Synthetic Data Generation for Storage Trace Augmentation

LU PANG and KRISHNA KANT, Temple University, USA

Due to the increasingly data-intensive nature of the applications, the storage system performance continues to increase in importance and is often substantially responsible for the overall processing rate of the application. Fortunately, the storage technologies themselves are evolving rapidly in numerous ways from low-level read/write of bits from a device, all the way to the management of the entire storage hierarchy in large enterprise and cloud settings. Studying many of the important issues in this entire spectrum often requires storage access traces from the storage server side, but these are often hard to come by. To address this gap, we present a method to generate synthetic traces that capture the realism and diversity of real data traces. The generated traces augment the existing workload traces of interest. These augmented traces can be used to test the storage system performance, train intelligent storage system models (e.g. intelligent tiering), or evaluate the longevity of storage devices. We demonstrate the versatility of our method generates traces that can replace existing traces in studying storage system aspects.

Additional Key Words and Phrases: Storage Traces, Generative Models, Synthetic Data, Time-series

ACM Reference Format:
Lu Pang and Krishna Kant. 2018. Synthetic Data Generation for Storage Trace Augmentation. 1, 1 (January 2018), 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The increasing data intensive nature of the data center applications coupled with the enormous complexity of modern storage systems, require that the storage systems be designed and tuned to be as performant as possible for the major applications of interest. Storage systems not only have numerous knobs to be tuned but also many dynamic data management policies that must work constantly to deliver the lowest latency and highest data volume. For example, it must dynamically decide the location of each type of data in the storage hierarchy and how/when it should be moved. Such aspects are typically evaluated using *storage traces* obtained from the real storage systems. Collecting such traces usually has a substantial performance impact on the storage systems since the desired metadata must be extracted on each access, stored in a buffer, and occasionally flushed to the disk. Such overhead, coupled with privacy concerns regarding access patterns, has resulted in only a small number of traces being made available publicly, and most of those tend to be of inadequate length. In contrast, many studies require quite long traces. Depending on the study objectives, the trace requirements may span from hours to weeks. For example, *tiering* is an essential feature for large scale storage systems, typically involving a slow but large HDD tier containing all the data, and a smaller/faster SSD tier for “hot" data. There may even be a third tier using emerging storage technologies for “very hot" data. Studying the performance of a tiering system adequately requires traces going over at least a few weeks to properly represent daily workload patterns. Similarly, backup/restore performance requires long traces.

Authors' address: Lu Pang, lpang@temple.edu; Krishna Kant, kkant@temple.edu, Temple University, 1925 N. 12th Street, Philadelphia, Pennsylvania, USA, 19122.

There are many tools to generate artificial storage or file system traces, but these are very limited in scope both in terms of the data they use and the type of accesses they generate. For example, FIO [3] and other similar tools can be used for standalone testing of a file system and its underlying storage systems. They initially create a whole bunch of files according to given parameters including the size distribution of files. Since the tool knows about these files, it can correctly create accesses to these files based on given parameters such as sequential/random access ratio, read/write ratio, access size taken from a specified distribution, etc. Clearly, such tools are very valuable for stress testing of storage systems in an artificial setting, but they do not generate realistic traffic or work with real data.

As has been clear from the previous discussion, our goal is to develop a generative model that is able to sample synthetic traces that have similar properties to real traces. The success of the method in generating synthetic data which has properties that are similar to real data can be measured by using it as a drop-in replacement for real data in data-based applications. In this paper, in addition to statistical evaluation methods, we also measure the quality of the trace generation by using a prediction model and evaluate the prediction of the synthetic traces versus that of the real traces. The model we use to evaluate the quality of the trace is one we have used to predict “heat" in our earlier work [29]. The notion of "heat" is used in storage systems to determine how actively the data is read or modified by the applications, as this has implications for many important operations such as tiering (i.e., movement to faster layers of the storage hierarchy), caching of data by the device, storage server, or the host, data prefetching, etc. The synthetic data will be used to train the model and will be evaluated on a real trace.

In this paper, we explore artificial traffic generation using Generative Adversarial Networks (GAN) [12]. There are various challenges when developing a synthetic data generation model that can generate realistic storage data. We will talk about the details of these challenges later in the paper. The key contribution of this paper is the design of a specialized GAN that can accurately generate spatio-temporal time-series corresponding to the storage traffic. Our design includes a mechanism to control the distribution of the generated traffic so that it is possible to match those approximately between the real and generated traffic. This also allows the generator to generate traffic with a different distribution without affecting other, more complex, aspects of the traffic. We also compare the generated trace with various evaluation methods, and show that it can accurately reproduce the general patterns of the real workload while still maintaining diversity.

