Performance Evaluation on Generative Adversarial Networks in a Distributed Setting

Distributed Deep Learning Systems

Aria David Darmanger[1] and Owen Gombas[1]

[1] *University of Bern*
University of Neuchâtel
University of Fribourg
Swiss Joint Master of Science in Computer Science

29.05.2024

Table of contents

1. Abstract

With the rise of Deep Learning, Generative Adversarial Networks (GANs) have become a popular method for generating synthetic data. However, training GANs is challenging, particularly when dealing with large and privacy-sensitive datasets. This work provides an extensive analysis of training GANs in a distributed setting. It presents an open-source implementation that operates over networked environments while enhancing privacy by keeping data local.

This research was conducted for the "Distributed Deep Learning" and "Software Engineering" courses from the Swiss Joint Master of Science in Computer Science program.

The final product of our work is available on both of our GitHub profiles https://github.com/darmangerd/distributed-gan or https://github.com/owengombas/distributed-gan since we both equally contributed to the project.

2. Introduction

2.1 Background

Generative Adversarial Networks (GANs) have revolutionized the field of Machine Learning by enabling the creation of highly realistic images. A great example of the results GANs can achieve is showcased on the website this-person-does-not-exist.com. These networks learn the distribution of a given dataset to generate new similar data points. The standard GAN framework involves a generator and a discriminator in an adversarial setup. The generator's goal is to produce data which are similar to real data, while the discriminator aims to differentiate between the generated and real data.

2.2 Challenges in training GANs

Despite their success, GANs face several challenges in the training process, particularly with large datasets. These challenges include the instability during training, where the generator and discriminator can become unbalanced, leading to poor performance. Training GANs is also computationally expensive because of the complex models and large amount of data involved. When data privacy is a concern, such as in medical or financial applications, the situation becomes even more complex as sensitive data cannot be easily shared or centralized, complicating the distributed training process.

2.3 Distributed GAN training

To address these challenges, distributed training methods for GANs have been developed. These methods allow GANs to be trained across multiple computational nodes, using parallel resources to improve training efficiency. One promising approach is training with multiple discriminators, where each discriminator accesses only its local data and contributes to training a global generator. This setup not only scales GAN training but also enhance data privacy by keeping sensitive data decentralized.

2.4 Our contribution

This work builds on the existing multi-discriminator GAN frameworks, specifically extending the approach described in the original MD-GAN paper [Hardy et al., 2019].

Our main contributions are:

1. **Open-Source implementation**: We provide an open-source implementation of the MD-GAN framework, which was not originally available. This implementation works across real networked environments, demonstrating the feasibility and scalability of distributed GANs in practical applications.
2. **Performance evaluation results**: We present a detailed analysis of GAN performance in a real distributed setting, taking into account networking constraints and bottlenecks that a production-ready product might encounter. We also describe a way to improve the process and reduce the idle time in every node, opening the way for speeding up the distributed training.
3. **Tools for further analysis**: Our implementation includes tools for deeper analysis, such as evaluating on non-IID datasets, offering a flexible framework for new datasets, and an automatic data collection system with detailed plot generation.

3. Scientific questions and hypothesis

In distributed Deep Learning systems, especially those involving GANs, several important scientific questions emerge. These questions and hypothesis help us understand the benefits and challenges of implementing GANs across distributed settings. The following questions and hypotheses are fundamental to our research and will assist in evaluating the capability and effectiveness of distributed GAN architecture. The empirical testing of these hypotheses through experiments will contribute to a deeper understanding of the trade-offs and benefits of distributed GAN training.

A) Does training GANs in a distributed setting helps converging to an optimal solution in fewer epochs than a standalone version?

In distributed GAN settings, multiple computational nodes collaborate to train the model. This arrangement potentially allows the model to simultaneously learn more diverse features from different data subsets. We aim to determine if this setup leads to faster convergence compared to traditional, centralized training methods; Where the model learns from a single data source. However, it is also possible that the lack of coordination among nodes could lead to conflicting updates, slowing down the convergence process.

Therefore we define our first hypothesis this way:
H1: Training a GAN in a distributed setting with multiple discriminators helps converging to an optimal solution in fewer epochs than a standalone version.

B) Does training GANs in a distributed setting slow down the time to complete an epoch compared to a standalone setting?

It might seem intuitive to assume that training a GAN with multiple workers would terminate an epoch faster than in a standalone setting, similar to how multiple cores can speed up processing. For example, one might expect that having two workers would be twice as fast as a standalone setting. However, this assumption requires a more detailed examination.

In distributed GAN training, the same operation is performed multiple times and the results are averaged, which does not necessarily reduce the overall processing time per epoch. Multiple workers could actually make the completion of an epoch slower in a distributed setting.

Several factors can affect the overall training time, including network latency, synchronization overhead, and the efficiency of data distribution and aggregation. Communication overhead between the server and workers can significantly impact training speed, and synchronization of gradients across all workers can become a bottleneck, especially as the number of workers increases. Moreover, the efficiency of distributed training also depends on network bandwidth.

Therefore, we define our second hypothesis this way:
H2: Training a GAN in a distributed setting with multiple discriminators extends the time it takes for an epoch to complete compared to a standalone setting.

4. Related works and fundamentals

4.1 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a type of generative model that has become popular recently. Introduced by Goodfellow et al. in 2014 [Goodfellow et al., 2014], GANs are used in many areas like image generation, style transfer, and data augmentation. The main idea behind GANs is to train two models, a generator and a discriminator, in a competition. The generator learns to create data that looks real, while the discriminator learns to tell apart the generator's fake data from real data. Both networks are trained at the same time: the generator tries to trick the discriminator, and the discriminator tries to correctly identify the real data. This adversarial training helps the generator learn the real data distribution and produce realistic samples.

Figure 4.1 shows the typical training process of a GAN. Step (1) is updating the discriminator's weights, which happens just before updating the generator, shown in step (2). To update the discriminator, we pass a batch of real images and a batch of fake images (created by the generator) through the discriminator to get its predictions. The discriminator's weights are then updated to reduce errors in prediction, such as misclassifying a real image as fake.

After updating the discriminator's weights, we focus on the generator. We pass a batch of fake images through the discriminator again to get its predictions. We then calculate a Binary Cross Entropy (BCE) loss between the discriminator's output for the fake images and the labels for real images. This loss will be high if the generator fails to trick the discriminator and low if it succeeds. Since the optimizer updates the weights to minimize this loss, the generator gets better at creating realistic fake images over time.

The important thing to note in this figure is that the generator never directly accesses the real data, it only uses the output from the discriminator to update its weights. This is why distributing the discriminators allows the real images to be kept locally, avoiding the need to share them with other nodes.

To grasp an even better understanding of this concept, we advice to reader to visit GAN Lab[1] [Kahng et al., 2019] which provides a great visualization helping to understand the process of training a GAN.

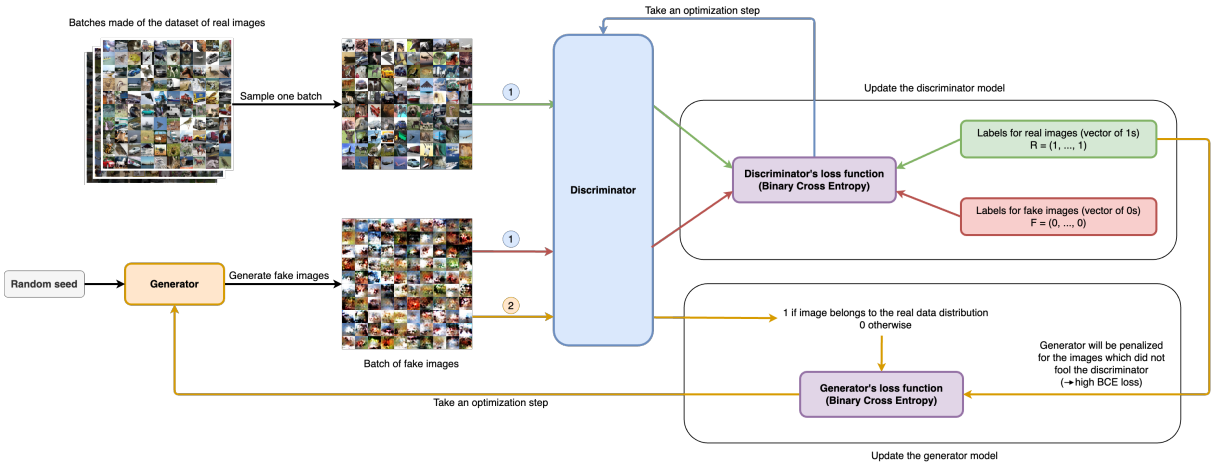

Figure 4.1: (1) Update the discriminator's model, (2) Update the generator's model

[1] https://poloclub.github.io/ganlab/

4.2 Distributed Deep Learning

Distributed Deep Learning is the process of training Deep Learning models across multiple computational nodes that are connected and can communicate. These nodes can be connected in different ways, like through a traditional Ethernet connection. Also, a single machine using multiple GPUs can be considered a distributed setting. The setup in both cases is quite similar; the main difference is the communication protocol used. This protocol is often managed by the most popular libraries for Distributed Deep Learning.

By spreading the work across multiple devices, we can train Deep Learning models more efficiently and on a larger scale, making the training process scalable. It also allows for collaborative training of models across multiple institutions with sensitive data, enabling them to train a single model together without sharing their data. This approach is especially useful for training large models on big datasets, which a single machine or a small cluster might not be able to handle due to limited memory or processing power. However, there are challenges, such as communication delays, synchronization problems, and workload balancing. With the right setup and optimizations, distributed deep learning can greatly enhance privacy, scalability, and efficiency.

4.3 Multi-Discriminator GANs

Multi-Discriminator GANs (MD-GANs) are a type of GAN that uses multiple discriminator models to effectively scale the training process. While the typical GAN architecture has a single discriminator that learns to tell apart the generator's fake data from real data, MD-GANs use multiple discriminators. These discriminators can be used either with multiple generators [Rasouli et al., 2020] or with a single one [Hardy et al., 2019], but in both cases, each discriminator learns from a different part of the real data we are training on. Training multiple discriminators at the same time helps distribute the computational load across devices and keeps privacy, as each discriminator only accesses its local data. This setup avoids centralizing sensitive data and allows the training of GANs on large, distributed datasets, improving both privacy and scalability.

5. Methodology

5.1 Distributed architectures

To address our scientific questions, we used the architecture and algorithm proposed by the MD-GAN paper [Hardy et al., 2019] which used a parameter server. Other architectures using Federated Learning, such as the one described in FedGan [Rasouli et al., 2020], involve each worker holding both a generator and a discriminator, training them during the learning phase, and periodically synchronizing to train a global generator. The advantage of the MD-GAN architecture is that it uses less computational power to train a GAN network while maintaining a relatively small memory footprint and requiring fewer computational resources per worker. This allows the GANs to scale better across distributed nodes. In contrast, FedGAN requires each worker to hold and train an additional generator, increasing the resource requirements. We evaluated that the most relevant existing solution to address our scientific question is therefore the MD-GAN.

Another reason why we chose MD-GAN over FedGAN is that, as we will observe in our results, the most important aspect in our distributed setting is the network. We believe that Federated Learning is less appropriate for training a GAN because it involves larger communication sizes. In Federated Learning, the entire model must be sent over the network, not just the gradients of the current batch of data. This results in significantly more data being transmitted, which can slow down the training process.

MD-GAN's proposed architecture consists of a single server holding a generator model and N workers, each with their own discriminator. At the beginning of each epoch, the server sends generated images, based on the current generator, to each worker. Each worker then uses a batch of their own real images and the received data to perform L local epochs. The model is evaluated on a portion of the images sent by the server at the beginning of the epoch, generating gradients (feedback) to send back to the server. Periodically, the discriminators are swapped among workers to prevent overfitting to local data without sharing any data with other workers or the server.

Another reason why we chose MD-GAN over FedGAN is that (as we will observe in our results) the most important aspect in speeding up the training is the network, therefore we believe that Federated Learning is less appriorate for training a GAN since it involves bigger communication size because we have to send the whole model over the network and not only gradients of a current batch of data.

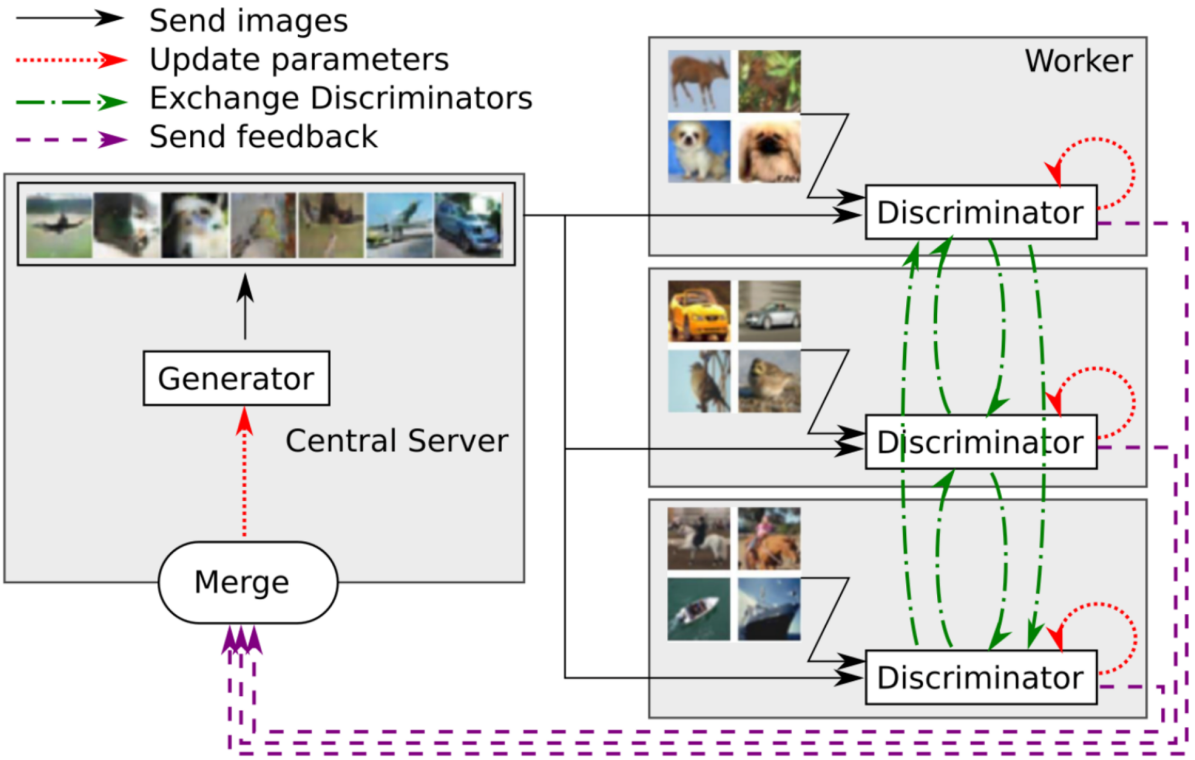

Figure 5.1: The Multi-Discriminator GANs architecture proposed by [Hardy et al., 2019]

5.2 The MD-GAN algorithm

At the start of an epoch, the server sends k generated batches. The value of k was determined to be optimal when $k = \lfloor \log_2(N) \rfloor$. However, this presents a problem when $N \leq 3$, as this value would be equal to 1. Having only one batch to choose from would force the workers to train and evaluate on the same batch of data, which would not guarantee optimal feedback.

To counter this problem, we slightly modified the calculation of k to enforce a minimum value of 2, as follow $k = \max(\lfloor \log_2(N) \rfloor)$. This ensures that in all epochs, every worker has a different training batch from the one they evaluate on.

More formally, the server will generate k batches $K = \{X^{(0)}, \ldots, X^{(k)}\}$ at the beginning of every epoch, and every worker n will receive the batches $X^{(g)} := X^{(n \mod k)}$ and $X^{(d)} := X^{((n+1) \mod k)}$.

Once the client received $X^{(g)}$ and $X^{(d)}$ it will perform L learning step on $X^{(d)}$ and batch of real images they own locally $X^{(r)}$. In our experiments L is always set to 1, simulating the standard way of training a GAN in a standalone setting. After the learning iterations finished it will evaluate the loss function on $X^{(g)}$, the gradients (so called feedbacks) of that evaluation will be send to the server for aggregation, since it corresponds to the error made on every data point in $X^{(g)}$ the size of the feedbacks is the size as the batch size b.

As soon as the server received all the gradients which corresponds to the error made by the generator on every $X^{(g)}$. The server will aggregate all the gradients as describe in [Hardy et al., 2019] and perform an optimization step. To perform this aggregation we use `torch.autograd.grad`[1] to calculate the partial derivatives of every element of $X^{(g)}$ with respect to all the weights of the current version of the generator (with the feedbacks as the vector in the vector-Jacobian product).

From this algorithm we can infer the number of messages sent and received per epoch for each

[1] https://pytorch.org/docs/stable/generated/torch.autograd.grad.html#torch-autograd-grad

type of node. Server will send two batches of image, both of size b, to every worker $\rightarrow N$ messages. Therefore, each workers receive 1 message from the server, after training they all send back their feedbacks of size b in one single message, summing up back to N message for the server to receive. This process is repeated at every epoch.

