an experiment in which the discriminator is trained in all iterations while the generator is trained only once each E iterations. Such a design reflects that of having one generator in the central server and multiple discriminators on the devices, which was proposed before, *e.g.*, in [5, 44]. Figure 11 shows that the latter design creates a dominant discriminator that can always defeat the generator, hindering the latter from generating output data that looks real. Yet, the co–location of both networks on all devices (applied by FeGAN) allows the generator to be powerful enough to generate such data, as demonstrated by the lower *FID* values. FeGAN thus clearly avoids that problem.

6 FeGAN System Performance

This section reports the performance of FeGAN with regards to baselines. The results being similar across datasets, we only present the results obtained with the Fashion-MNIST dataset unless stated otherwise.

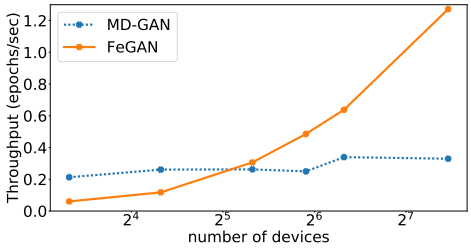


Figure 12. Scalability with the number of devices.

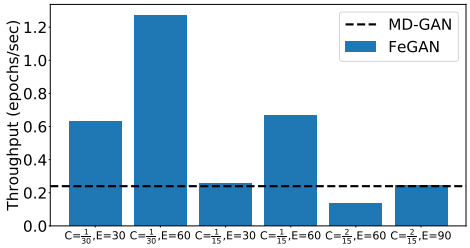


Figure 13. The impact of the values of E and C (60 devices).

Throughput. We measure the system's throughput as the number of epochs achieved per second. Figure 12 plots the scaling of the throughput of both MD-GAN and FeGAN with the number of devices. We observe a stable throughput for MD-GAN, which is almost non–sensitive to an increasing number of devices. This is because the single generator on the server has a bounded processing capacity regardless of the number of devices interacting with it. Results show that FeGAN however exhibits a close to perfect scaling. The observed throughput increases linearly with the number of devices, fully leveraging the potential of distributing the training. With 176 devices, FeGAN exhibits a 5× throughput increase over MD-GAN. Note that such scalability also depends on the chosen values for E and C. For instance, we do not expect FeGAN to scale with $C = 1, E = 1$ as it will incur the same amount of communication as with MD-GAN in this case. Hence, the values of C and E largely control whether the bottleneck is the communication or the computation.

Figure 13 presents the results obtained when varying the values of parameters E, the local number of iterations performed on each device in one FL round, and C, the fraction of devices chosen by the server in each round, for FeGAN and MD-GAN (as a baseline) in a 60–device configuration.

We observe that the performances of FeGAN are significantly better in several configurations and in any case at least as good as the ones of MD-GAN, but in the $C = 2/15$, $E = 60$ configuration. We believe this latter issue is a pure implementation issue, which we detail in the following. On the one hand, since the server in MD-GAN only averages the devices' updates, we use the *reduce* abstraction in PyTorch which has an efficient GPU-to-GPU NCCL [35] implementation. On the other hand, we use the *gather* abstraction with

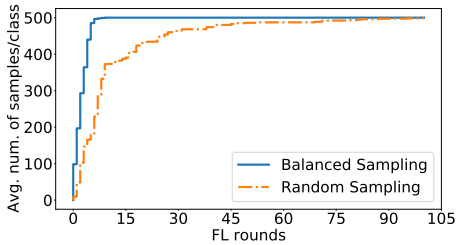


Figure 14. Average number of samples per class seen by the server, through devices' updates, as a function of FL rounds. Here, the maximum number of samples at each device is 50. (ImageNet, 80 devices).

FeGAN to enable *KL weighting*, *i.e.*, the server first gathers the updates and then applies a weighted sum on them. Such a communication abstraction is not supported on GPUs (to–date in PyTorch) and hence, the models are copied to the CPU memory first, leading to slower end-to-end communication.

From the results, we observe that lower values of C give a significant advantage to FeGAN, with one configuration ($C = 1/30$, $E = 60$) even performing close to 4 times faster than the MD-GAN configuration. This is explained by the communication overhead between the server and the devices: the lower the value of C, the lower the number of devices chosen per round and hence, the lower the communication overhead. Since the communication overhead is the main bottleneck in distributed machine learning applications in general [18, 47], reducing the communication drastically affects the system throughput, without impacting the convergence as demonstrated in the previous section.