The rest of the paper is organized as follows. Section 2 introduces the background related background work. Section 3 describes the challenges, data preparation, and essential details of our method. Section 4 analyses and evaluates the generated storage trace results. Finally, section 6 concludes the paper.

2 BACKGROUND

2.1 Emerging Storage Systems

Flexible and efficient storage systems form the backbone for working with big data and deriving intelligence from it. Storage systems continue to evolve rapidly along multiple aspects: underlying storage technologies, storage access protocols, storage management, and higher level abstractions such as file systems and how they map to the storage. In particular, while the traditional hard disk drive (HDD) technology continues to evolve rapidly to increase storage density [22, 24], it has also been largely augmented with the solid-state drives (SSDs) for primary storage (i.e., data that is actively used). The underlying NAND "flash" technology of SSD itself continues to evolve and provides a wide range of tradeoffs in terms of storage density, performance, cost, and endurance depending on the number of bits per basic "cell" of the technology. There are numerous other high performance emerging technologies that are beginning to fill the traditional large gap between storage and memory. Some examples of these technologies are Kioxia XL-flash, Intel Optane, Everspins's MRAM, etc. [10, 16, 17] These technologies naturally form a hierarchy, with HDD as the bottom layer (large capacity, slow, and cheap), SSD the middle layer (substantial capacity, fast, but more expensive), and emerging technology as the top layer (small, very fast and expensive). Often, the same basic technology may form multiple layers, such as slow vs. fast HDDs or SSDs. The entire hierarchy typically resides in a *storage server* which typically includes local DRAM cache and complex management to easily allocate requested space across one or more devices. Furthermore, storage servers invariably sit on the enterprise network, separate from the compute servers requesting storage services.

The basic unit of storage IO is typically a “block" of size 4KB, addressed using a sequential number called LBA (Logical Block Address), and much of the IO performance is usually quoted in terms of IOPS (IOs per second), meaning block transfers/sec. A typical HDD can provide $\approx 25K$ IOs/sec (IOPS) for sequential transfers and as low as 1/100th as much for random transfers, while the latencies can be as high as $\approx$ 5ms (depending on the seek and rotational latency components). HDDs achieve higher bandwidth by striping data across many drives. As a result, HDD "farms" in data centers provide a large amount of storage capacity but slow transfers. SSDs, in contrast, provide a latency in $\tilde{1}00\mu$s range and throughput of hundreds of K-IOPS to 1M or more IOPs, and no substantial difference between sequential and random IO). Higher speed technologies can support latencies under 10 μs and may support higher throughputs too. However, high speed technologies generally have a much higher cost, and thus much smaller installed amounts.

Most enterprise storage systems are huge, ranging from tens of terabytes to hundreds of petabytes. Thus while the users may still access storage in 4KB blocks, using such a small size for storage management and transfer operations is extremely inefficient. Thus the storage operations often use a much bigger unit, usually known as a *chunk*. Chunk sizes may range from hundreds of KBs to multiple MB sizes. We will also use the notion of chunks in this paper, typically defined as 8MB.

Since the storage servers are removed from the hosts that use them, the view of the storage from the storage server side is radically different from the one of a host. A host only sees the storage allocated to it, usually in terms of the file system residing on it, and has no idea where the storage comes from or whether it is spread over multiple storage devices. In contrast, a storage server works with the *storage volumes*, each of which could be carved out of one or more physical devices. Although a storage volume could be used by multiple applications, generally, the major enterprise applications use separate volumes. In this paper, we are only concerned with the storage server side view of things, and storage traces obtained thus is a *block trace*, i.e., a list of accesses to the blocks, stamped by the time, starting LBA, operation (read/write), transfer size, etc.

2.2 Data Augmentation

A method to increase the amount of data in a dataset is *data augmentation*, which has been a popular approach in machine learning and computer vision fields [4, 21, 36]. The new samples used to increase dataset size are generated from the existing data via a set of transformations, the appropriate transformations being application dependent. For example, in the case of images, it is easy to generate additional data by random cropping, translation, flipping, and adding some degree of noise. For time-series data, the suitable transformations are multiplicative scaling of signal amplitude (thereby making peaks more or less pronounced), stretching/compressing the time axis (to decrease/increase traffic density), scaling of mean and variance, and adding some random perturbation to it. This could be done in each dimension for a multi-dimensional time series.