5.3 Client swapping

The most challenging part is the client swapping, we choose a different strategy than the one proposed by MD-GAN [Hardy et al., 2019]. Although their method work, we estimated that it wasn't optimal for many reason:

1. Workers choosing other workers to swap with can cause conflict, for instance worker 1 could decide to swap with worker 2 which decide to swap with worker 1, creating a loop. This would induces to send two times the size of the discriminator for no benefits because worker 1 would end up with its initial weights as well as for worker 2.
2. Workers might not be considered for swapping which doesn't help prevent over-fitting.
3. Detecting these conflicts in a interaction graph would not be optimal because, because there is no central entity that could detect them.
4. $N-1$ separate threads have to be created on every worker to listen whenever one of the other workers wants to swap with the current worker. These threads holds a infinite loop waiting for an answer for another worker, which might never arrive. This leads to unclosed TCP connections, which cause instability.
5. MD-GAN's swap frequency is variable and different on every worker because it depends on local variables such as the number of local epochs, the size of the real images dataset (which can differ from worker to worker) and the batch size. This enforces us to implement the different threads mentioned in 4., since the every workers should always be ready to receive the discriminator weights from another worker.
6. Using separate threads would enforce us to use the `tag` argument in `dist.send` and `dist.recv` which isn't supported on the NCCL backend[1], therefore we are forced to use the GLOO backend and transmit all the tensors from the GPU to the CPU whenever we want to communicate with another node, inducing more delay.

For all these reasons we introduced a novel swapping strategy that allows for using the NCCL backend. Instead of letting every worker decide with which other worker they should swap with, the server will generate pairs non-overlapping pairs (solve the 1st and 2nd and 3rd problem) of workers and it sends to every worker the rank of the other worker they will swap with. By non-overlapping pair we mean that a worker rank will never appear in more than one pair, for instance $\{(1,2),(3,4)\}$ aren't overlapping but $\{(1,2),(2,4)\}$ are, this will naturally prevent the 2nd problem from occurring. The swapping operation will be triggered at the same epoch for every node at a frequency defined by the `swap_interval` parameter, solving the 4th, 5th and 6th problems. The only constraint this introduce is to have a even number of worker (`world_size` must be odd because it takes into account the server) so every worker can find a partner to swap with.

A message count analysis can be done very easily from that method:

1. Every time a swap is triggered we introduce N messages to send containing one 32bits integer (4 bytes)
   Message count: N, bytes sent over the network: $4N$
2. Every worker will send the model to another worker and receive back another model, considering the size of the model being $|W|$. Message count: N, bytes sent over the

[1] https://pytorch.org/docs/stable/distributed.html#torch.distributed.isend

network: $|W|N$

Each swap event sends $2N$ messages will transit over the network with a total size in byte of $N(4+W)$. In the MD-GAN implementation whenever all the workers have been swapped, N messages count occurred because every worker sent their weights directly to another worker, without any other communication, with a total size of $N|W|$ bytes. Our method has a bigger network overhead in terms of swapping but solves numerous problems and allow for not transmitting tensor from the CPU to the GPU and vis-versa. Additionally the cost of this operation is amortized by the fact that it occurs at relatively big intervals, therefore we believe that this is not a problematic trade-off. For simplicity and to transcript our results in a more transparent way we chose to set the swapping frequency to a constant value of 5,000 epochs.

The pseudo-code of the whole training phase is given in the appendix C.

	Sent	**Received**
Server sends generated data (Server to Workers)	$2bN$	$2b$
Worker n send feedback (Worker to Server)	b	bN
Server send swap instructions (Server to Workers)	N	1
Worker send discriminator (Worker to Worker)	$\|W\|$	$\|W\|$

Table 5.1: All the type of communication occurring in our training procedure, the size is the number of individual floating point number sent/received within PyTorch tensors (float64).

5.4 Evaluation of the results

Evaluating generative models is challenging. The effectiveness of these models depends on the quality of data they produce, which ideally should be judged by humans. To simulate this human evaluation, researchers have developed automated methods. The Inception Score (IS) [Salimans et al., 2016] and the Fréchet Inception Distance (FID) [Heusel et al., 2018] are two of these methods. We chose to use them as our primary metrics due to their proven effectiveness in evaluating generative models. Both metrics are commonly use to asses the performances of GANs, enabling us for direct comparison of our results with other GANs' benchmarks. This alignment ensures that our evaluations are relevant and scientifically rigorous, improving the validity of our findings within the broader research context.

5.4.1 Inception Score (IS)

One well-known method is the Inception Score (IS). This score helps assess how good the generated data is by using a pre-trained classifier called the Inception network. The Inception Score takes two things in consideration:

1. **Confidence**: Checks if the Inception network can confidently recognize what the generated data represents, a higher confidence level indicates that the generated data closely resembles the real data categories.
2. **Diversity**: Evaluates whether the generated images are varied and not just copies of each other, this assesses the variety in the generated data.

5.4.2 Fréchet Inception Distance (FID)

Another important metric we use is the Fréchet Inception Distance (FID). This metric compares the distribution of generated data with the one from the real data. The steps involved in computing this metric are:

1. **Application of the Inception Network**: Both generated and real data samples are processed through the Inception network to extract feature vectors. It assumes that the feature vectors of both the real and generated samples are distributed normally.

2. **Calculation of the FID**: This statistical measure calculates the distance between these two Gaussian distributions. A lower FID indicates that the generated data distribution more closely resembles the real data distribution.

To compute these scores we used the InceptionScore[1] and the FrechetInceptionDistance[2] classes provided by torchmetrics[3].

5.5 Time and communication size data collection

To answer our second scientific question, we collected data during training, such as the time for each operation (send data, receive feedback, perform optimization step, etc.) which compose an epoch on different type of nodes. These time metrics helps to decompose every operations performed in an epoch and therefore, identify what is the most costly action in terms of duration. In other perspective, this can also help finding the bottlenecks and where do we get the longest idle times, which allows us to find a way to optimize the distributed training procedure. Along with the time related metrics we also collected the communication size (in MB) involved in a given epoch for every node, in both direction (in/out).

[1] https://lightning.ai/docs/torchmetrics/stable/image/inception_score.html
[2] https://lightning.ai/docs/torchmetrics/stable/image/frechet_inception_distance.html
[3] https://lightning.ai/docs/torchmetrics/stable/

6. Experimental setup

6.1 PyTorch distributed package

PyTorch's distributed package c10d[1] allows for sending and receiving tensors across various processes, devices, and machines. It operates at a low level, achieving great performance while providing the user with considerable flexibility. Since we are already very familiar with the PyTorch library, it was a natural choice for us.

We used a TCP backend within this framework to enable strong network communication between machines over the network. This setup allows us to efficiently send feedback to the central server and distribute generated images to the workers, facilitating collaborative training of the GAN across multiple nodes. However, we cannot send dictionaries or complex objects, which are essential for the MD-GAN architecture, as we need to exchange the state of discriminators to avoid overfitting. Therefore, we decided to use TensorDict[2] [Bou et al., 2023], a module integrated into PyTorch, that turns dictionaries into tensors and vice versa. It is also capable of sending and receiving these tensors across different processes using the same TCP backend, fitting perfectly with our requirements.

6.1.1 Backend choice

PyTorch's distributed package offers two main options, GLOO and NCCL, each with its own advantages and caveats. The primary difference lies in the location of the tensors being sent and received. With the NCCL backend, communication occurs directly through an NVIDIA device, allowing for better performance. In contrast, GLOO passes through the CPU, offering greater flexibility.

However, NCCL does not support having multiple nodes running on one GPU, which goes against our architecture as shown in figure 6.2. To take full advantage of the computational power of our instances, we run multiple workers on a single machine. Therefore, we chose to use the GLOO backend, offering us more flexibility in how many workers we instantiate on a Compute Engine. By doing so, since the tensors are processed on the GPU in our experimental setup, we have to move them to the CPU to allow for sending and receiving data. This introduces some latency as messages are transferred back to the GPU for processing, but this trade-off is justified by the flexibility it offers.

In cases where only a single worker lies in each instance, our novel swapping strategy makes it possible to use NCCL which would boost the performance.

6.2 Baseline Comparison

To evaluate the performance of our distributed MD-GAN implementation, we compared it to a standalone centralized implementation. This baseline runs on a single machine with a single NVIDIA T4 GPU to train the GAN on the entire dataset. This setup serves as a reference point, allowing us to measure the benefits of distributed training and the impact of spreading the training process across multiple machines. We used the same hyperparameters and training settings for both the distributed and centralized implementations to ensure a fair comparison. It is important to note that the centralized implementation requires all data to be on one machine, meaning the data must be owned by the machine running the training process. This is not the case for the distributed implementation.

[1] https://pytorch.org/docs/stable/distributed.html
[2] https://github.com/pytorch/tensordict

6.3 Datasets used

To evaluate our distributed Multi-Discriminator GAN implementation, we used three datasets: MNIST [LeCun et al., 2010], CelebA [Liu et al., 2015] and CIFAR-10 [Krizhevsky et al., 2009].

- The MNIST dataset consists of 60,000 grayscale images of handwritten digits with 10 classes (0-9). Each image is 28x28 pixels in size, summing up to 784 input features, it is a relatively small and simple dataset.
- CelebA consists of 202,599 colored image of celebrity faces, each of size 178x218 and which sums up to 116,412 input features (including 3 channels), to reduce the input size of our model we resize every image to 64x64 and preserve the 3 channels, ending up with 12,288 input parameters instead.
- The CIFAR-10 dataset consists of 60,000 32x32 colored images which sums up to 3,072 input features (including 3 channels), in 10 classes, with 6,000 images per class.

Even if CelebA contains more input parameters it is more conceptually simple than CIFAR-10, due to that fact that every image consists of single human face, the angle could vary but the subject is still the same. In the other hand CIFAR-10 contains very different classes, making CIFAR-10 the most complex data distribution to learn.

We believe that these three datasets are a great choice to accurately evaluate our implementation, they are widely used in the literature and cover various type of classes and use cases, especially in the computer vision field and generative models.

In all our experiments, we distributed the data in an IID (Independent and Identically Distributed) manner across the workers. However, it is possible to distribute the data non-IID (referring to B), which could produce different results. Although this feature is implemented, we chose not to include it in our study due to time constraints.

6.4 Models

For each dataset in our experiment, we need to define a generator model, a discriminator model, a class to load the dataset (Partitioner), and some global properties of the dataset. To do this, we created an application architecture that makes it easy to add new datasets. The `datasets` folder should have one file for each dataset, and each file must include the following three classes and two constants:

- `Partitioner`: This class is responsible for loading the dataset and providing methods to perform operations such as partitioning or selecting a subset of data.
- `Discriminator`: This class defines the architecture of the discriminator for the dataset.
- `Generator`: This class defines the architecture of the generator for the dataset.
- `SHAPE`: This global constant specifies the shape of the data that the server will use to generate fake batches for the workers. It should be a tuple of three integers (`Tuple[int, int, int]`) representing the number of channels, the width in pixels, and the height in pixels.
- `Z_DIM`: This constant specifies the dimension of the input noise that the generator will use to generate a new image.

We have established all the necessary components for evaluating MNIST, CIFAR-10, and CelebA. To find suitable model architectures adapted for each dataset, we opted for a DCGAN[1] for CIFAR-10 and CelebA. This choice is based on the expectation that, for more complex images,

[1] A DCGAN is a GAN that explicitly uses convolutional layers https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

a Convolutional Neural Network (CNN) will perform better than a Multi-Layer Perceptron (MLP). CNNs can learn many optimal filters to apply to the input image for specific tasks, taking advantage of the local connectivity of subsets of pixels since the filters are applied through convolution.

Additionally, a CNN can be viewed as a special type of feed-forward neural network where the weights are constrained. Unlike a feed-forward neural network, the units in the hidden layers of a CNN are not fully connected to the input units. Instead, each unit in the hidden layer is connected to a small region of the input. The weights are shared across the input space because the same filter is applied to each input location. In a feed-forward neural network, each input is connected to each hidden unit with different weights that are not shared and are learned independently (fully connected) [James et al., 2023]. However, CNN and pooling layers are more computationally expensive, so for small and simple datasets, using a Multi-Layer Perceptron is a reasonable choice.

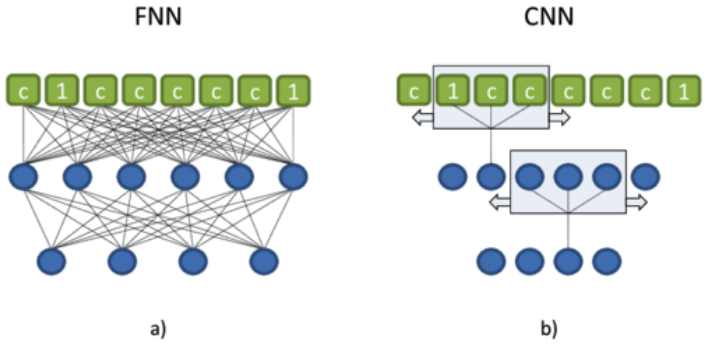

Figure 6.1: Illustration of a FNN and CNN weight sharing from [Winter et al., 2018]

Therefore, for MNIST, we chose to implement a Multi-Layer Perceptron due to its simplicity. Each original image in the MNIST dataset contains only a single channel, unlike the other two datasets, which are colored. Additionally, MNIST images are the smallest, with a size of 28x28 pixels.

The specific architecture of our models are specifed in the appendix A.

6.5 Launch scripts

Our distributed MD-GAN implementation allows running multiple nodes either on a single machine or on separate machines. Since each node operates as a separate Python process, they can communicate with each other as long as the network-related parameters are set correctly. To facilitate this, we created two scripts: `run-standalone.sh` and `run-distributed.sh`. These scripts help start the server and workers in different settings.

By default, we set the generator and discriminator learning rates to 0.0002, β_1 to 0.5, and β_2 to 0.999. The number of epochs is set to 3,000, and the batch size b is set to 10. These values are based on the optimal learning rates identified for the models corresponding to the datasets we train on. We chose a small batch size because it can transit over the network more quickly. A larger batch size would take longer per epoch because more data needs to move over the network.

The arguments for each script are defined at the beginning of the script. For the arguments shared by both scripts, we have an additional file named `shared-args.sh`, which defines the common parameters across our different experiment settings, ensuring consistency. Detailed descriptions of each argument are provided in the appendix B.

6.6 Google Cloud setup

We received $200 worth of Google Cloud credits from Prof. Dr. Lydia Chen and her teaching assistant Abel Malan in the context of the Distributed Deep Learning class at the University of Neuchâtel. We used these credits to launch our experiments on two Google Cloud Compute Engine instances. Each instance is equipped with an NVIDIA T4 TPU, 15GB of RAM, and 4 VCPUs. This setup allows us to use the `cuda` device to achieve shorter training times and provide faster results.

These two instances communicate with each other through a Virtual Private Cloud, which acts as a local private network. Virtually, both instances are on the same sub-network, enhancing security by avoiding to communicate over Internet. However, it is important to note that even though they are on the same virtual sub-network, they are located in different data centers: one in central Europe and the other in western Europe. This geographic separation is an important factor to consider when analyzing our results.

The network architecture of our application is shown in the figure 6.2.

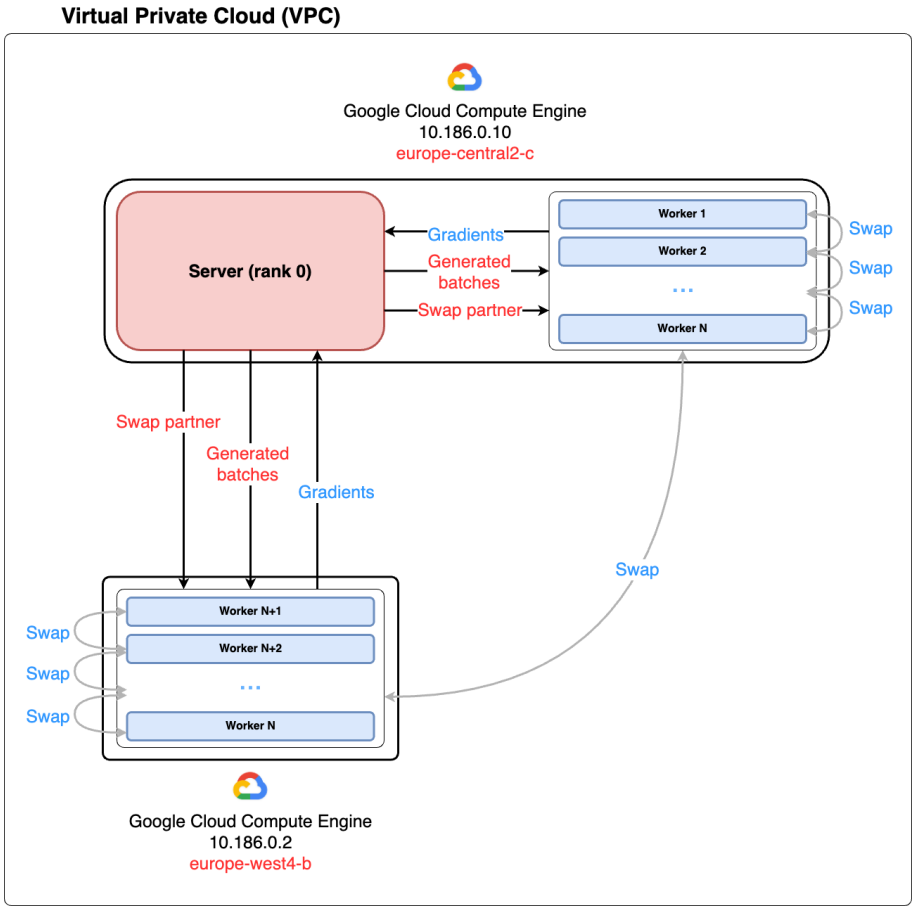

Figure 6.2: Experiments setup. To keep the scheme readable, we simplified the number of black arrows displayed. The three communication arrows represent the connections for all individual workers within the wrapping rectangle.