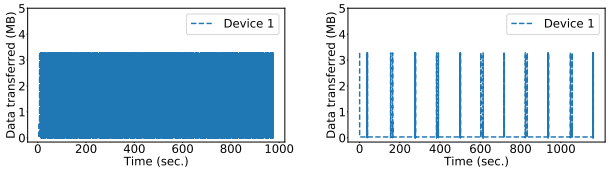
Figure 13 also shows a positive correlation between the value of E and the throughput. The higher the value of E, the more local iterations per device per round, the lower the communication between the server and the devices per unit time and hence, the higher the system's throughput. Note that while studying the impact of the values of C and E gives us some indication, choosing the best values for C and E remains data and model–dependent.

Network bandwidth. Figure 14 shows the impact of FeGAN's *balanced sampling* on the data diversity (number of samples per class) observed by the server as a function of the number of FL rounds. We observe that *balanced sampling* significantly impacts the training time: it takes FeGAN 10 rounds to come across 500 samples against 10x this figure when random sampling is applied.

The effectiveness of *balanced sampling* directly translates into network bandwidth gain. Effectively, random sampling has a higher probability to visit the same data batch twice consuming unnecessary bandwidth. Table 1 summarizes the gains in terms of time and communication. This confirms the results displayed in Figure 14. Based on these numbers, *random sampling* requires around 1.5× more bandwidth, compared to our *balanced sampling*, per epoch.

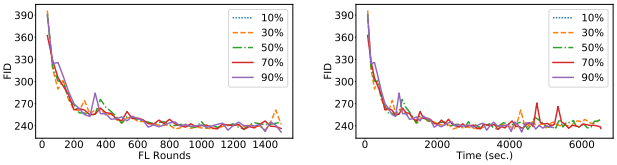
Setup	5 - 3000	10 - 2000	20 - 3000	40 - 4000
FL-vanilla	3596.9s	3733.3s	4825.7s	3815.5s
FeGAN	2441.4s	2379.9s	3031.7s	2560.2s
Gain	1.47×	1.57×	1.59×	1.49×

Table 1. Time to finish 200 epochs by FeGAN and FL-vanilla. With the former, we use *balanced sampling* while with the latter, we use random sampling. The given two numbers in the first row represent *max_class* and *max_samples* respectively. The gain in time to finish one epoch can be also viewed as a gain in consumed bandwidth over random sampling for not looking at diverse data batches.

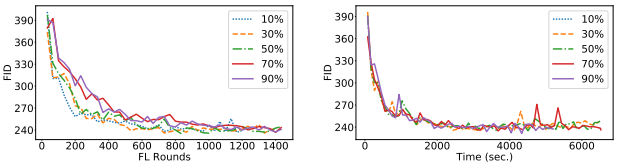


(a) MD-GAN **(b)** FeGAN

Figure 15. Bandwidth consumption at one random device for both MD-GAN and FeGAN. We observe the same distribution (as plotted) at different sampled devices. Using FeGAN, each device is picked (by the server), on average, less than 10 times in a 200-second interval.



(a) Convergence with FL rounds **(b)** Convergence with time
Limited memory



(c) Convergence with FL rounds **(d)** Convergence with time
Limited processing power

Figure 16. Convergence of FeGAN in the presence of weak devices in terms of memory or processing power. FeGAN is adaptive to heterogeneous devices. (ImageNet, 60 devices)

Moreover, as Figure 15 shows, devices in FeGAN consume less bandwidth than with MD-GAN as each device is selected only once every few FL rounds as opposed to MD-GAN in which all devices train the model each round. In this figure, each device is sampled less than 10 times every 200 seconds.

Heterogeneous devices. Figure 16 shows an experiment with a set of devices, some of them with limited memory or processing power (we denote those as *weak* devices). Based on

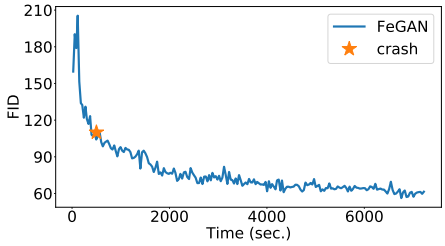


Figure 17. FeGAN tolerates server crash.

FeGAN's design, such devices are assigned smaller batch sizes (to respect the memory constraints) or less number of local iterations (to respect the limited processing power). In this experiment, weak devices are assigned up to half of the batch size and half of the local number of iterations. The figure also shows the effect of having different ratios of weak devices compared to the total number of devices (in the range of 10–90%). All figures show that the presence of weak devices almost does not affect FeGAN's convergence. This shows the efficiency and the practicality of the adaptive methods included in the design of FeGAN.