Such operations create more diverse versions of the original data. If these variants are properly labeled, such data can be highly valuable in training machine learning models since it would force the model to learn more complex features and thereby avoid overfitting. However, creating additional data this way for driving a real system can be questionable; in particular, while highly diverse traffic episodes are good for stress testing, they are inappropriate for normal performance evaluation. At the same time, a highly constrained transformation may not introduce adequate diversity. In the GAN context, the two uses of transformation coincide – training the GAN on transformed data can improve the training but such training will also affect the generated traffic.

2.3 Generative Models

Generative models are statistical models that learn the data distribution and use it to generate samples of this distribution. This is in contrast with discriminative models, which are used to make a prediction based on the given samples.

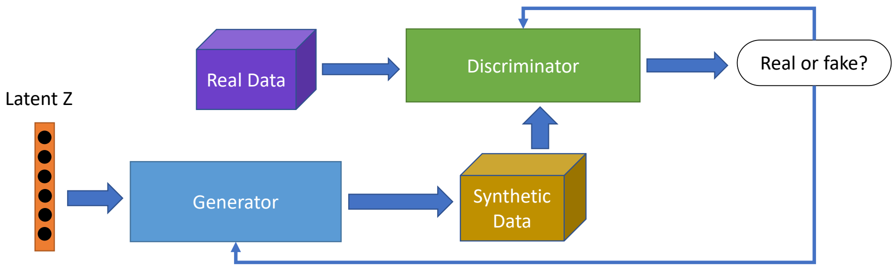


Fig. 1. Training Progresses of GAN

There are several methods that fall under the class of generative models. Variational Autoencoders (VAEs) are one such model. VAEs are autoencoders that consist of an encoder and a decoder. The encoder learns a Gaussian distribution that can be used to sample a latent vector. The decoder learns a Gaussian distribution representing the distribution of data of the sample from the latent vector. After the decoder is trained, we could generate samples from the learned distributions on demand. One could interpolate the output of the encoder to generate data that contains features of the input. However, the requirement of Gaussian distribution assumes smoothness that is usually not present in storage traffic traces. Diffusion models have emerged as a new class of generative models. Diffusion models gradually add noise to the original data samples through a Markov chain of diffusion steps and then reverse the process. Thus, the diffusion models are able to generate samples from the trained reverse diffusion process. However, these models tend to be rather slow and computationally expensive.

Generative adversarial networks (GANs) generate data via a transformation of a latent random vector into a data sample. The transformation is learned via co-training of the generator neural network and a discriminator network. These two networks play a game where the discriminator optimizes for discrimination between the generated synthetic data (or "fake" data) and the real training data. Figure 1 shows the general architecture of GANs. Given a Generative network G, Discriminator network D, distribution of original dataset P_{data}, distribution of the random noise vector P_z, this can be written more formally as a min-max equation [12] where $\min_G \max_D V(D, G)$ equals

$$E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(Z)))]$$

The first term encourages the discriminator to learn to maximize the value of real data and the second encourages the discriminator to provide a low value for the data generated by the generator. In the second term, the generator learns to generate data that reduces the value that the discriminator outputs for generated data.

Ideally, the generator and discriminator come to an equilibrium where both the discriminator and the generator can not do better. One danger of such a generative model is that the generator may get stuck into generating a random or specific pattern and be difficult to improve it further.

Several works in the computer vision domain show it is possible to generate realistic images and videos using the GAN model. And GANs have been used extensively and successfully in that role [2, 32, 35, 39]. Although GAN models have been recently used to generate a variety of time-series data, the existing work is rather limited. Recently, GANs have been used to generate network traffic [9, 34]. There are several works using the GAN model to generate continuous time-series data for medical and other types of signals. Esteban et al. [11] proposed a GAN framework for generating real-valued medical data sequences. They use recurrent neural networks to build both the generator and the discriminator. EEG-GAN [14] focuses on generating electroencephalographic (EEG) brain signals in a stable fashion. It does this by gradually relaxing the gradient constraint of the improved WGAN-GP training. TimeGAN [41] is a model that attempts to capture the temporal properties of time series data. It does this by including a supervised loss that encourages the model to capture the existing stepwise dependencies in the real data.

However, the generation of continuous and especially medical signals is different from generating storage accesses. In particular, in some medical domains, signals are expected to largely have approximately repeating characteristics, and while they can have abnormalities, those abnormalities are likely to be very infrequent. In contrast, there are no such requirements for the storage trace; instead, storage traces are characterized by almost random shifts in characteristics both over temporal and spatial domains. We are not aware of any work that attempts to generate complex storage traffic using GANs.