7. Results and experiments

To address our scientific questions, we conducted training experiments with various numbers of workers set at 4, 10, 20, and 40. All experiments were performed exclusively on the CIFAR-10 dataset due to time constraints, as it provides a sufficiently complex distribution to learn (refer to 6.3). During each epoch, we measured the time required to perform specific operations that constitute an epoch.

Due to the fact that our Google credits were running out rapidly, we had to stop the runs with 20 and 40 workers earlier than planned, as these configurations required the most time to complete an epoch. Consequently, the run with 20 workers was stopped at 10,000 epochs, and the run with 40 workers at 5,000 epochs. However, we allowed the standalone run and the distributed runs with 4 and 10 workers to complete their 30,000 epochs.

This decision was also motivated by the fact that while these experiments were running, we observed that we could already derive the necessary conclusions we needed to showcase our results without extending the runs to the full 30,000 epochs. Therefore, all the figures shown below are cropped at 5,000 epochs. Full data plots are in the appendix E.

7.1 Epoch duration

The figure 7.1 shows the time it took to complete each epoch. Compared to the standalone setting, the distributed experiments exhibit unpredictable outliers. To better visualize the typical duration of an epoch, we removed these outliers and present the refined results in figure 7.2.

As mentioned above, we cropped the results at 5,000 epochs to treat all settings equally. The full data plots are available in the appendix E.1.

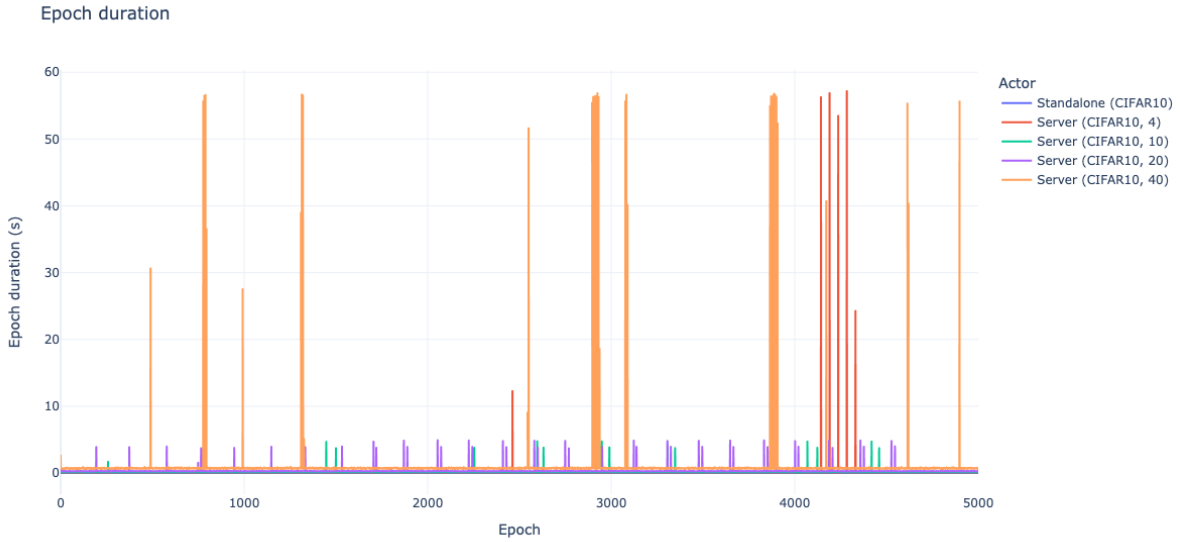

Figure 7.1: Epoch duration for all settings up to 5000 epochs

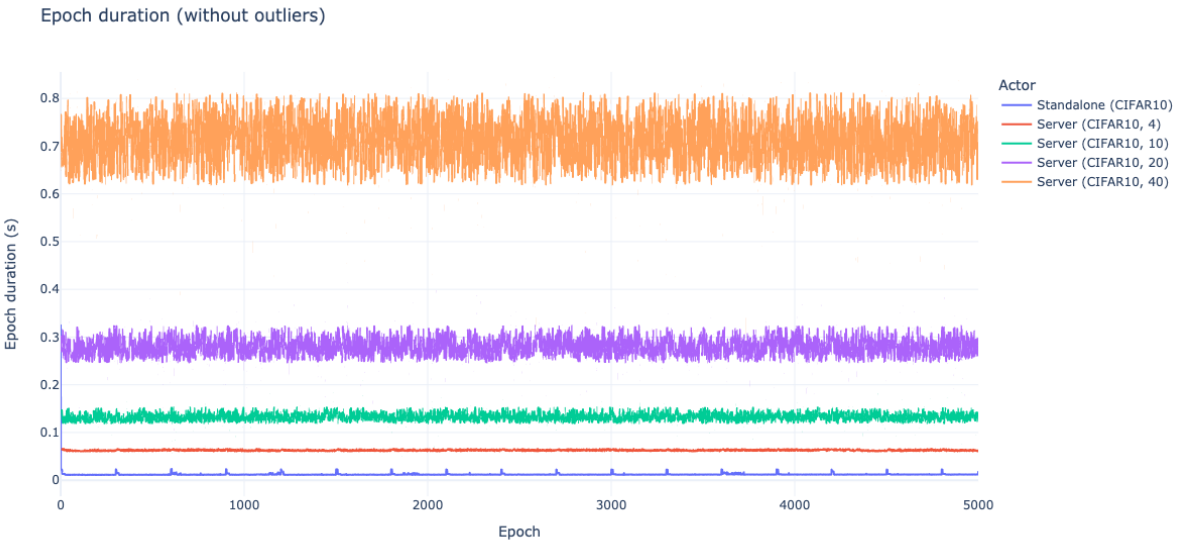

Figure 7.2: Epoch duration without outliers for all settings up to 5000 epochs

7.2 Communication size between the nodes

Figures 7.3 and 7.4 show the average communication size between the different nodes. We can observe that the size varies on the server side since more workers mean the server will send more data over the network as the number of workers grows. Specifically, the server sends $2N$ generated batches of images and receives N feedbacks in every epoch.

However, on the worker side, the communication size does not vary with the number of workers used during training. This is because the workers are not aware of each other, their communication size remains constant regardless of the world size. Workers only communicate with the server and, potentially, swap with one other worker. Therefore, the number of workers does not impact the size of data sent and received by each individual worker, keeping it constant as the world size grows.

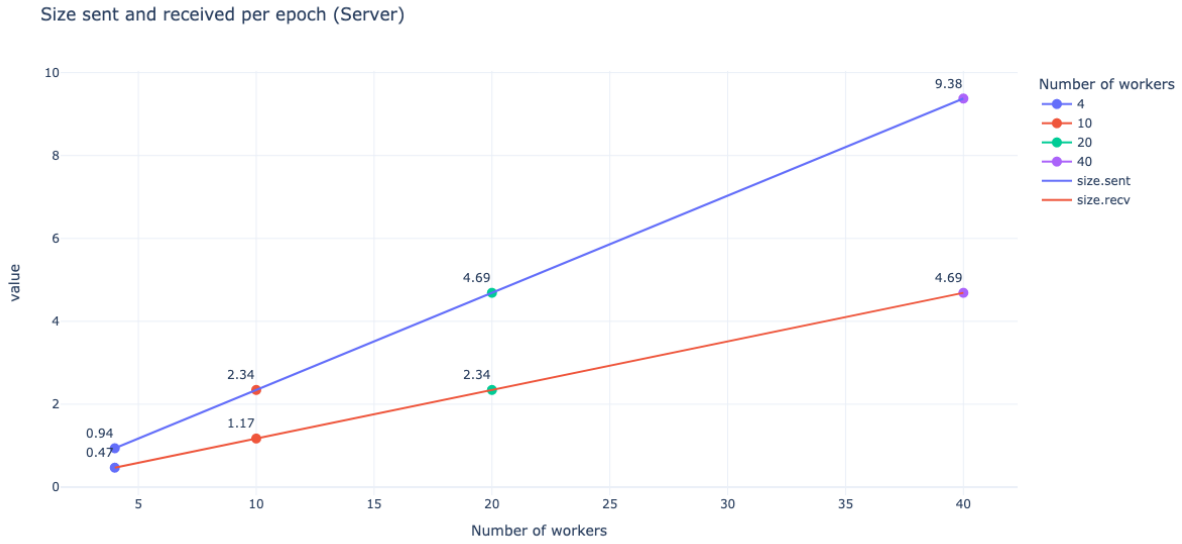

Figure 7.3: Average communication for every epoch on the server side. Blue: data sent over the network in MB, Red: data received over the network (MB)

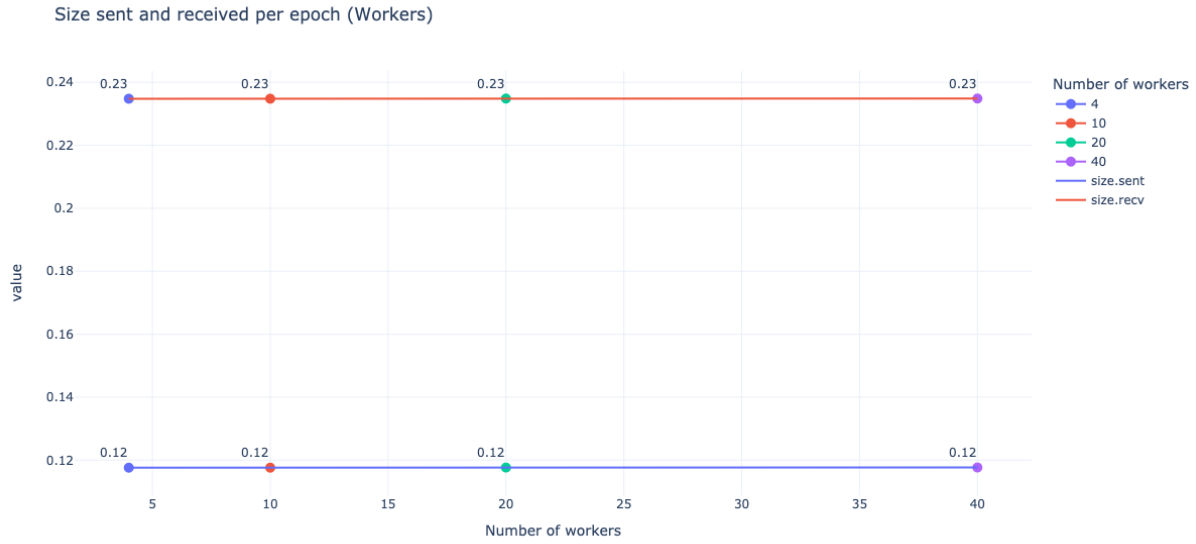

Figure 7.4: Average communication for every epoch on the workers side. Blue: data sent over the network in MB, Red: data received over the network (MB)

7.3 Average duration per world-size

The following plots might be the most important ones we provided, they directly answer our second scientific question 3.

To simplify the time series shown above, we present figure E.1 and 7.5, which illustrate the average time it takes to complete an epoch for each run with a specific world size. Using figure 7.5, we can identify how much the epoch duration increases as the world size grows. This helps us better understand the relationship between epoch duration and world size (linear, exponential, logarithmic, etc.).

We used the average calculation after removing the outliers of the time series (as for figure 7.2).

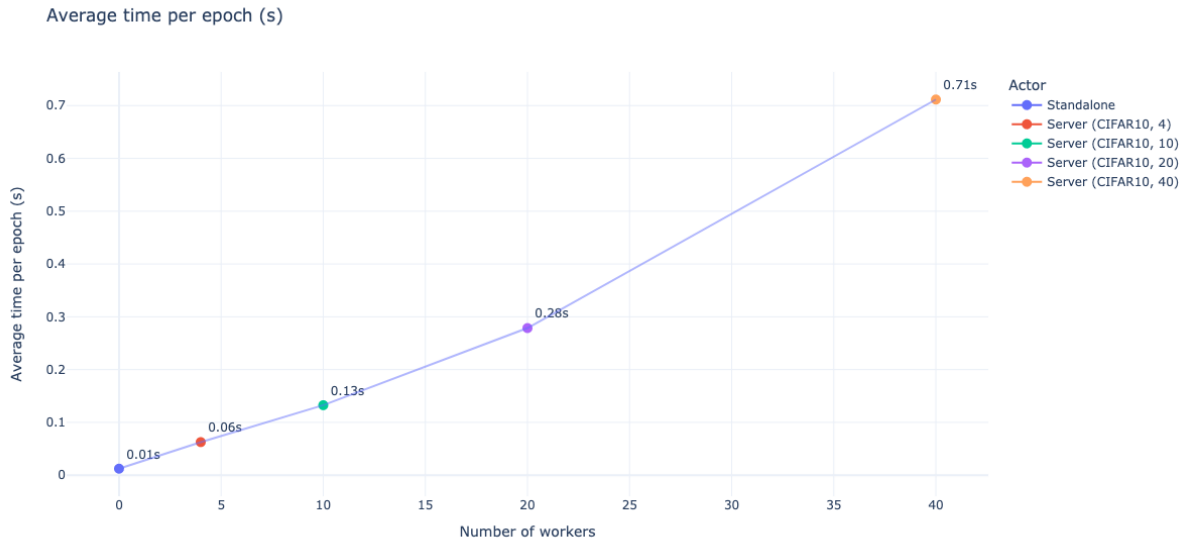

Figure 7.5: Relation between average time per epoch and world size

7.4 Average time elapsed per operation

To gain a deeper understanding of the epoch durations shown above, we split each epoch duration into the various operations done during an epoch. This segmentation allowed us to gather the average time taken per operation within an epoch. By doing so, we could identify the operations where our implementation spends the most time.

This analysis also enabled us to compare the scalability of different operations as the world size increases. Specifically, we could determine which operations do not scale well with an increasing number of workers and which ones remain relatively constant in duration.

The operations vary between the worker and the server, which is why the data related to the server are shown in separate plots, respectively in figures 7.6 and 7.7. Since this analysis is not relevant for the standalone setting, the plot related to it is available in the appendix E.2.

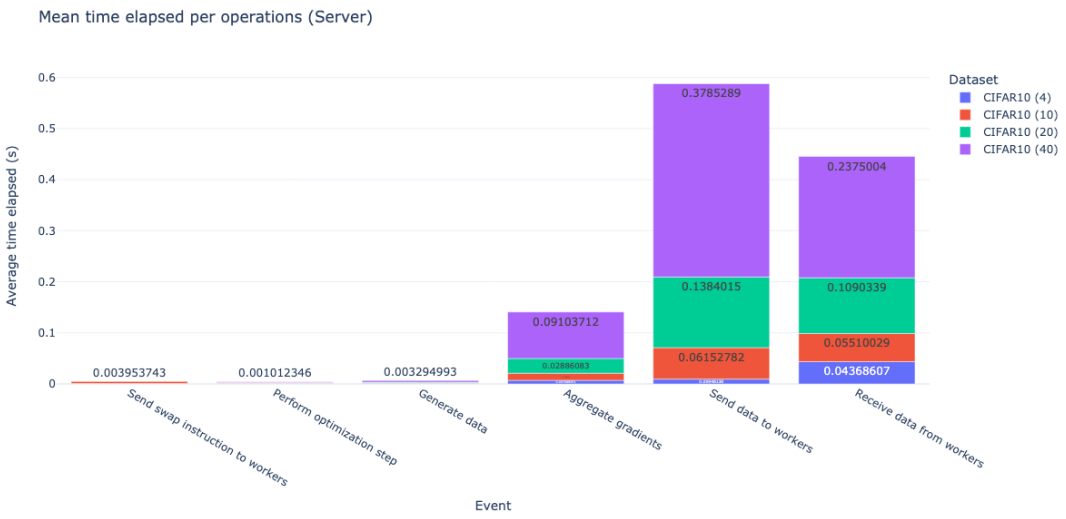

Figure 7.6: Average time elapsed per operation on the server

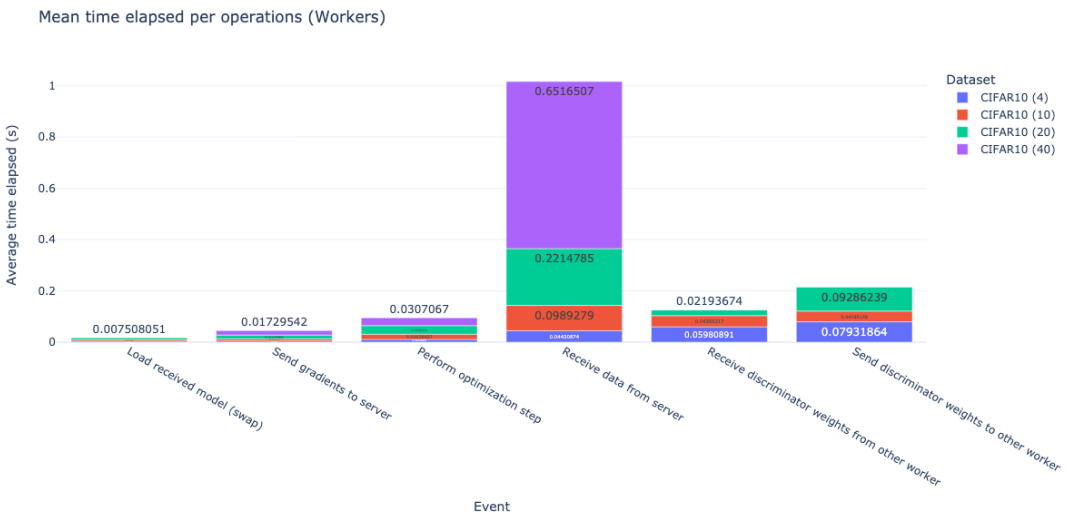

Figure 7.7: Average time elapsed per operation on the workers

7.5 Scoring metrics

Figures 7.8 and 7.9 show the FID and IS scores of the models in different settings as the number of epochs increases. These scores were computed at intervals of 300 epochs because calculating them at every epoch would significantly slow down the training and provide irrelevant time-related performance data.