Tolerating server crashes. As the central server failure is more critical than the devices' failures, we focus here on the former case. Figure 17 shows an experiment in which we crash the central server after 500 seconds from the beginning of the experiment. In such a case, a backup server loads the latest stored checkpoint, announces to the devices that it is the new primary, and continues the training normally from that checkpoint. The figure shows that the server failure/crash does not cause troubles (*e.g.*, big stalls) to the training process, which continues normally after the crash.

7 Related Work

Few proposals suggested distributing GANs computation on multiple machines. MD-GAN [16] aims at reducing the computation on workers by relying on a single generator, hosted on the central server of the parameter server model. In this setup, every worker hosts a discriminator. Such an architecture then breaks the usual generator–discriminator couple, by setting up a 1-to-n game. The discriminators perform local learning steps on their datasets (that are never shared with other machines) and compute the error feedback on the generated samples they are given by the server. Discriminators are periodically swapped between the workers in a peer-to-peer fashion, in order to avoid overfitting the local datasets. MD-GAN is built with scalability to few tens of workers in mind for targeting the within-datacenter learning scheme; it also only targets the *iid* distribution of data.

A recent approach by Yonetani et al. [52] builds on the architecture of MD-GAN, with also one single generator. They aim at tackling some distribution skew on worker nodes by proposing two independent unsupervised learning approaches: F2U and F2A. F2U is designed to fool the most *forgiving* discriminator for a given sample (as they judge the sample real and close to the data they own). F2A adaptively aggregates the feedback of discriminators by emphasizing those from the more forgiving ones. Augenstein et al. [5] proposed a similar federated GAN algorithm but also with one generator at the server and multiple discriminators at the devices. Such an algorithm focuses on the privacy of users' data on devices with the goal of synthesizing data from similar distribution of real data for diagnosis. The scalability of a single generator is here also at stake; we propose instead to have multiple generator-discriminator couples.

Hardy et al. [15] proposed a fully decentralized, peer-to-peer architecture for distributing GANs. The authors experimented with GANs being gossiped and averaged between independent devices. While acceptable on small datasets, the performances are observed to be inferior to the baselines we use to evaluate FeGAN with bigger datasets.

8 Concluding Remarks

We evaluated FeGAN extensively, and here is our conclusion: *(i)* its careful design allows for scaling the training of GANs. While MD-GAN provides better throughput at a small scale (up to 32 devices), FeGAN then largely prevails, achieving a linear improvement in throughput with the number of devices (experimented with up to 176 devices) with even a lower bandwidth consumption. This scaling does not come at the cost of learning quality; at the opposite, we show lower FID values in both *iid* data and *non–iid* data contexts. *(ii)* We have shown that the proposed components of the FeGAN systems can circumvent GAN specific issues that a vanilla deployment of an FL-based scheme can encounter. In that respect, both the *balanced sampling* and the *KL weighting* schemes are instrumental in the effectiveness of FeGAN.

Such components essentially utilize the skew information released by the devices at the beginning of the learning, with regards to their local data. The requirement for extra information might constitute a privacy concern. Yet, recent work has shown how to employ differential privacy with training vanilla federated GANs [50]. Exploring approaches that tolerate data imbalance while ensuring full privacy of devices' data is an avenue for future research.

Acknowledgement

We thank Georgios Damaskinos for his useful comments.

References

[1] Fashion-mnist dataset. https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/.

[2] Grid5000. https://www.grid5000.fr/.

[3] Mnist dataset. http://yann.lecun.com/exdb/mnist/.

[4] Arjovsky, M., and Bottou, L. Towards principled methods for training generative adversarial networks, 2017.

[5] Augenstein, S., McMahan, H. B., Ramage, D., Ramaswamy, S., Kairouz, P., Chen, M., Mathews, R., et al. Generative models

for effective ml on private, decentralized datasets. *arXiv preprint arXiv:1911.06679* (2019).

[6] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482* (2016).

[7] Borji, A. Pros and cons of gan evaluation measures. *Computer Vision and Image Understanding 179* (2019), 41–65.

[8] Brock, A., Donahue, J., and Simonyan, K. Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations* (2019).

[9] Che, T., Li, Y., Jacob, A. P., Bengio, Y., and Li, W. Mode regularized generative adversarial networks. In *ICLR* (2017).

[10] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[11] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09* (2009).

[12] Durugkar, I., Gemp, I., and Mahadevan, S. Generative multi-adversarial networks. *arXiv preprint arXiv:1611.01673* (2016).

[13] Faber, V. Clustering and the continuous k-means algorithm. *Los Alamos Science 22*, 138144.21 (1994), 67.

[14] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680.

[15] Hardy, C., Le Merrer, E., and Sericola, B. Gossiping gans. In *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning: DIDL* (2018), vol. 22.