3 DATA GENERATION METHOD

3.1 Challenges in Storage Trace Generation Model

Using GANs to generate storage traffic is quite challenging as we need to capture complex correlations in time and space (block address). Unlike the generation of continuous time series like the medical signals, the successive storage requirements can access blocks that are very far from each other without obvious patterns. There are several other challenges as well, including the avoidance of mode collapse that is endemic to GANs. We also need to choose the GAN architecture in a way that the model can be trained well without itself requiring an enormous amount of data for training or requiring huge computing resources. There is also the issue of diversity in the generated traffic, On one hand, we do not want the GAN to simply duplicate the training traffic or even modify it in simple ways (e.g., scale its mean, variance, etc.), as that will not represent the kind of phase shifts one sees in the real storage traffic. On the other hand, we do want the generated traffic to be representative of the storage traffic trace. However, a precise definition of the desired diversity becomes quite difficult.

GANs when trained allow us to draw samples from a distribution. The problem is that when the dimension size of the samples is large, GANs become very difficult to train and require a huge amount of compute/memory resources. In the case of storage systems, the space dimension can easily range from millions to billions of chunks as mentioned above. Similarly, temporal variations in the traffic cover many time-scales, from milliseconds to weeks. It is not easy and requires sustained effort to generate realistic storage traffic while keeping both the data requirements for training the GAN and the training resources reasonable. This was essential not only for the feasibility of our current work but also to keep the mechanism practical in view of ever expanding scale and size of storage systems in the data centers. Towards this end, we have tried many different GAN architectures from the burgeoning GAN literature and their extensions.

To be able to train a GAN at these larger dimension sizes to sufficient quality, we examined several extensions that have been proposed in the literature. The two that we explored in detail are BigGAN [6] and Progressive Growing of GANs (PGGAN) [19]. We dismissed the former as the basis of our model because it requires high resources (in terms of computational power and dataset size) to train such a model, which makes it less practical. PGGAN instead progressively builds larger size samples till they reach the desired sample dimension size. Thus, we develop our data generation model based on PGGAN.

3.2 Data Preparation

We use several publicly available traces of enterprise storage servers. These generally provide 1-2 weeks' worth of block I/O traces, often taken from storage volumes dedicated to certain applications. (See section 4.1 for details). We preprocess the raw trace before we use it for our synthetic data generation model. The traces are data series of I/O accesses that contain the timestamp, offset, request size, and other metadata information. We then separate the trace into seven one-day long traces. We accumulate the data access frequencies into fairly large size access (or "chunk" access) within a certain timeslot t separately for read access and write access. We then represent the traces as data grids (successive time windows along x-axis, chunk addresses along the y-axis, and the value being the number of accesses). Figure 2 shows an example of read and write accesses of MSR dataset. The x-axis represents the successive timeslots, y-axis represents the chunk addresses, and the color represents the accesses number.

We then cut the read and write data grids into multiple data sample grids using the sliding window technique. Each data sample consists of m time slots. We concatenate the read and write data samples together on a new axis. The shape of each data sample is $2 \times n \times m$, where 2 is the axis that concatenates both read and write data samples and n is the total number of chunks on the server. We represent the dimension size of data samples as $n \times m$. The value of each element in the data samples varies a lot because of the large range of chunk addresses. Also, most of the values in the data samples are zero, since most chunks do not get accessed within m windows. This is typical of storage access; in fact, as the storage capacities and data set sizes increase, the actively used fraction is likely to go down. That is, the more extensive the value range of access frequency, the fewer the occurrence of high values. Therefore, we use a logarithmic scale to map the access frequencies to $[0, 255]$. We then normalize the data grids linearly to $[0, 1]$ in order to reduce the training time and make the GAN less likely to be trapped into a local optimum point. In addition, we perform downsampling of the data samples once before we train our data generation model so that we can use the downsampled versions for training directly rather than doing the downsampling each time on the fly. By downsampling the data samples, the size of the data sample is reduced. That is, the downsampling results in having less total number of chunks and time-slots, but the chunks and timeslots are larger. We denote the different sizes as $n_0 \times m_0, 2n_0 \times 2m_0, \ldots, 2^{k-1}n_0 \times 2^{k-1}m_0, n \times m$, where $n = 2^k n_0$ and $m = 2^k m_0$. $n_0 \times m_0$ is the initial size of the data sample to start the training of our synthetic data generation model. Note that since n_0 is only 2^{-k} times n, the chunk size at the lowest level is 2^k times the chunk/timeslot size used in our original $n \times m$ representation.