As with the other figures, the complete plots are available in E.3. This plot demonstrates that the trends observed over the initial 5,000 epochs tend to continue beyond this point.

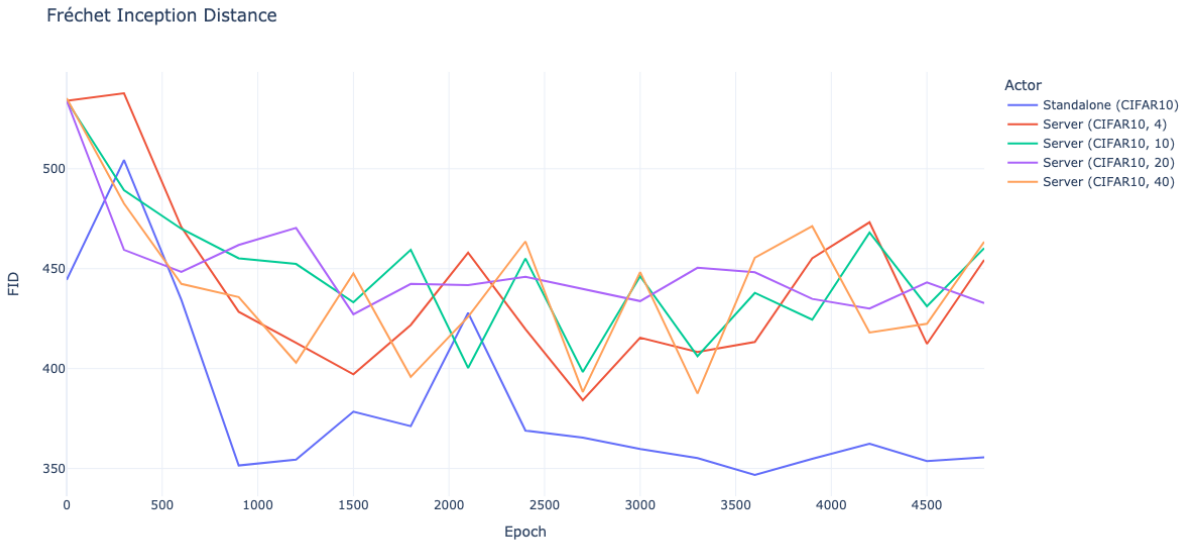

Figure 7.8: FID scores for different settings up to 5000 epochs

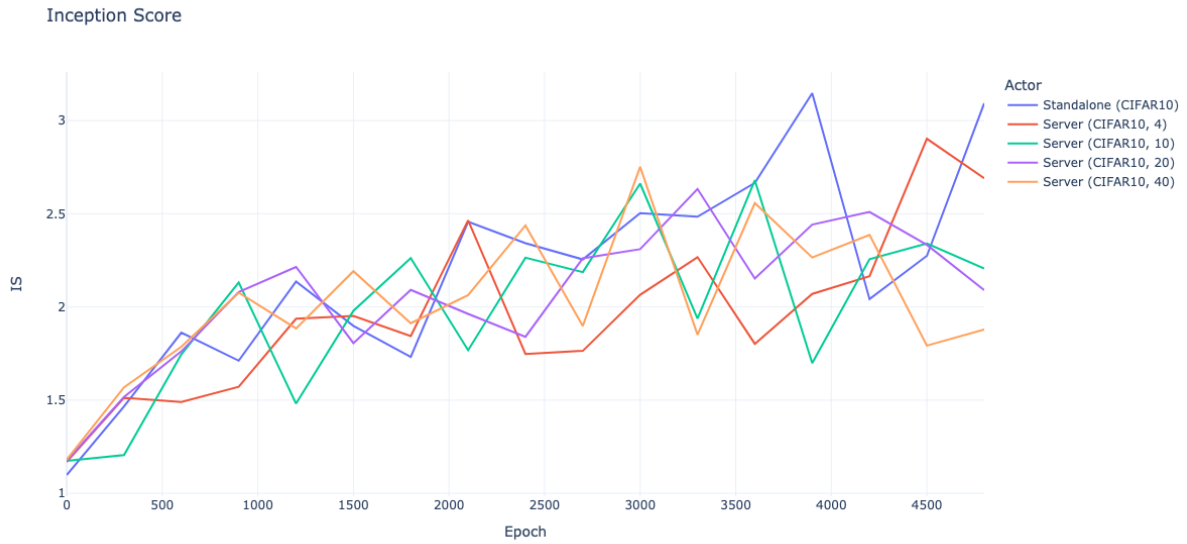

Figure 7.9: IS scores for different settings up to 5000 epochs

7.6 Timeline

Figure 7.10 shows the timeline over 10 epochs selected from the beginning of the training. It depicts the operations performed by the server and each worker over time, providing better insights into which operations induce more computation time. Additionally, unlike figures 7.6 and 7.7, this figure helps to identify idle times, indicating when a node is not doing anything except for waiting.

Along with the 10-epoch decomposition, we provide a zoomed-in view of one epoch in figure 7.11. This helps to visualize the typical decomposition of operations within a single epoch in a distributed setting.

This figure was created using the CIFAR-10 dataset with a 4-worker setting. We chose this configuration because such plots are computationally intensive to generate and difficult to interpret when many workers are involved. However, we believe that this figure provides sufficient insights for us to make meaningful conclusions.

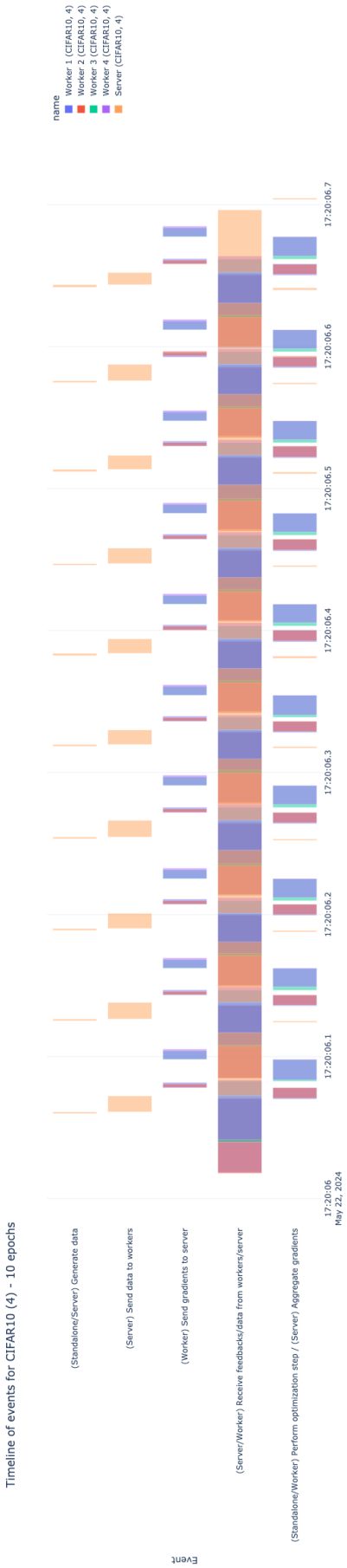

Figure 7.10: Timeline over 10 epochs for 4 workers

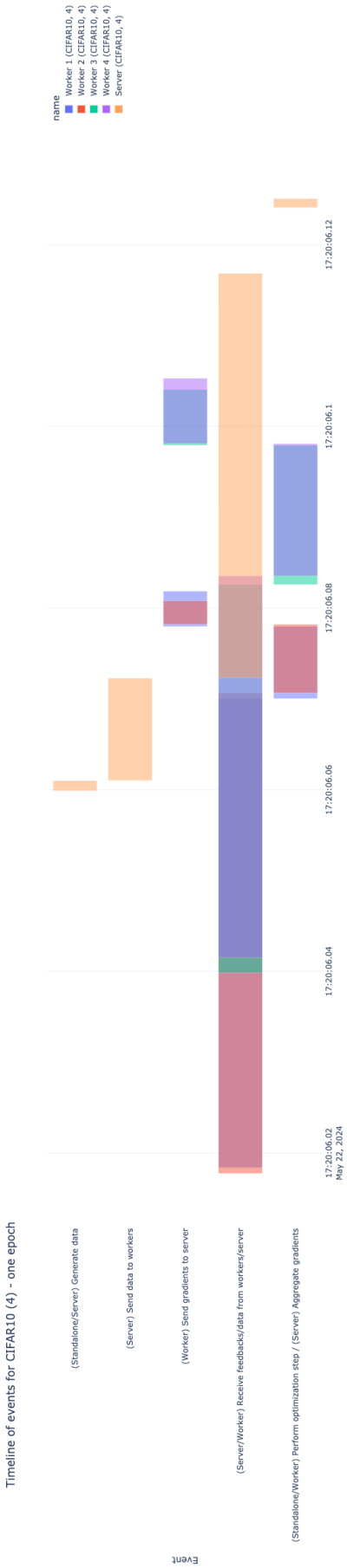

Figure 7.11: Timeline of a single epoch for 4 workers

7.7 Images

After training our models, we extracted the resulting generator models. The figures 7.12, 7.13, 7.14 showcase the capabilities of the models trained for 30,000 epochs for the standalone setting and for the distributed settings with 4 and 10 workers.

Just below, in figure 7.15 and 7.16, we extracted the generator models for the 20 workers setup after 10,000 epochs and for the 40 workers setup after 5,000 epochs.

At first sight, there are hardly any noticeable differences between the models trained for 10,000 epochs and those trained for 30,000 epochs. However, after a closer look, it becomes evident that the shapes and concepts generated by the model after 30,000 epochs are much closer to the training data with real images.

Figure 7.12: Generated images after 30,000 epochs in standalone setting

Figure 7.13: Generated images after 30,000 epochs with 4 workers

Figure 7.14: Generated images after 30,000 epochs with 10 workers

Figure 7.15: Generated images after 10,000 epochs with 20 workers

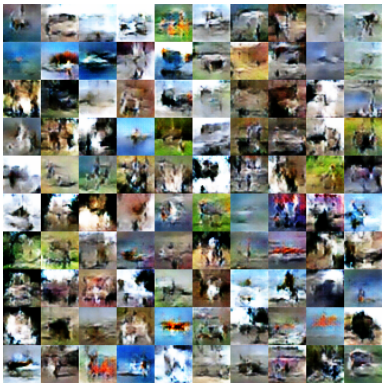

Figure 7.16: Generated images after 5,000 epochs with 40 workers

8. Discussion

8.1 Network disturbance

Figure 7.1 shows evident perturbations that are not the result of periodic calculations since the score computation is not taken into account in this figure, and the swapping of workers occurs at intervals of 5,000 epochs, which is also not considered. The most probable cause of these disturbance is network perturbations.

Referring to our network architecture in figure 6.2, we see that the instances are not located in the same datacenter, which could amplify these for potential network perturbations. Another argument supporting this hypothesis is that the standalone setting does not suffer from this problem, thereby discarding any potential local perturbations within the instance.

To validate this hypothesis, we retrieved all the time-related data for the perturbed epochs and analyzed the operations on which these problematic epochs spend the most time on. For both the server and the workers, we found that these issues are indeed related to network communications.

On the server side, figure 8.1 shows that during problematic epochs, the server spends most of its time sending data to the workers. Conversely, figure 8.2 demonstrates that during these problematic epochs, the workers spend most of their time receiving data from the server.

Therefore, we show that our implementation does not suffer from any random and unpredictable perturbations and that the bumps we see in figure 7.1 are indeed related to network perturbations. Since we identify the source of these perturbations it becomes more relevant to discuss the results after removing these outliers which could fake the conclusions we take from our data.

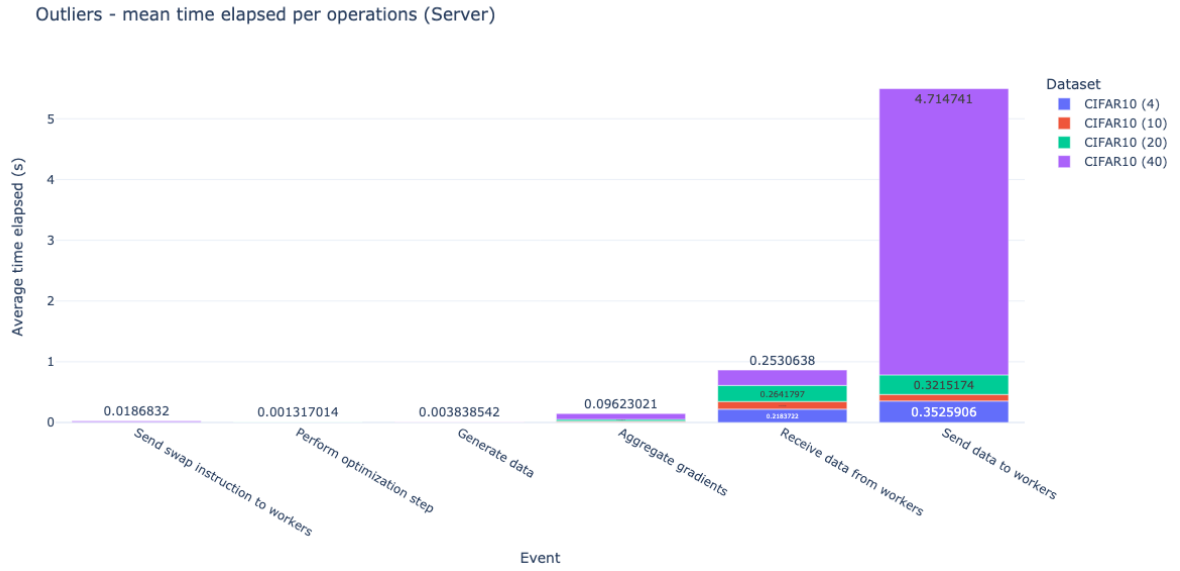

Figure 8.1: Average time elapsed per operation on the server during problematic epochs

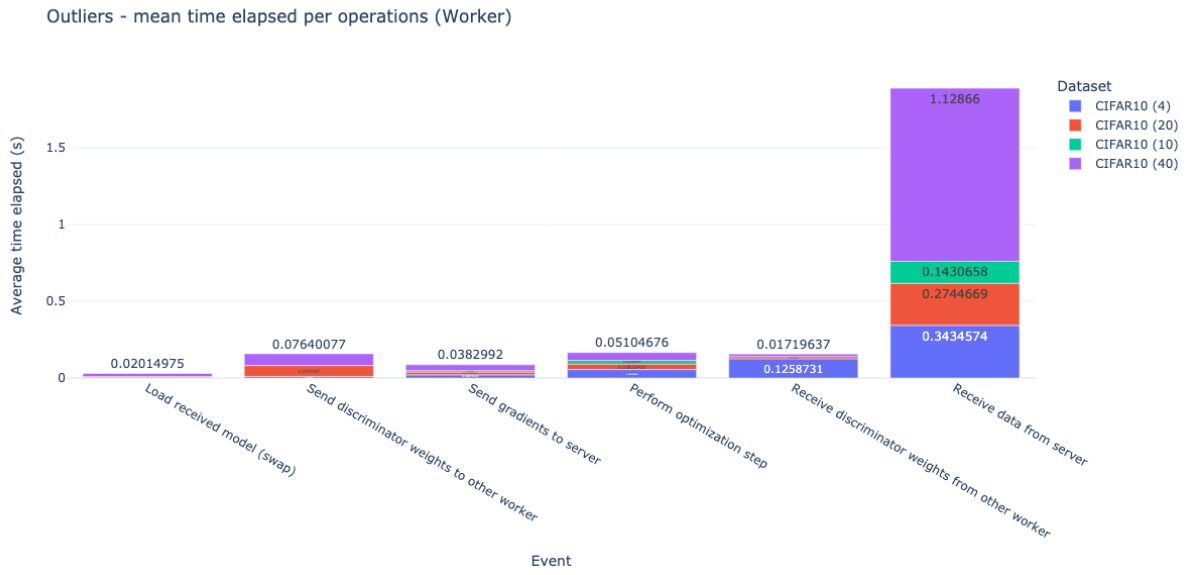

Figure 8.2: Average time elapsed per operation on the workers during problematic epochs

8.2 Potential linear relation #workers and epoch duration

Figure 7.5, which depicts the average time to complete an epoch (without outliers) in the different settings, shows that the relationship between the time of a typical epoch takes to complete and the number of workers involved is not completely linear. However, to fully assess and confirm these results, more runs should be performed on the same hardware and network configuration. It is possible that this trend is the beginning of a slowly growing exponential relationship.

Figure 7.3 provides an explanation for this linear-looking relationship. As the number of workers increases, the communication size grows linearly, since communication size naturally relates to the time required to transmit data over the network. Therefore, the linearly growing communication size curve translates to the one depicting epoch duration. We also observe that the impact of world size is primarily on the server side. To reduce this network overhead, solutions such as data compression could be employed, although they introduce additional compression and decompression overhead. More experiments should be conducted to determine the time savings that could be gained by using data compression.

We can confirm that within the interval from 4 to 20 workers, the relationship is almost perfectly linear, which is probably caused by the communication size which grows linearly as the number of workers grows. This could indicate that for the 40-worker setting, there is a small time-consuming factor that does not significantly affect the linear relationship at smaller world sizes. In the 40-worker setting, however, this small factor adds up and becomes significant enough to suggest a slowly growing exponential relationship.