[16] Hardy, C., "Le Merrer", E., and Sericola, B. Md-gan: Multi-discriminator generative adversarial networks for distributed datasets. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), IEEE, pp. 866–877.

[17] Hoang, Q., Nguyen, T. D., Le, T., and Phung, D. Multi-generator generative adversarial nets. *arXiv preprint arXiv:1708.02556* (2017).

[18] Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI* (2017), pp. 629–647.

[19] Incubator, F. Gloo. https://github.com/facebookincubator/gloo.

[20] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[21] Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[22] Kullback, S. *Information theory and statistics*. Courier Corporation, 1997.

[23] Kullback, S., and Leibler, R. A. On information and sufficiency. *The annals of mathematical statistics 22*, 1 (1951), 79–86.

[24] Kurach, K., Lučić, M., Zhai, X., Michalski, M., and Gelly, S., Eds. *A Large-Scale Study on Regularization and Normalization in GANs* (2019).

[25] Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., and Shi, W. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *CVPR* (2017).

[26] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014).

[27] Li, Y., and Lyu, S. Exposing deepfake videos by detecting face warping artifacts. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2019).

[28] Lim, W. Y. B., Luong, N. C., Hoang, D. T., Jiao, Y., Liang, Y.-C., Yang, Q., Niyato, D., and Miao, C. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials* (2020).

[29] Liu, Z., Luo, P., Wang, X., and Tang, X. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)* (December 2015).

[30] Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. Are gans created equal? a large-scale study. In *Advances in neural information processing systems* (2018), pp. 700–709.

[31] Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., and Paul Smolley, S. Least squares generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 2794–2802.

[32] McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In *AISTATS* (2017).

[33] Neyshabur, B., Bhojanapalli, S., and Chakrabarti, A. Stabilizing gan training with multiple random projections. *arXiv preprint arXiv:1705.07831* (2017).

[34] Nishio, T., and Yonetani, R. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)* (2019), IEEE, pp. 1–7.

[35] NVIDIA. Nccl. https://developer.nvidia.com/nccl, 2017.

[36] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* (2019), pp. 8024–8035.

[37] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[38] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. Generative Adversarial Text to Image Synthesis. *ArXiv e-prints* (May 2016).

[39] Richardson, S., and Green, P. J. On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: series B (statistical methodology) 59*, 4 (1997), 731–792.

[40] 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 (2015), 211–252.

[41] Sattler, F., Wiedemann, S., Müller, K.-R., and Samek, W. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems* (2019).

[42] Sutskever, I., Hinton, G. E., and Krizhevsky, A. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (2012), 1097–1105.

[43] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 2818–2826.

[44] Triastcyn, A., and Faltings, B. Federated generative privacy. *arXiv preprint arXiv:1910.08385* (2019).

[45] Vondrick, C., Pirsiavash, H., and Torralba, A. Generating Videos with Scene Dynamics. *ArXiv e-prints* (Sept. 2016).

[46] Williams, C. K., and Rasmussen, C. E. Gaussian processes for regression. In *Advances in neural information processing systems* (1996), pp. 514–520.

[47] Xie, C., Koyejo, O., and Gupta, I. Slsgd: Secure and efficient distributed on-device machine learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2019), Springer, pp. 213–228.

[48] Xie, C., Koyejo, S., and Gupta, I. Practical distributed learning: Secure machine learning with communication-efficient local updates. *arXiv preprint arXiv:1903.06996* (2019).

[49] Xie, P. *Diversity-promoting and Large-scale Machine Learning for Healthcare*. PhD thesis, University of Pittsburgh Medical Center, 2018.

[50] Xin, B., Yang, W., Geng, Y., Chen, S., Wang, S., and Huang, L. Private fl-gan: Differential privacy synthetic data generation based on federated learning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2020), IEEE, pp. 2927–2931.

[51] Yang, Q., Liu, Y., Chen, T., and Tong, Y. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol. 10*, 2 (Jan. 2019), 12:1–12:19.

[52] Yonetani, R., Takahashi, T., Hashimoto, A., and Ushiku, Y. Decentralized learning of generative adversarial networks from multi-client non-iid data. *CoRR abs/1905.09684* (2019).

[53] Zhao, Y., Li, M., Lai, L., Suda, N., Civin, D., and Chandra, V. Federated learning with non-iid data. *CoRR abs/1806.00582* (2018).

[54] Zhou, T., Wang, S., and Bilmes, J. A. Diverse ensemble evolution: Curriculum data-model marriage. In *Advances in Neural Information Processing Systems* (2018), pp. 5905–5916.