3.3 Model and Training

Our model is a modification of PGGAN. We use PGGAN mentioned earlier as the basis of our model to progressively scale the traces. It does this in layers. That is, we start with a GAN that learns to generate downsampled trace at the lowest level of $n_0 \times m_0$. Once that training phase is done, it adds

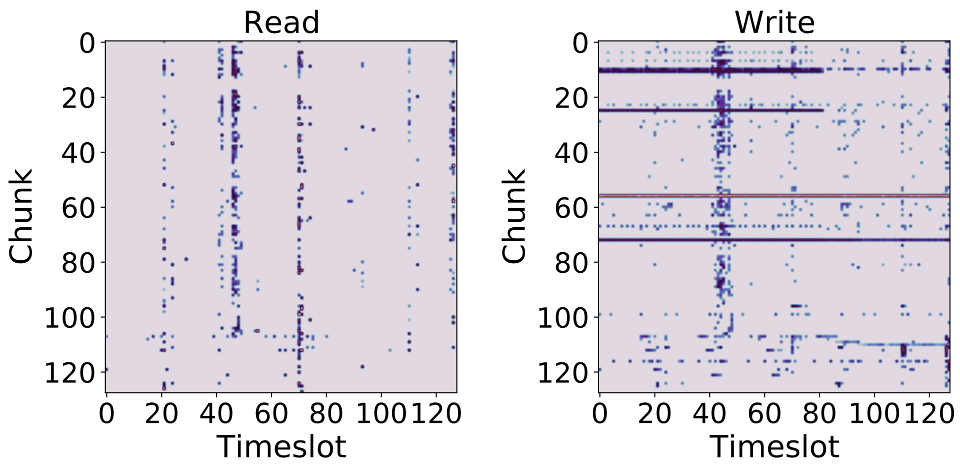


Fig. 2. Heatmap of read and write accesses of MSR (usr) workload. For clarity, only the first couple of chunks are shown.

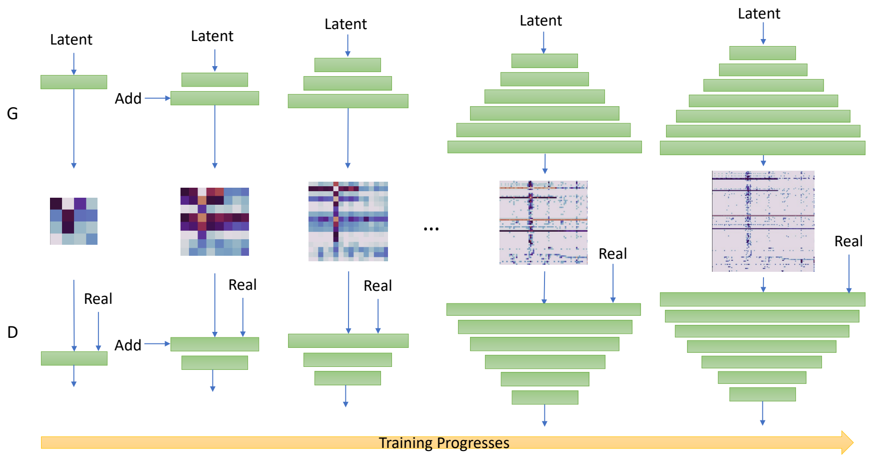


Fig. 3. Training Progresses of PGGAN

an extra layer to the model which is trained to generate a larger trace. It repeats this process until it reaches the desired trace sample dimension size. The training progresses of PGGAN is shown in Figure 3.

PGGAN connects the new layer to the upscaled trained output of the previous layer. Figure 4 illustrates the transition from data samples with a dimension size of $n_0 \times m_0$ to data samples with a size of $2n_0 \times 2m_0$. It slowly adjusts a parameter α to linearly mix the output of the upscaled previous layer with the output of the new layer. That is, the output of the upscaled previous layer is weighted by $(1 - \alpha)$, whereas the output of the new layer is weighted by α. Initially, α is set to 0 so that the generated data in training comes from the upscaled sample and as training continues, it