8.3 Longest operations

Even after removing the outliers, figures 7.6 and 7.7 show that the most time-consuming operations are still related to the network, rather than any kind of computation. Specifically, the most time-consuming task for the workers is receiving data from the server, while for the server, it is sending data to the workers. The second most time-consuming task for the server is receiving gradients from the workers.

Figure 7.6 and 7.7 shows The first non-network-related task that is the most time-consuming is aggregating the received gradients. This suggests a potential area for algorithmic optimization that could improve efficiency in that part of the process.

This directly answers our second scientific question:
"Does training GANs in a distributed setting slow down the time to complete an epoch compared to a standalone setting?"
Yes, it does. The number of workers in the distributed setting results in epochs with a longer duration. Thus, we confirm that our second hypothesis **H2** is true. Additionally, we determined that there is a potential "near-linear" relationship between the number of workers and the average time it takes to complete an epoch.

As a side note, the time elapsed for sending and receiving data between the server and the workers does not perfectly match because the server does not wait for the workers to complete the data reception before considering the sending operation complete on its side.

8.4 Harder to converge for the distributed setting

Figures 7.8 and 7.9 both suggest that the distributed settings do not converge as well as the standalone version. This is likely due to the fact that the aggregated gradients tend to cancel each other out when averaged, making it difficult for the model to converge to an optimal solution.

To extend this analysis with a larger number of epochs, we refer to figures E.6 and E.7 in the appendix. These figures display the scoring metrics more frequently and show the scores we collected beyond 5,000 epochs. They indicate that the 20-worker setting continues to provide similar values to the other distributed settings. This allows us to conclude that increasing the number of workers does not help the model converge to a better optimum faster than the standalone setting.

This directly answers our first scientific question:
"Does training GANs in a distributed setting helps converging to an optimal solution in fewer epochs than a standalone version?"
No, it does not. Our results show the opposite: it is harder for a distributed GAN to converge to an optimal solution. This is likely due to the fact that workers provide feedback to the server, and by averaging these feedbacks, some information is lost, thereby slowing down the convergence rate. Thus we can reject our first hypothesis **H1**.

As the FID and IS metrics aim to reproduce the capabilities of the human eye, one should be able to verify this statement using figures 7.12, 7.13, and 7.14.

As a side note, since the IS score is not bounded, it is difficult to define the exact point at which it indicates that the generated images are of high quality. On the other hand, an FID score close to 0 means that the generated images coincide very well with the original batch of images, indicating higher quality. For the FID score, we aim to get the lowest value possible. These metrics are primarily used for comparison with other works. For instance, the state of the art (SOTA) for GANs on CIFAR-10 can achieve an IS score around 9.00 and an FID score around 15.00 [Brock et al., 2019]. However, our work does not focus on obtaining SOTA results but rather on assessing the optimization aspects that a distributed GAN can still achieve.

8.5 Source of idle time

By analyzing the timeline figures 7.11 and 7.10 and the algorithm in C, we can identify the main source of idle time: the server waits until all the workers finish their training and send their feedback. This is difficult to optimize. For instance, one approach could be to use this idle time on the server side to pre-generate data for the next epoch. However, this would mean the discriminators would not be trained using the current state of the generator since the fake data would be generated before the generator takes its optimization step.

Similarly, from the worker's perspective, one might think a worker could start the next epoch immediately. However, this induces the same issue: if a worker starts the next epoch without waiting for the generator to complete its optimization step, the workers and the generator will be out of sync in terms of optimization steps taken, which is likely to penalize the training process.

A potential solution to minimize idle time without causing synchronization issues is to implement a timeout on the server while receiving feedback from the workers. This would mean the server waits until a specified timeout period elapses. If at least one feedback is received within this period, the server proceeds with the available feedbacks and discards all the other which arrived too late. This approach ensures that the server does not wait excessively for all workers while still maintaining synchronization with the current version of the generator. The main advantage is that it preserves a good convergence behavior while reducing idle time and improving performance.

9. A note on the Open Science principles

Our project utilized open-source software, ensuring that all code and development environments are accessible to the research community and beyond. We developed our MD-GAN using only open-source dependencies to facilitate ease of use and adaptation by other researchers. This approach promotes transparency, collaboration, and reproducibility in research, allowing others to build upon and extend our work.

Included within these dependencies are the publicly available datasets we used to produce our results: MNIST, CelebA, and CIFAR-10. These datasets are open and widely used, forming the core of our experiments and allowing us to benchmark our model's performance against established standards. By detailing our data usage and experimental setups in our documentation, we ensure that other researchers can replicate our studies or extend them with new data under similar conditions.

The entire implementation of our GAN model and all the tools to reproduce our results, including the architecture and algorithms, are available on GitHub[1].

A key aspect of our project's alignment with open science is our commitment to reproducibility. We have documented all experimental procedures, model configurations, and hyperparameter settings. This comprehensive documentation ensures that others can reproduce our results and verify our claims. Additionally, by reproducing the results from the foundational MD-GAN paper, we contribute to the validation of previous findings within the community.

[1] https://github.com/darmangerd/distributed-gan

10. Limitations

10.1 Scalability and computational overhead

While the MD-GAN framework facilitates the distribution of GAN training across multiple nodes, this approach introduces significant computational overhead. The need to synchronize discriminators and manage communication between nodes can lead to bottlenecks, particularly as the number of nodes increases. In our implementation, although deploying on Google Cloud allowed for scalability, the network latency and data transmission costs sometimes overcome the benefits of distributed processing. This reflects a common challenge in distributed systems: optimizing the trade-off between scalability and efficiency.

10.2 Dependency on network infrastructure

The performance of distributed GANs relies on the network infrastructure. Effective training requires a robust and reliable network to facilitate the frequent exchange of model parameters and discriminator states. In our experiments, variations in network quality, such as bandwidth fluctuations and connectivity issues, occasionally disrupted the training process and led to inconsistent model performance. This dependency highlights the need for a stable, controlled and monitor network environment to fully leverage the potential of distributed Deep Learning systems more generally.

10.3 Privacy concerns

Despite the distributed nature of MD-GAN offering advantages in data privacy by keeping data localized, complete privacy cannot be guaranteed. The exchange of discriminator models among nodes exposes the system to potential security vulnerabilities, such as inference attacks where sensitive information could be reconstructed. This show the need for more robust privacy-preserving mechanisms.

But the most concerning type of attacks which could cancel the benefit of keeping the workers' data locally, are the gradient inversion attacks, which aims to reconstruct the clients datas from the gradients received by the server

10.4 Reproducibility and stability

Finally, both the original paper and our project experienced challenges related to the reproducibility and stability of GAN training. The adversarial nature of GANs makes them sensitive to hyperparameter settings, initialization, and random seed values. Small changes in any of these can lead to significant variations in output quality and convergence performance. This issue is even more highlighted in a distributed setting, where ensuring consistency across various training sessions becomes more complex.

11. Conclusion

The implementation of the MD-GAN in a distributed setting has provided valuable insights into the complexities and potential of distributed Deep Learning systems. This research has shown both the benefits and challenges of training GANs across multiple nodes, especially regarding scalability, efficiency, and data privacy.

Our experiments have shown that distributed training can offer significant advantages in terms of scalability. By using multiple nodes and machines, we were able to distribute the workload over multiple nodes and exploit more computational and memory resources than a single machine could provide. This setup also allows the interconnect machines with different architectures and computing powers.

Additionally, the decentralized nature of the MD-GAN framework offers natural privacy benefits. Thanks to its distributed approach, sensitive data does not need to be centralized, which reduces the risk of data breaches. However, the exchange of model parameters and discriminator states among nodes still poses potential security risks. To further enhance privacy, using strong privacy-preserving methods, like advanced encryption methods and differential privacy techniques, is essential to protect sensitive information.

However, the distributed setting also introduced several challenges that need to be addressed to fully realize the potential of this approach. Network issues and communication overhead were identified as major bottlenecks, affecting overall training efficiency. The results showed that the relationship between epoch duration and the number of workers is almost linear, suggesting there are factors that increase training times as the number of nodes grows.

Moreover, the convergence of the distributed model was found to be less effective compared to the standalone version. The aggregated gradients from multiple workers tended to cancel each other out, making it harder for the model to converge to an optimal solution. This highlights some space for improvement regarding the aggregation process and the exploration of other methods to improve convergence in distributed settings.

In conclusion, this research has improved the understanding of distributed GAN training and provided a foundation for future improvements. By addressing the identified limitations and exploring new strategies for optimizing communication, we can enhance the performance and applicability of distributed GANs in various real-world scenarios. The open-source implementation and detailed evaluation presented in this study contribute to the ongoing development of distributed Deep Learning systems.

12. Improvements

12.1 Non-IID data and computational ressource

A method to distribute the data in a non-IID fashion was implemented, but we did not include results from this setting due to time constraints. However, studying the effect of non-IID data on the different metrics we collected could provide more insight into potential improvements needed for the MD-GAN architecture. In fact, transitioning from an IID to a non-IID setting could significantly impact convergence to an optimal solution, potentially necessitating updates to the model architecture.

Non-IID can manifest in various forms, not only in the form of data distribution. For instance, differences in computing power across different clients can lead to significant variations in processing speed. Therefore, an interesting aspect of the MD-GAN to explore is the impact of workers with varying computational power. This variation could lead to increased idle time between when the server sends data to the workers and when it receives their feedback. To address this idle time, a solution could be implemented using the method described in section 8.5.

Future work could include deeper analysis of these variations and their impacts, enhancing the robustness of the Multi-Discriminator GAN architecture.

12.2 Attack and defense mechanisms

Research on attacks and defenses in the field of distributed Deep Learning is very active, with many new methods emerging. One such attack is "model stealing," where an attacker replicates our model by querying it with their own data, and uses the outputs to reproduce the results the original model.x

Figure 12.1 describes another type of attack that could occur when deploying this model on a crowd of workers with potentially malicious intentions, known as a "free-rider" attack [Fraboni et al., 2021]. In this scenario, one of the workers might decide not to send accurate feedback to the server, significantly impacting the training of the generator. In our work, we assumed that all workers are honest and cooperative. However, in a situation where we deploy this model on a crowd of smartphones, an individual could easily tweak the feedback sent to our server, potentially disrupting the entire training process. These types of attacks are relatively easy to perform, but fortunately, some defenses exist to counteract them such as WEF-Defense [Chen et al., 2022] or STD-DAGMM [Fraboni et al., 2021].

In a more advanced version of our system, and depending on the specific use case, it's important to implement strategies to defend against free-rider nodes.

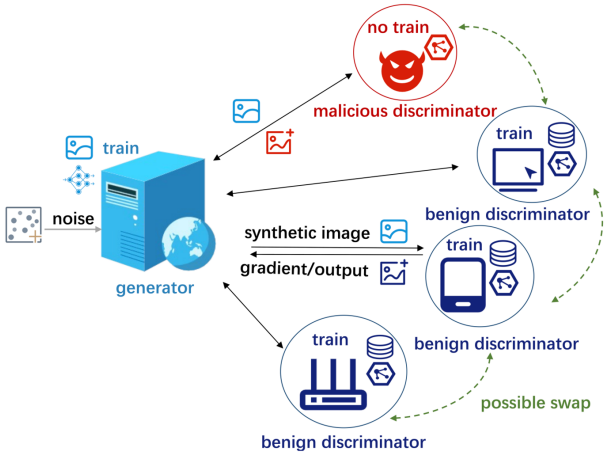

Figure 12.1: Source: Prof. Dr. Lydia Y. Chen for the Distributed Deep Learning class at University of Neuchâtel

12.3 Enhancing data privacy

The distributed nature of our model inherently supports data privacy by processing data locally at the client level. However, to reinforce privacy guarantees, implementing advanced encryption methods for data in transit could prevent interception and unauthorized access.

Another concerning type of attack is "gradient inversion", which occurs when the server attempts to reconstruct the real images held locally by the workers. Several studies have demonstrated the effectiveness of such attacks, including AGIC [Wang et al., 2020], InvG, and AGIC [Xu et al., 2022]. However, fewer works have been published on defense strategies. An interesting defense approach is detailed in [Huang et al., 2021], and the authors provide an open-source tool for their method, available at the following GitHub repository: `https://github.com/Princeton-SysML/GradAttack`.

In a potential production setting where the server could be malicious, it is crucial to consider using a defense strategy to keep the data completely invisible to the server. Implementing robust defense mechanisms can help protect the privacy and integrity of the data held by the workers.

12.4 Improving the model's performance

Lastly, optimizing the performance of our distributed GAN model is essential. Adjusting model configurations and tuning hyperparameters could lead to significant improvements.

Although such refinements might not drastically modify our main findings, they are crucial for achieving the best possible outcomes from our implementations. This process involves methodical experimentation and could produce valuable insights into the most effective configurations for distributed GANs.

A. Models

A.1 MNIST models

Based on https://github.com/lyeoni/pytorch-mnist-GAN.

Discriminator

1. **Input:** Image of shape 1x28x28 (784).
2. **Layer 1:**
   - Fully connected layer with 1024 units.
   - Leaky ReLU activation (slope 0.2).
   - Dropout (0.3).
3. **Layer 2:**
   - Fully connected layer with 512 units.
   - Leaky ReLU activation (slope 0.2).
   - Dropout (0.3).
4. **Layer 3:**
   - Fully connected layer with 256 units.
   - Leaky ReLU activation (slope 0.2).
   - Dropout (0.3).
5. **Layer 4:**
   - Fully connected layer with 1 unit.
   - Sigmoid activation.
6. **Output:** 1 value, probability that the input data is real.

Generator

1. **Input:** Latent vector of dimension 100.
2. **Layer 1:**
   - Fully connected layer with 256 units.
   - Leaky ReLU activation (slope 0.2).
3. **Layer 2:**
   - Fully connected layer with 512 units.
   - Leaky ReLU activation (slope 0.2).
4. **Layer 3:**
   - Fully connected layer with 1024 units.
   - Leaky ReLU activation (slope 0.2).
5. **Layer 4:**

- Fully connected layer with units equal to the number of pixels in the output image: 784.
- Tanh activation.

6. **Output:** 28x28 1-channel image.

A.2 CIFAR-10 models

Based on https://github.com/Ksuryateja/DCGAN-CIFAR10-pytorch.

Discriminator

1. **Input:** 32x32 3-channel image.
2. **Layer 1:**
   - Convolutional layer with 64 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Leaky ReLU activation.
3. **Layer 2:**
   - Convolutional layer with 128 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
   - Leaky ReLU activation.
4. **Layer 3:**
   - Convolutional layer with 256 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
   - Leaky ReLU activation.
5. **Layer 4:**
   - Convolutional layer with 1 filter, kernel size 4x4, stride 1x1.
   - Sigmoid activation.
6. **Output:** 1 value, probability that the input data is real.

Generator

1. **Input:** 100-dimensional latent vector.
2. **Layer 1:**
   - Transposed Convolutional layer with 512 filters, kernel size 4x4, stride 1x1.
   - Batch Normalization.
   - ReLU activation.
3. **Layer 2:**
   - Transposed Convolutional layer with 256 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
   - ReLU activation.
4. **Layer 3:**

- Transposed Convolutional layer with 128 filters, kernel size 4x4, stride 2x2, padding 1x1.
- Batch Normalization.
- ReLU activation.

5. **Layer 4:**
   - Transposed Convolutional layer with 3 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Tanh activation.
6. **Output:** 32x32 3-channel image.

A.3 CelebA models

Based on https://github.com/AKASHKADEL/dcgan-celeba.

Discriminator

1. **Input:** 3-channel image.
2. **Layer 1:**
   - Convolutional layer with 64 filters, kernel size 4x4, stride 2x2, padding 1x1.
3. **Layer 2:**
   - Convolutional layer with 128 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
4. **Layer 3:**
   - Convolutional layer with 256 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
5. **Layer 4:**
   - Convolutional layer with 512 filters, kernel size 4x4, stride 2x2, padding 1x1.
   - Batch Normalization.
6. **Layer 5:**
   - Convolutional layer with 1 filter, kernel size 4x4, stride 1x1.
7. **Output:** 1 value, probability that the input data is real.

Generator

1. **Input:** 100-dimensional latent vector.
2. **Layer 1:**
   - Transposed Convolutional layer with 512 filters, kernel size 4x4, stride 1x1.
   - Batch Normalization.
3. **Layer 2:**
   - Transposed Convolutional layer with 256 filters, kernel size 4x4, stride 2x2, padding 1x1.

- Batch Normalization.

4. **Layer 3:**
    - Transposed Convolutional layer with 128 filters, kernel size 4x4, stride 2x2, padding 1x1.
    - Batch Normalization.
5. **Layer 4:**
    - Transposed Convolutional layer with 64 filters, kernel size 4x4, stride 2x2, padding 1x1.
    - Batch Normalization.
6. **Layer 5:**
    - Transposed Convolutional layer with 3 filters, kernel size 4x4, stride 2x2, padding 1x1.
7. **Output:** 64x64 3-channel image.

B. Script arguments

B.1 Shared

These arguments are shared across `run-distributed.sh` and `run-standalone.sh`.

- **batch_size**: The batch size of the data being processed in every epoch. It determines the number of samples that will be propagated through the network in a single pass.
- **discriminator_lr**: The learning rate for the discriminator. It controls how much to change the model in response to the estimated error each time the model weights are updated.
- **generator_lr**: The learning rate for the generator, similar in function to the discriminator learning rate.
- **dataset**: The dataset used for training. It has to match with the file name (case sensitive) in the `datasets` folder (MNIST, CIFAR10 or CelebA).
- **model**: The model used for training, which in this setup always corresponds to the dataset.
- **epochs**: The number of times the entire training dataset will be passed through the network.
- **local_epochs**: The number of epochs for local training in federated learning settings. In our implementation it is always set to 1.
- **iid**: Indicates whether the data distribution is independent and identically distributed (IID). A value of 1 means that the data is IID, 0 for non-IID.
- **n_samples_fid**: The number of samples used to compute the Fréchet Inception Distance (FID), a metric for evaluating the quality of generated images. Indicating a very uge value will consider all the images contained in the K batches generated for every client.
- **device**: The device used for computation, to run on the CPU indicate 'cpu', to run on a Apple device indicate 'mps' and to run on a NVIDIA GPU indicate 'cuda'.
- **log_interval**: The frequency (in terms of iterations) at which the training process logs information. This is because logging could take time and slow down the training, so we should avoid doing it at every iteration.
- **beta_1**: The exponential decay rate for the first moment estimates in the Adam optimizer, set to 0.0 in this setup.
- **beta_2**: The exponential decay rate for the second moment estimates in the Adam optimizer, set to 0.999 in this setup.

B.2 Standalone

These arguments are dedicted to the `run-standalone.sh` script.

- **seed**: The seed of the experiment, for reproducibility. The same seed will generate the exact same output from a run to another one.

B.3 Distributed

These arguments are dedicated to the `run-distributed.sh` script.

- **seed**: The seed of the experiment, for reproducibility. The same seed will generate the exact same output from a run to another one. Every worker will have a different seed which is the addition of their rank and the given seed, for instance if the seed is 1, the server (rank 0) will have the seed seed+0, worker 1 seed+1, worker 2 seed+2, ..., worker N seed+N.
- **world_size**: How many nodes participate to the training, including the server.
- **backend**: The PyTorch distributed backend to use, in our experiments we will always use GLOO because is allow for more flexibility. NCCL will not allow use to run multiple nodes on a single GPU device, only one can run per GPU, which isn't the case with the GLOO backend. However GLOO requires to move the data on the CPU before sending a tensor which makes the process of sending and receiving slower. If you attend to run a single node per GPU, use NCCL, otherwise use GLOO.
- **master_port**: The port the server will listen to.
- **master_hostname**: The hostname of the server. It can also be an IP address.
- **swap_interval**: The E multiplicator to the $i \mod \frac{mE}{b} = 0$ condition.
- **network_interface**: The network interface the programme should use to communicate, usually `lo` on Google Cloud, `en0` on Apple devices. To find a network interface which work on a Linux machine you should type `ip addr show` in a terminal.
- **ranks** (at script call): This argument should be a parameter given when calling the script. It provides the list of ranks that should be started, they can be separated by a comma (e.g. 0,1,2,3,4) to define the specific ranks to start, or by two dots to define an inclusive range (e.g. 0..30 will start the ranks from 0 to 30 included). An even number of workers should be started when calling the script otherwise the error `ValueError: The number of workers should be even` will be thrown.

C. Algorithm

Notation	Description
G	Generator
D	Discriminator
N	Number of workers
S	The rank of the central server (0)
W_n	Worker n
P_{data}	Data distribution
P_G	Distribution of generator G
w (resp. θ)	Parameters of G (resp. D)
w_i (resp. θ_i)	i-th parameter of G (resp. D)
B	Distributed training dataset
B_n	Local training dataset on worker n
m	Number of objects in a local dataset B_n
d	Object size (e.g., image in Mb)
b	Batch size
I	Number of training iterations
K	The set of all batches $X(1), \ldots, X(k)$ generated by G during one iteration
F_n	The error feedback computed by worker n
S	The frequency swaps occurs
J_{disc}	Discriminator loss function $J_{\text{disc}}(X_r, X_g) = \tilde{A}(X_r) + \tilde{B}(X_g)$
$\tilde{A}$	Discriminator loss on real data $\tilde{A} = \frac{1}{b} \sum_{x_i \in X_r} \log D_\theta(x_i)$
$\tilde{B}$	Discriminator loss on generated data $\tilde{B} = \frac{1}{b} \sum_{x_i \in X_g} \log(1 - D_\theta(x_i))$

Table C.1: Notation for MD-GAN algorithm

Algorithm 1 MD-GAN algorithm

procedure WORKER(C, B, I, L, b)
  Initialize weights for for D
  for $i \leftarrow 1$ to I **do**
    $X^{(r)} \leftarrow$ SAMPLES(B)
    $(X^{(g)}, X^{(d)}) \leftarrow$ RECEIVE_BATCHES(C)
    for $l \leftarrow 0$ to L **do**
      $D \leftarrow$ LEARNING_STEP(J_{disc}, D)
    end for
    $F \leftarrow \{\frac{\partial \tilde{B}(X^{(g)})}{\partial x_i} \mid x_i \in X^{(g)}\}$
    SEND_FEEDBACKS(C, F)
    if $i \mod S = 0$ AND $i > 0$ **then**
      PARTNER $\leftarrow$ RECEIVE_PARTNER(C)
      SEND_DISCRIMINATOR(PARTNER, D)
      $D \leftarrow$ RECEIVE_DISCRIMINATOR(PARTNER)
    end if
  end for
end procedure

procedure SERVER(k, I)
  Initialize w for G
  for $i \leftarrow 1$ to I **do**
    X $\leftarrow G_w$(GAUSSIAN_NOISE, $2k$)
    for $n \leftarrow 1$ to N **do**
      $X^{(g)} \leftarrow X^{(n \mod k)}$
      $X^{(d)} \leftarrow X^{(n \mod k)+1}$
      SEND($w_n, (X^{(d)}, X^{(g)})$)
    end for
  end for
  $F_1, \ldots, F_N \leftarrow$ RECEIVE_FEEDBACKS_FROM_WORKERS()
  Compute Δw according to $F_1, \ldots, F_N$
  for $w_i \in w$ **do**
    $w_i \leftarrow w_i +$ ADAM(Δw_i)
  end for
  if $i \mod S = 0$ AND $i > 0$ **then**
    SWAP_PAIRS $\leftarrow$ CREATE_NON_OVERLAPPING_PAIRS_OF_WORKERS()
    for PAIR $\in$ SWAP_PAIRS **do**
      SEND_PARTNER(PAIR$_0$, PAIR$_1$)
      SEND_PARTNER(PAIR$_1$, PAIR$_0$)
    end for
  end if
end procedure

D. Operations

D.1 Server

1. `generate_data`: The server generates the $2k$ batches of b images.
2. `send_data`: The server send the batches to the corresponding workers, therefore sending N tensors to the workers over the network (each of shape $2b$).
3. `recv_data`: The server receive the feedbacks from the workers after they finished training, therefore receiving N tensors to the workers over the network (each of shape b).
4. `agg_gradients`: The server aggregates the gradients and compute δ_w.
5. `calc_gradients`: The server take an optimization step using the previously computed δ_w.
6. `swap`: The server generates the swap pairs to the workers, inducing N tensors to send to the workers (each containing only one integer).
7. `epoch_calculation`: This operation gather all the operations listed above (1 to 6).
8. `start_fid`: The FID score is computed (at interval of 300 epochs).
9. `start_fid`: The IS score is computed (at interval of 300 epochs).
10. `epoch`: This operation gather all the operations listed above (1 to 9).

D.2 Worker

1. `recv_data`: The workers receive the two batches $X^{(g)}$ and $X^{(d)}$ from the server.
2. `calc_gradients`: The worker perform L local epochs ($L = 1$).
3. `send`: The worker compute the gradients it has to send to the server and send it.
4. `agg_gradients`: The server aggregates the gradients and compute δ_w.
5. `recv_swap_instruction`: The eventually worker receives the swap instructions from the server.
6. `swap`: The server generates the swap pairs to the workers, inducing N tensors to send to the workers (each containing only one integer).
7. `epoch_calculation`: This operation gather all the operations listed above (1 to 6).
8. `start_fid`: The FID score is computed (at interval of 300 epochs).
9. `start_fid`: The IS score is computed (at interval of 300 epochs).
10. `epoch`: This operation gather all the operations listed above (1 to 9).

E. Non-cropped results

E.1 Epoch duration

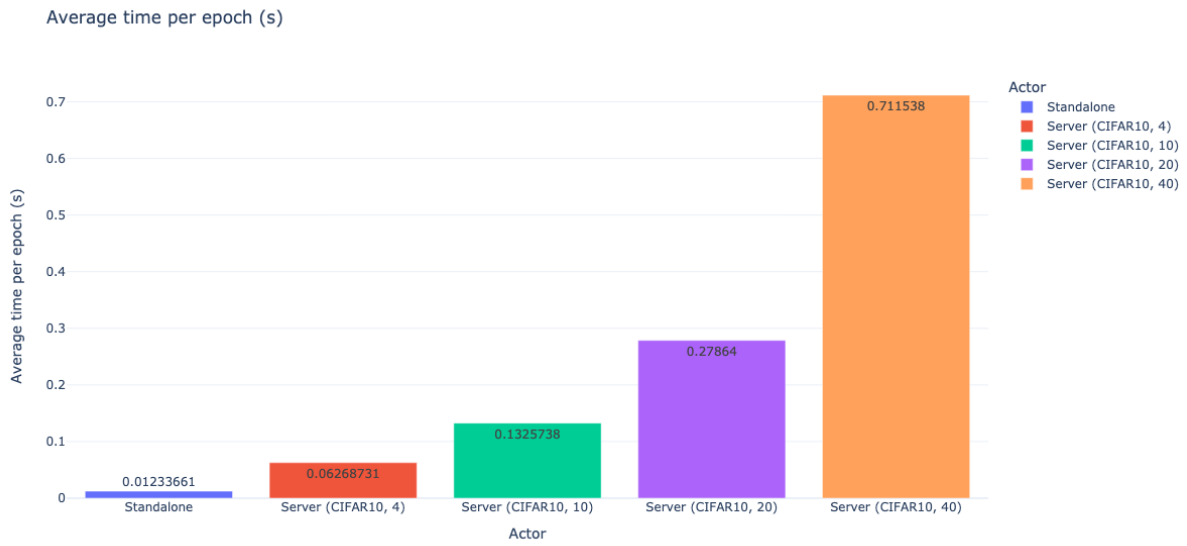

Figure E.1: Average time per epoch for different world sizes up to 5000 epochs

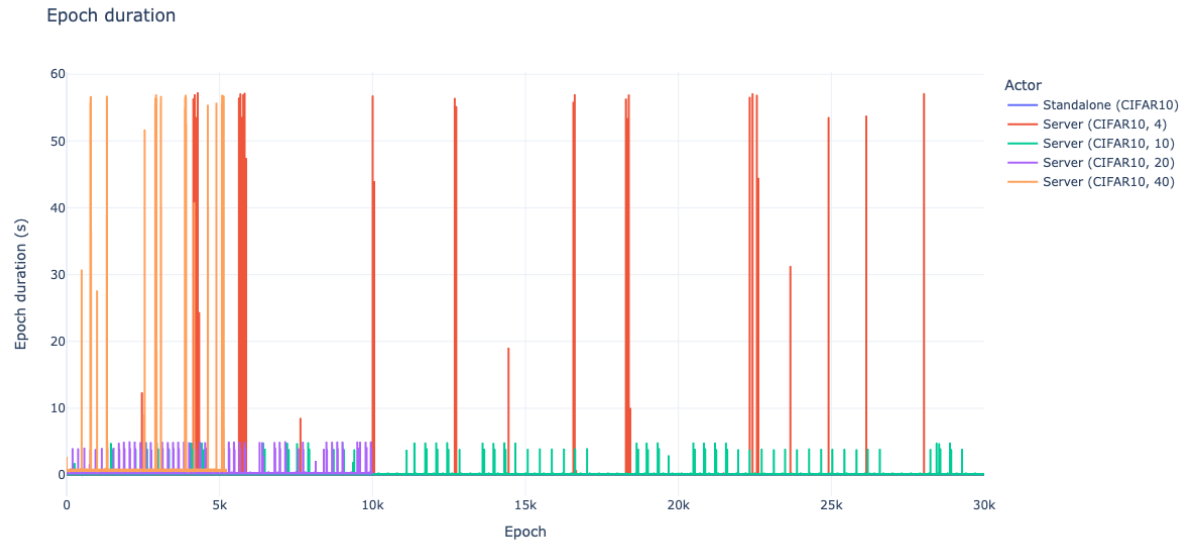

Figure E.2: Epoch duration for all settings

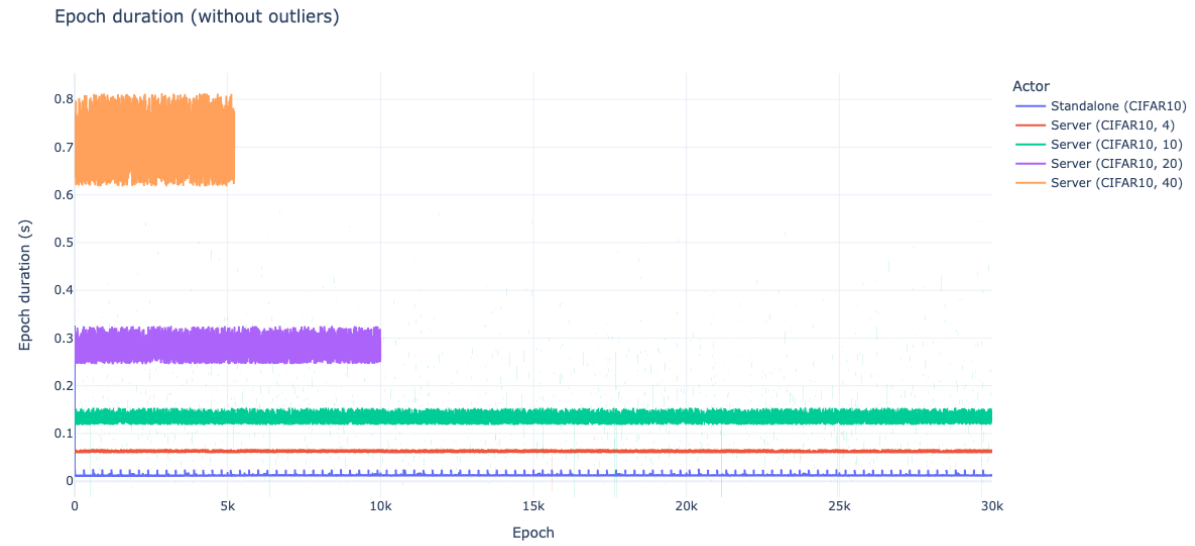

Figure E.3: Epoch duration without outliers for all settings

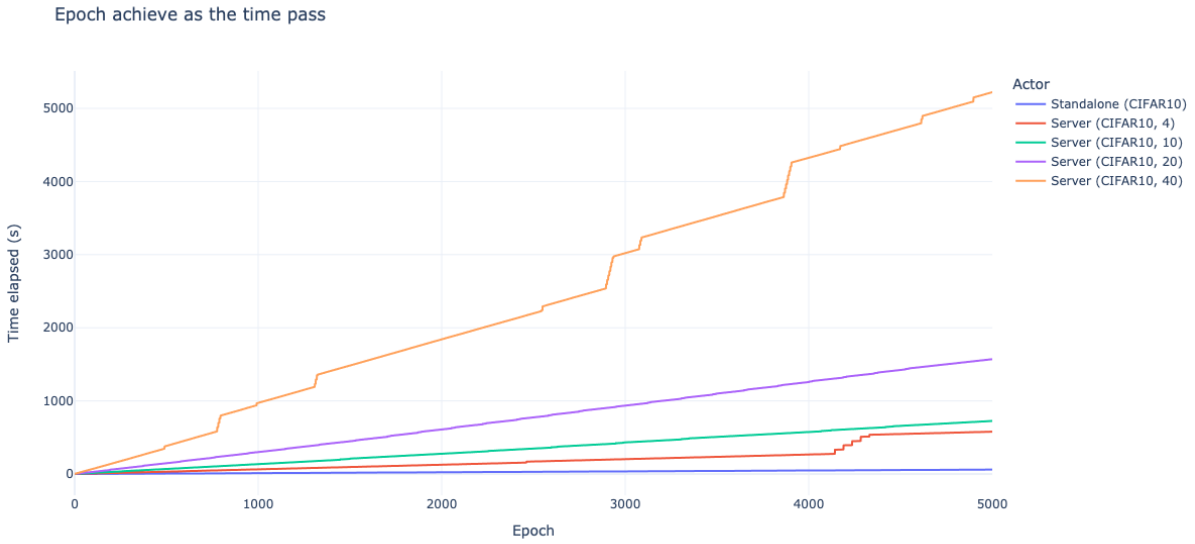

Figure E.4: Time taken to complete 5000 epochs for different world sizes

E.2 Average time elapsed per operations (standalone)