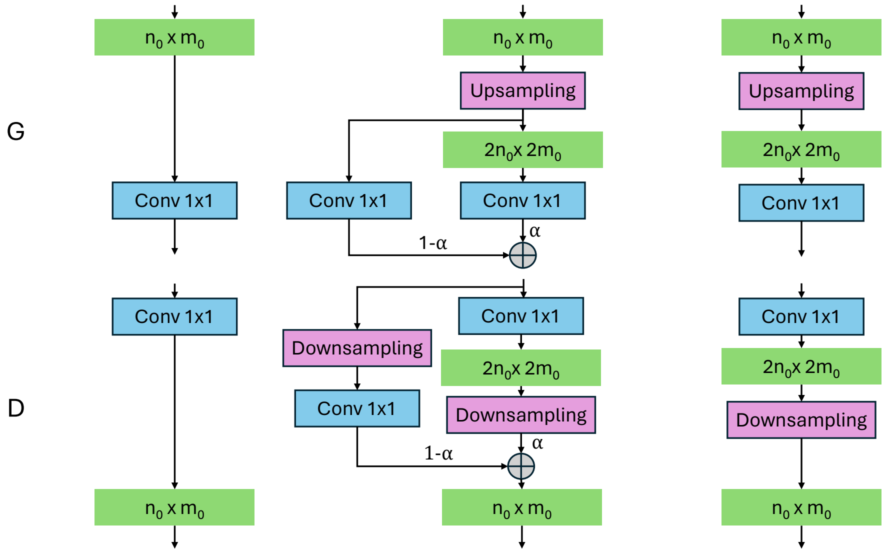


Fig. 4. Fading in the added new layers to G And D smoothly.

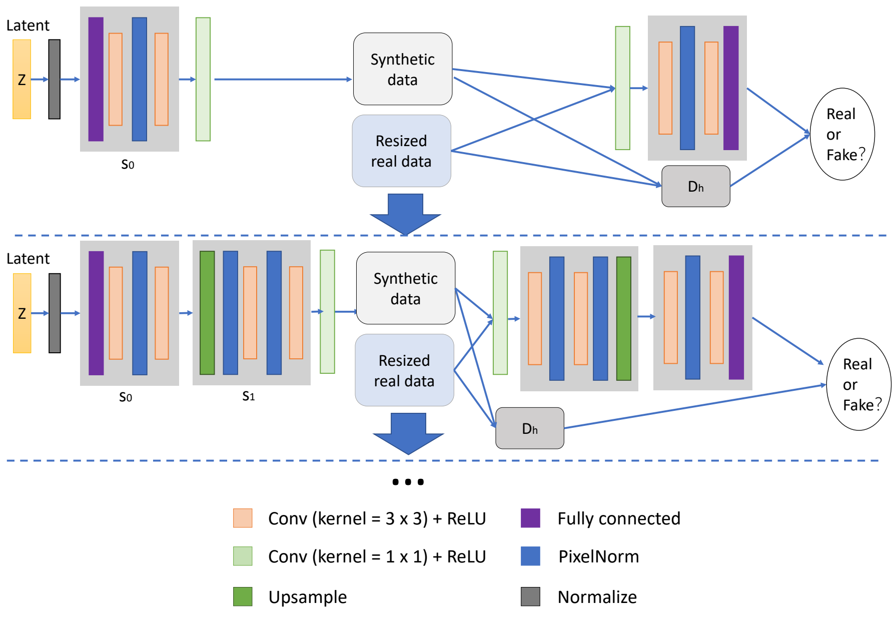


Fig. 5. Architecture of the synthetic storage trace generation model.

puts more emphasis on the new layer by increasing α linearly from 0 to 1. In this way, PGGAN makes use of the trained low layers to help optimize the parameters of the new layers.

Representation of the storage trace (i.e., # accesses to each chunk vs. time) may be visualized as an image, the considerations in generating images are very different than for generating realistic traces. In particular, with images, the key considerations are aesthetics and accurate rendering of objects. With storage traces, we are interested in statistical characteristics such as achieving a similar distribution of accesses in addition to capturing prominent patterns such as frequent chunk accesses. What they both share is the ability to ensure diversity in the generated data. Since our model operates on storage traces, it works on sparse data. That is, at any point in time, most of the chunks do not receive any access. In order to better capture the features of the sparse data, we add another discriminator to determine if the distribution of input data is real data or generated data.

Thus, our model consists of one generator network and two discriminator networks. One of the discriminators, denoted D, focuses on capturing the element-wise features. The other discriminator, D_{hist}, uses a histogram to capture the distribution features of the element values. A data sample x is input to D and D_{hist}, which output $D(x)$ and $D_{hist}(x)$ representing the probability of whether the input data is real or synthetic. The generator G uses an input noise vector z and generates the output $G(z)$. During training, the generator and the two discriminators are trained simultaneously.

As mentioned in the previous section, we obtain a dataset that consists of data samples of several different dimension sizes by performing downsampling. The original dimension size of each data sample is $n \times m$, where n is the total number of chunks on the server and m is the total time slots for each data sample. We start our training of the synthetic data generation model with $n_0 \times m_0$ and progressively scale to the original data sample size $n \times m$.

The input of the generator network is a random vector z drawn from a uniform distribution. The initial output of G is the generated data of size $n_0 \times m_0$. We input both the down-sampled version of real data samples of size $n_0 \times m_0$ and generated synthetic data of size $n_0 \times m_0$ to train the discriminators to first differentiate between the real and generated data of dimension size $n_0 \times m_0$. Then we add layers progressively to G and D to generate data of dimension size $n \times m$. We do not add layers to D_{hist} since we build the histogram with a constant number of bins that cover the range of $[0, 1]$. Instead, we reinitialize the weights of D_{hist}. As training progresses, We keep on adding layers to G and D as well as reset the weights of D_{hist} when we finish the training of the previous dimension size level and move to the next dimension size level. In the end, the generator and the two discriminators are trained to operate on data samples with dimension size $n \times m$.

The architecture of our synthetic data generation model is shown in Figure 5. The network is slowly growing as each layer is trained. The initial generator G consists of one Fully Connected (FC) layer, one two-dimensional (2-D) Convolutional Neural Network (CNN) layer with 3×3 kernel, and one 2-D CNN layer with 1×1 kernel. We first train the generator G to generate data of dimension size $n_1 \times m_1$ using initial input z. Then we add the replicated 3-layer blocks one by one to the G network thus increasing the dimension size of generated data in an incremental manner. The 3-layer block is made of one upsample layer and two 2-D CNN layers with 3×3 kernel. The components of discriminator network D are mirrored layers of G. We use average pooling in D to downsample the data size. The discriminator D_{hist} consists of one histogram construction unit, ten 1-D CNN layers, and one FC layer. The choice of CNN layers is based on their ability to find local correlation features. With deeper CNN layers, we are able to capture longer range correlations. We use the FC layer to map the learned features.

Our model has been trained with a dual-GPU machine with 11GB of GPU memory on the card and 64GB of main memory. In all three networks, G, D, and D_{hist}, every layer except the last uses the Leaky ReLU activation function [23]. The last layers of G and D use linear activation. The last layer of D_{hist} uses a tanh activation function [18]. We initialize the bias parameters to zero and set all the weights to follow a normal distribution. We then scale the weights with the per-layer normalization constant to ensure the equalized learning rate [19]. We also perform pixel-wise normalization in G and D [19]. We use Xavier uniform initialization [26] to initialize the weights in D_{hist} network in order to achieve convergence considerably faster. We also normalize the output of the histogram construction unit to stabilize the training. The average pooling layers of size 2×2 is used in discriminator D to bring down the feature dimension size to half of the previous layer. We use the nearest neighbor upsample to double the feature dimension size in G. The CNN layers have 16 channels. We optimize the synthetic data generation model using the Adam optimizer [20] with a learning rate of 0.001. We use a batch size of 4 to train the data generation model. Our training starts with the sample of dimension size 4×32 and generates the final samples of size 256×2048, where 256 represents the how many timeslots in one trace sample and 2048 represents the number of chunks in one trace sample. We use a timeslot size of 150 seconds in our training. The optimal values for these parameters may be different when using different datasets.

3.4 Loss Functions

The loss function provides the objective we wish to optimize. In the case of our model, this serves several purposes. First, it creates a competition between the generator and the discriminator. Second, it promotes the stability of the training. Finally, the choice of loss functions used for our model directs the network to train towards a certain behavior and induces capturing the features we deem appropriate for the task.

Binary cross-entropy loss: The goal of the discriminator is to distinguish between a real trace and a synthetic trace from the generator. We trained to output a high probability for real data and a low probability for generated data. Then the binary cross-entropy measures the difference between what probability the discriminator gives the data and the true value. This value is used to train the discriminator so that it has a high probability for real traces and low probability values for synthetic data. The weights in the generator are updated so that it encourages the discriminator to give high values for synthetic data.

Wasserstein GAN with Gradient Penalty (WGAN-GP): To encourage stability of training a GAN, WGAN [2] was developed. It uses the Wasserstein distance (also known as earth mover distance), as a measure between the generated probability distribution and real probability distribution. WGAN-GP [13] was then developed to add a gradient penalty term to the loss function in order to enforce the Lipschitz constraint on the discriminator. This ensures that the gradient is bounded and the training is well behaved.

Differential histogram: In addition to the previous losses, we have added a differential histogram [38] that is fed into its own discriminator, so as to capture the distribution of the accesses in the trace. The distribution in the traces is not Gaussian since the accesses are sparse and workloads in storage systems may have a multi-modal distribution.

4 EVALUATION AND DISCUSSION

In this section, we evaluate our synthetic storage trace generation method with various evaluation metrics. We use several publicly available block trace datasets in this evaluation and demonstrate that we can capture the essential characteristics of the trace in each case.