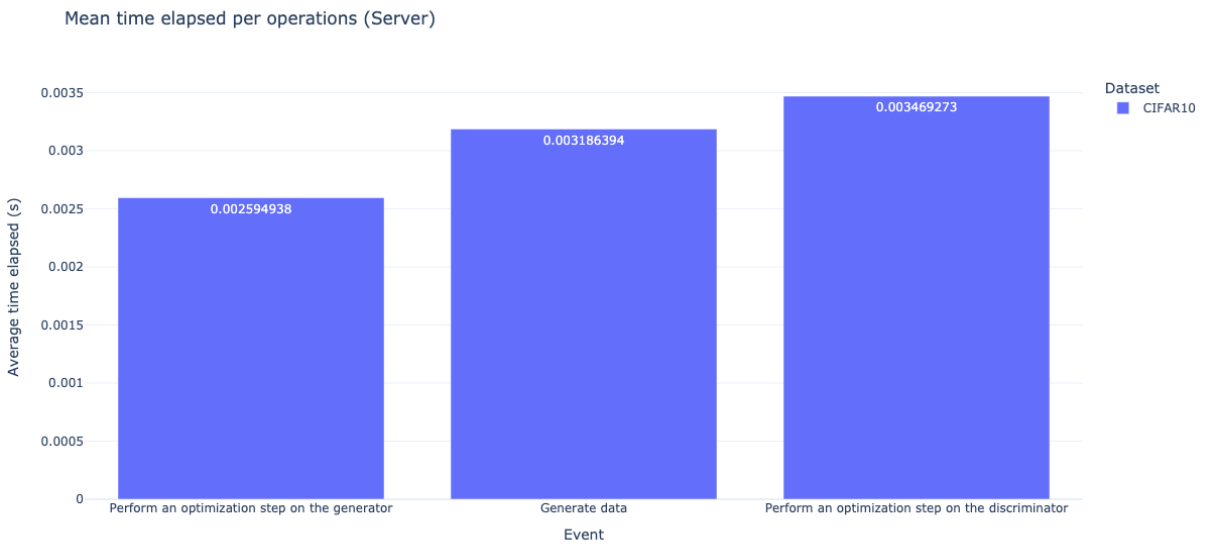

Figure E.5: Average time elapsed per operation in the standalone setting

E.3 Scoring metrics

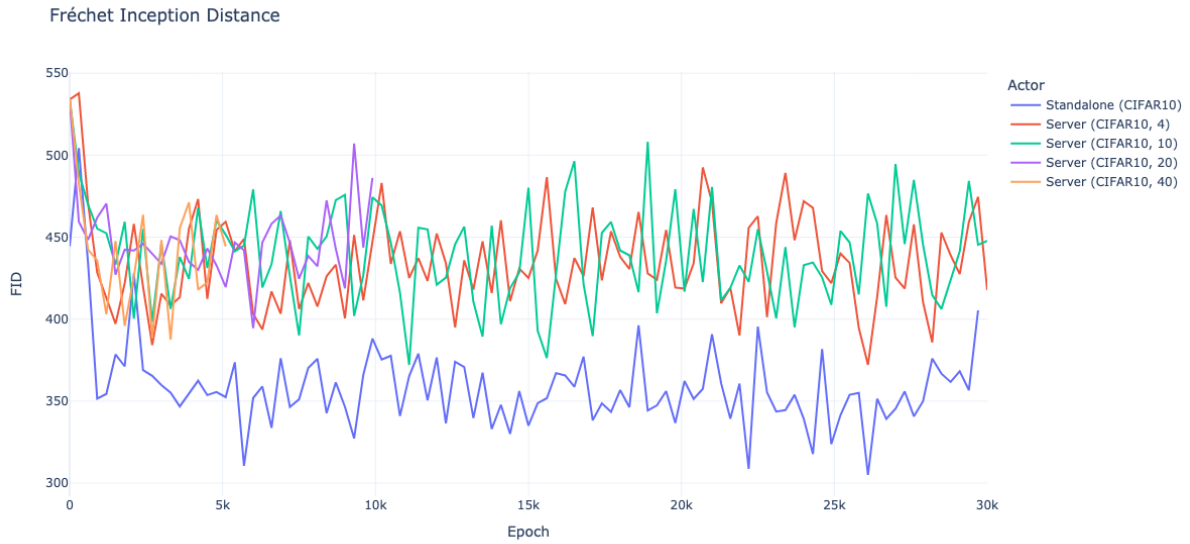

Figure E.6: FID scores for different settings

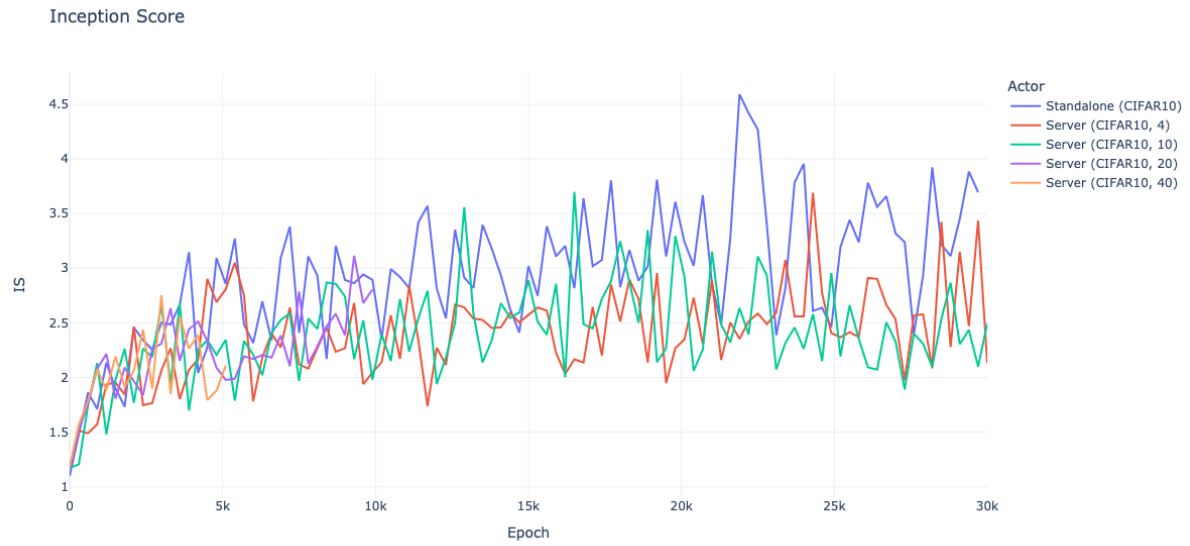

Figure E.7: IS scores for different settings

E.4 Epoch and time relation

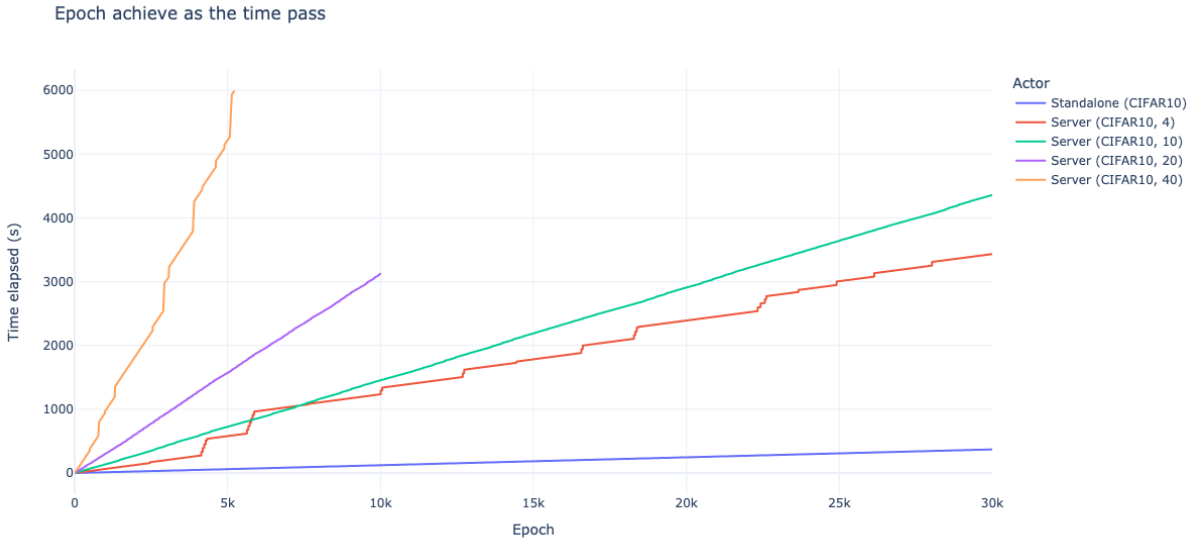

Figure E.8: Time taken to complete various numbers of epochs for different settings

F. Google Compute Engine statistics

F.1 CPU

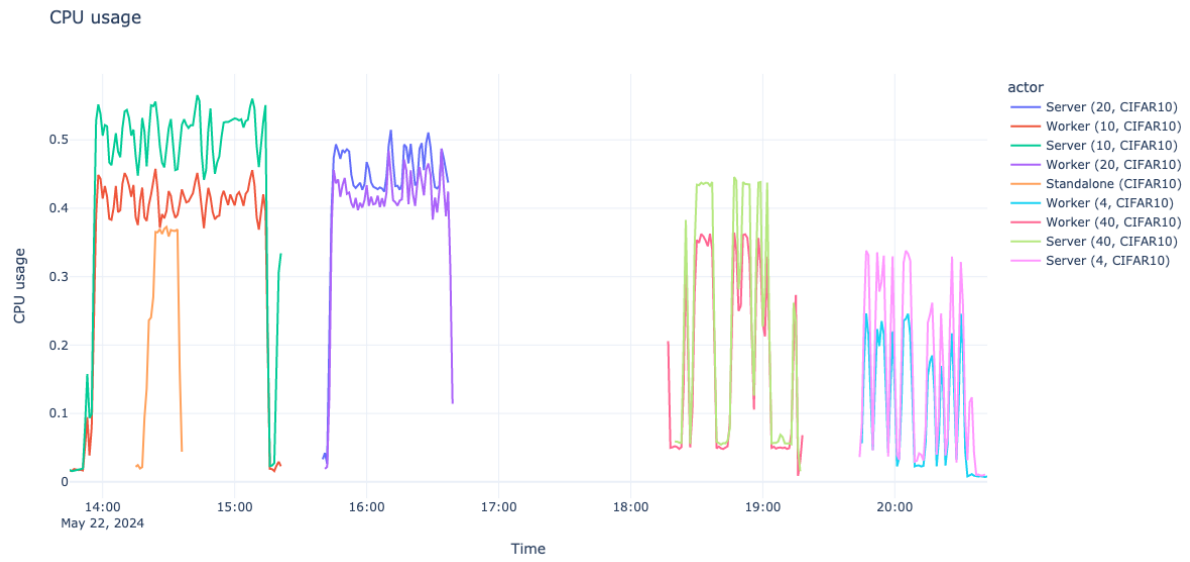

Figure F.1: Percent utilization of the vCPUs reservation for this VM machine type as measured by the hypervisor. This may differ from the CPU utilization as reported by the guest operating system. For shared-core VMs, this can go over 100% during bursting.

F.2 Disk

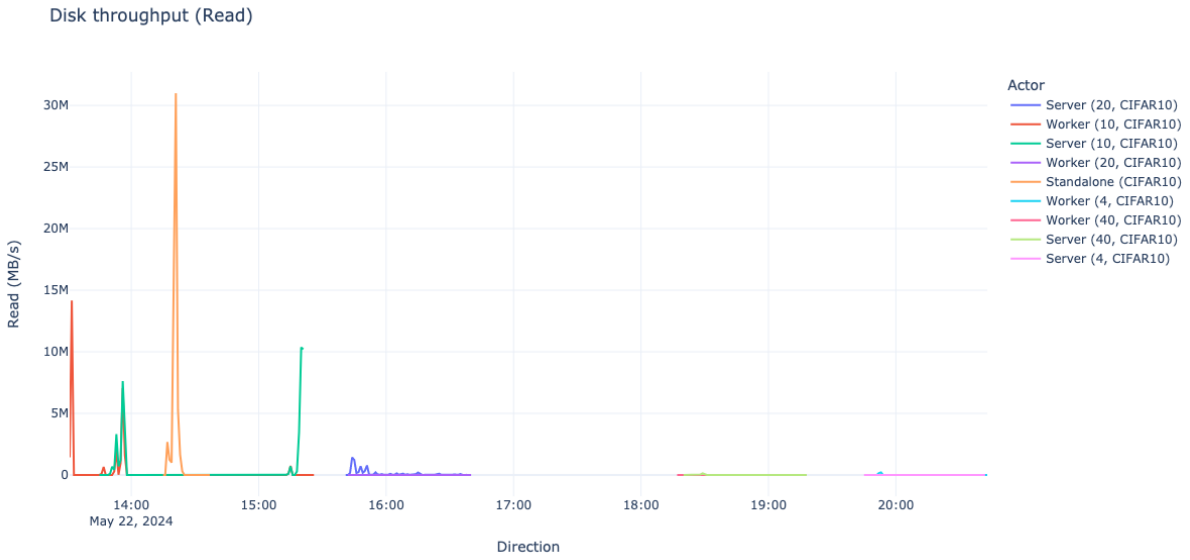

Figure F.2: The average rate of bytes written to and read from the VM's disks in one-minute time periods. Disk performance is influenced by a variety of factors like storage type, VM size, and disk size.

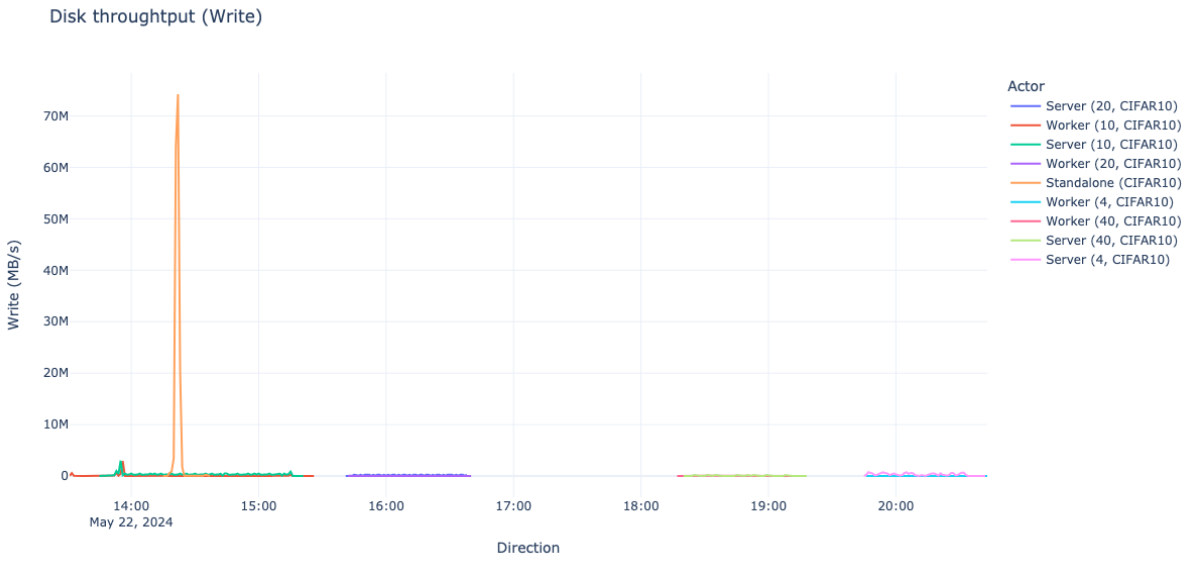

Figure F.3: The average rate of read and write I/O operations to the disk in one-minute time periods, grouped by the storage type and device type. Disk performance is influenced by a variety of factors like storage type, VM size, and disk size.

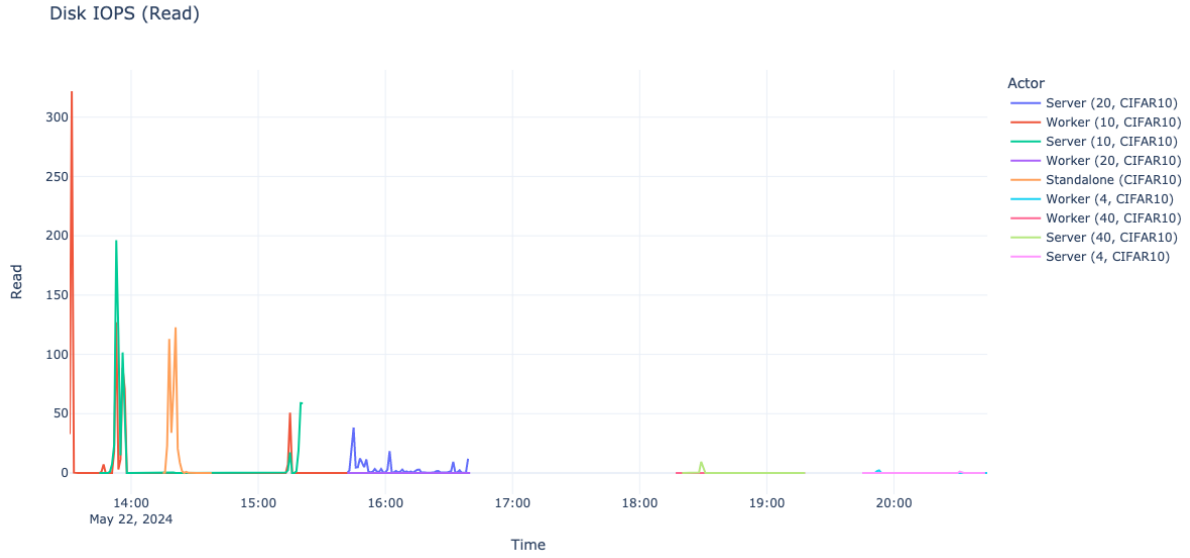

Figure F.4: The average rate of read and write I/O operations to the disk in one-minute time periods, grouped by the storage type and device type. Disk performance is influenced by a variety of factors like storage type, VM size, and disk size.

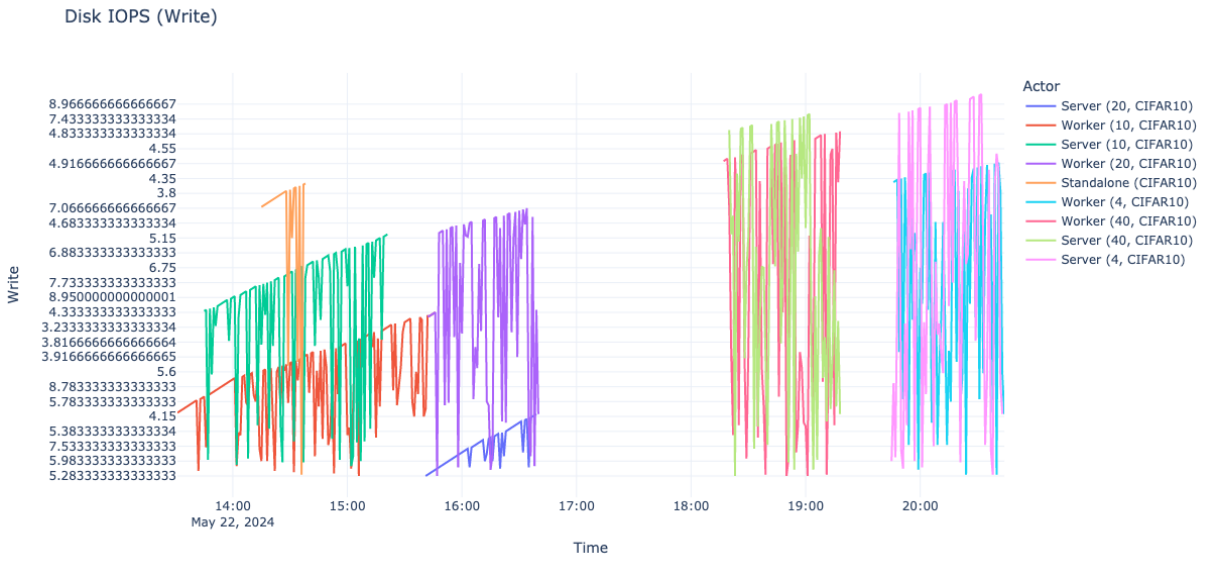

Figure F.5: The average rate of read and write I/O operations to the disk in one-minute time periods, grouped by the storage type and device type. Disk performance is influenced by a variety of factors like storage type, VM size, and disk size.

F.3 Network

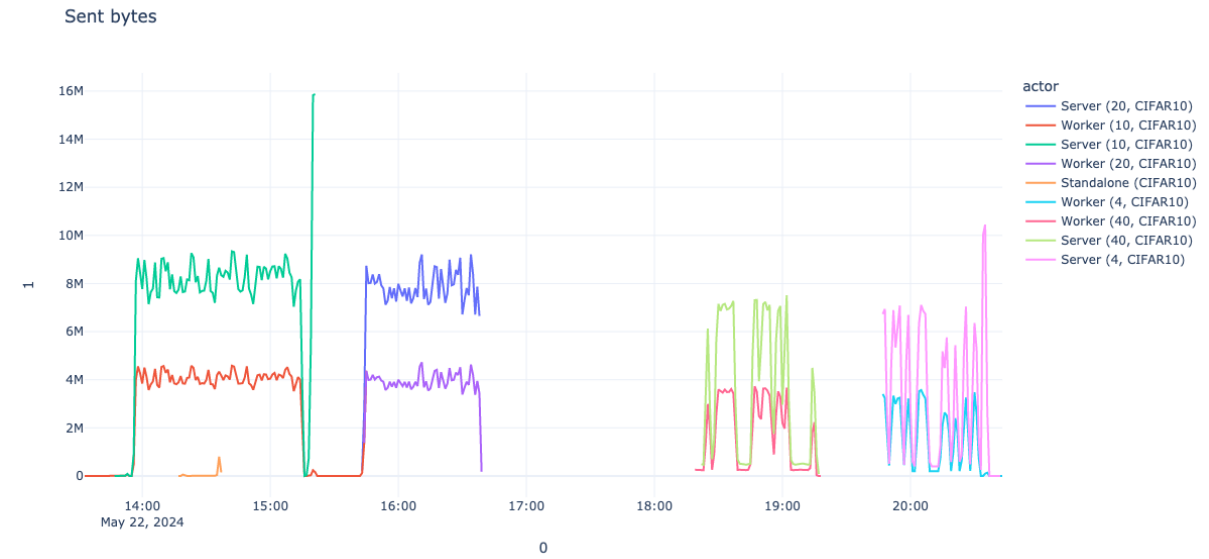

Figure F.6: Rate of sent and received network traffic bytes aligned in 1-minute intervals.

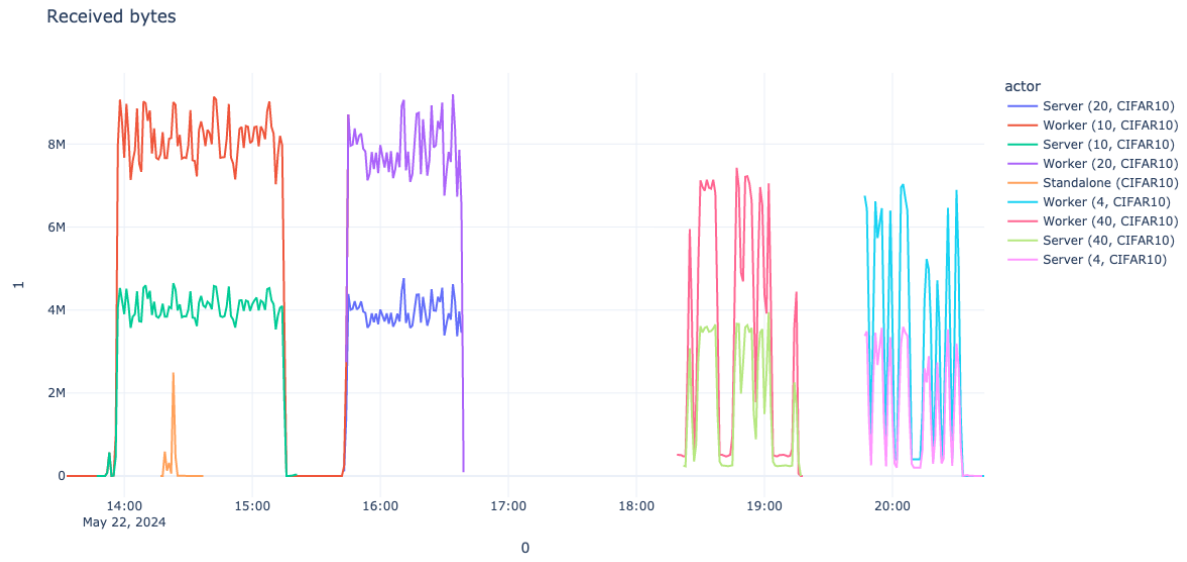

Figure F.7: Rate of sent and received network traffic bytes aligned in 1-minute intervals.

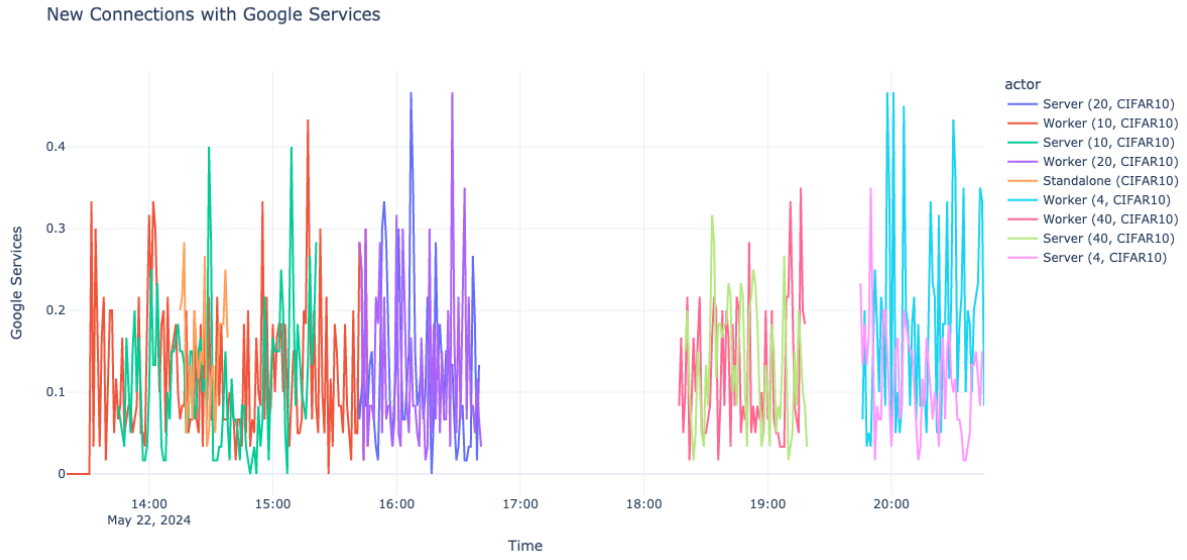

Figure F.8: Estimated number of distinct flows in a given minute. A flow is defined as a combination of (source IP, destination IP, source port, destination port, protocol). This estimate is based on sampled packets and may miss some flows. If a connection spans multiple minutes it can also be counted multiple times. Please note that this metric is currently in preview.

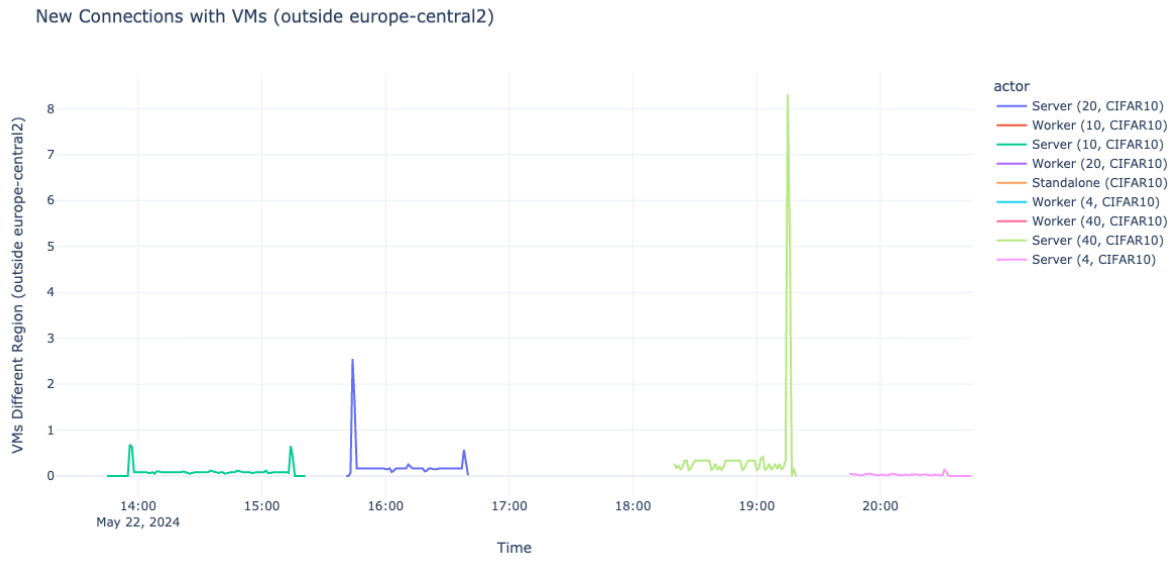

Figure F.9: Estimated number of distinct flows in a given minute. A flow is defined as a combination of (source IP, destination IP, source port, destination port, protocol). This estimate is based on sampled packets and may miss some flows. If a connection spans multiple minutes it can also be counted multiple times. Please note that this metric is currently in preview.

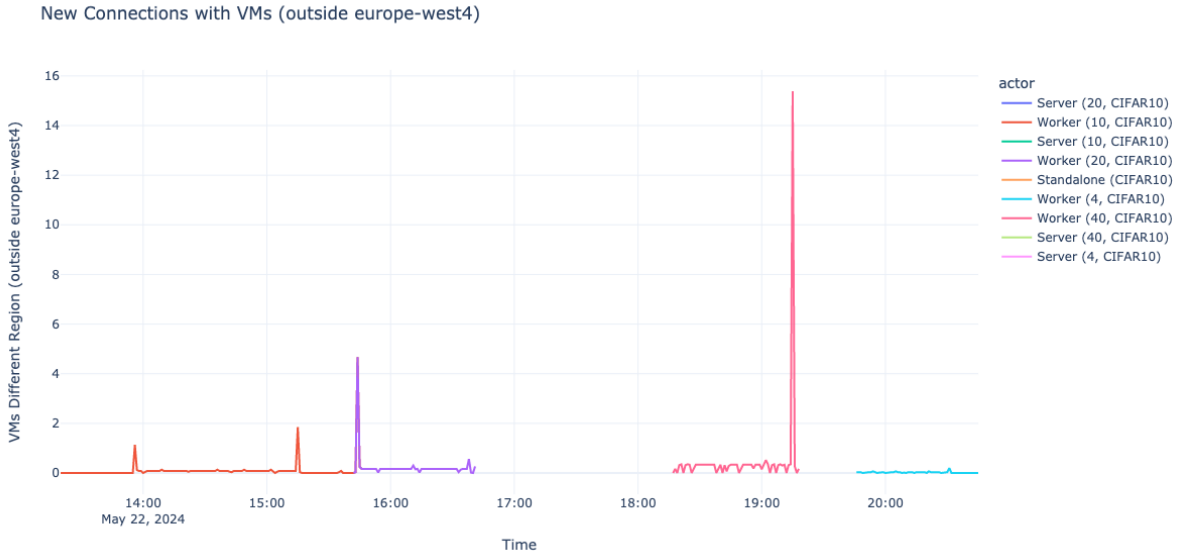

Figure F.10: Estimated number of distinct flows in a given minute. A flow is defined as a combination of (source IP, destination IP, source port, destination port, protocol). This estimate is based on sampled packets and may miss some flows. If a connection spans multiple minutes it can also be counted multiple times. Please note that this metric is currently in preview.

Bibliography

[Bou et al., 2023] Bou, A., Bettini, M., Dittert, S., Kumar, V., Sodhani, S., Yang, X., Fabritiis, G. D., and Moens, V. (2023). Torchrl: A data-driven decision-making library for pytorch.

[Brock et al., 2019] Brock, A., Donahue, J., and Simonyan, K. (2019). Large scale gan training for high fidelity natural image synthesis.

[Chen et al., 2022] Chen, J., Li, M., Liu, T., Zheng, H., Cheng, Y., and Lin, C. (2022). Rethinking the defense against free-rider attack from the perspective of model weight evolving frequency.

[Chen et al., 2018] Chen, P.-Y., Sharma, Y., Zhang, H., Yi, J., and Hsieh, C.-J. (2018). Ead: Elastic-net attacks to deep neural networks via adversarial examples.

[Fraboni et al., 2021] Fraboni, Y., Vidal, R., and Lorenzi, M. (2021). Free-rider attacks on model aggregation in federated learning.

[Goodfellow et al., 2014] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks.

[Goodfellow et al., 2015] Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples.

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

[Heusel et al., 2018] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2018). Gans trained by a two time-scale update rule converge to a local nash equilibrium.

[Huang et al., 2021] Huang, Y., Gupta, S., Song, Z., Li, K., and Arora, S. (2021). Evaluating gradient inversion attacks and defenses in federated learning.

[James et al., 2023] James, G., Witten, D., Hastie, T., Tibshirani, R., and Taylor, J. (2023). *An Introduction to Statistical Learning: With Applications in Python*. Springer International Publishing.

[Kahng et al., 2019] Kahng, M., Thorat, N., Chau, D. H. P., Viegas, F. B., and Wattenberg, M. (2019). Gan lab: Understanding complex deep generative models using interactive visual experimentation. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):310–320.

[Krizhevsky et al., 2009] Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar-10 (canadian institute for advanced research).

[Kurakin et al., 2017] Kurakin, A., Goodfellow, I., and Bengio, S. (2017). Adversarial examples in the physical world.

[LeCun et al., 2010] LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

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

[Moosavi-Dezfooli et al., 2016] Moosavi-Dezfooli, S.-M., Fawzi, A., and Frossard, P. (2016). Deepfool: a simple and accurate method to fool deep neural networks.

[Papernot et al., 2015] Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. (2015). The limitations of deep learning in adversarial settings.

[Rasouli et al., 2020] Rasouli, M., Sun, T., and Rajagopal, R. (2020). Fedgan: Federated generative adversarial networks for distributed data.

[Salimans et al., 2016] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans.

[Su et al., 2019] Su, J., Vargas, D. V., and Sakurai, K. (2019). One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841.

[Szegedy et al., 2014] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks.

[Wang et al., 2020] Wang, Y., Deng, J., Guo, D., Wang, C., Meng, X., Liu, H., Ding, C., and Rajasekaran, S. (2020). Sapag: A self-adaptive privacy attack from gradients.

[Winter et al., 2018] Winter, R., Montanari, F., Noé, F., and Clevert, D.-A. (2018). Learning continuous and data-driven molecular descriptors by translating equivalent chemical representations. *ChemRxiv*. This content is a preprint and has not been peer-reviewed.

[Xiao et al., 2018] Xiao, C., Zhu, J.-Y., Li, B., He, W., Liu, M., and Song, D. (2018). Spatially transformed adversarial examples.

[Xu et al., 2022] Xu, J., Hong, C., Huang, J., Chen, L. Y., and Decouchant, J. (2022). Agic: Approximate gradient inversion attack on federated learning. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 12–22.