4.1 Publicly Available Traces and Characteristics

Our experiments are conducted with several block storage datasets coming from two sources. The first source is the Microsoft Research Cambridge (MSR) [27] dataset which contains one-week long block I/O trace of several different enterprise server workloads (using different storage volumes). We use the traces of the User home directories (*usr*), Hardware monitoring (*hm*), and Project directories (*proj*). Each trace also has the metadata information of the associated disk from which the traces are generated with. We select traces that are associated with one of the disks for our experiments.

The second data source is the Tencent production Cloud Block Storage (CBS) system [42]. The traces are collected from a production CBS system using a proxy server over 10 days. The proxy

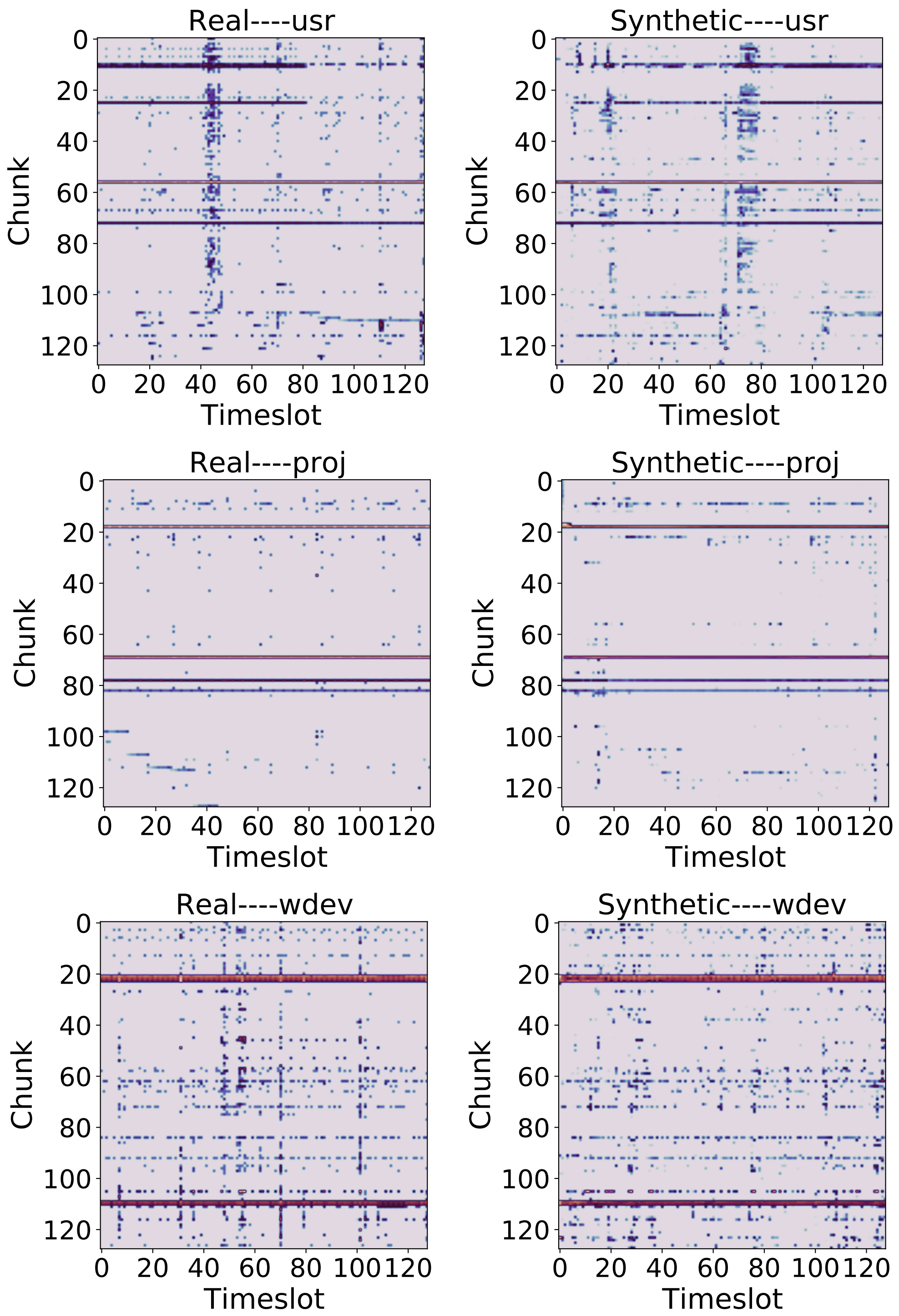


Fig. 6. Heatmap of real (left) and synthetic (right) write accesses of usr, proj, and wdev MSR workload. For clarity, only the first couple of chunks are shown.

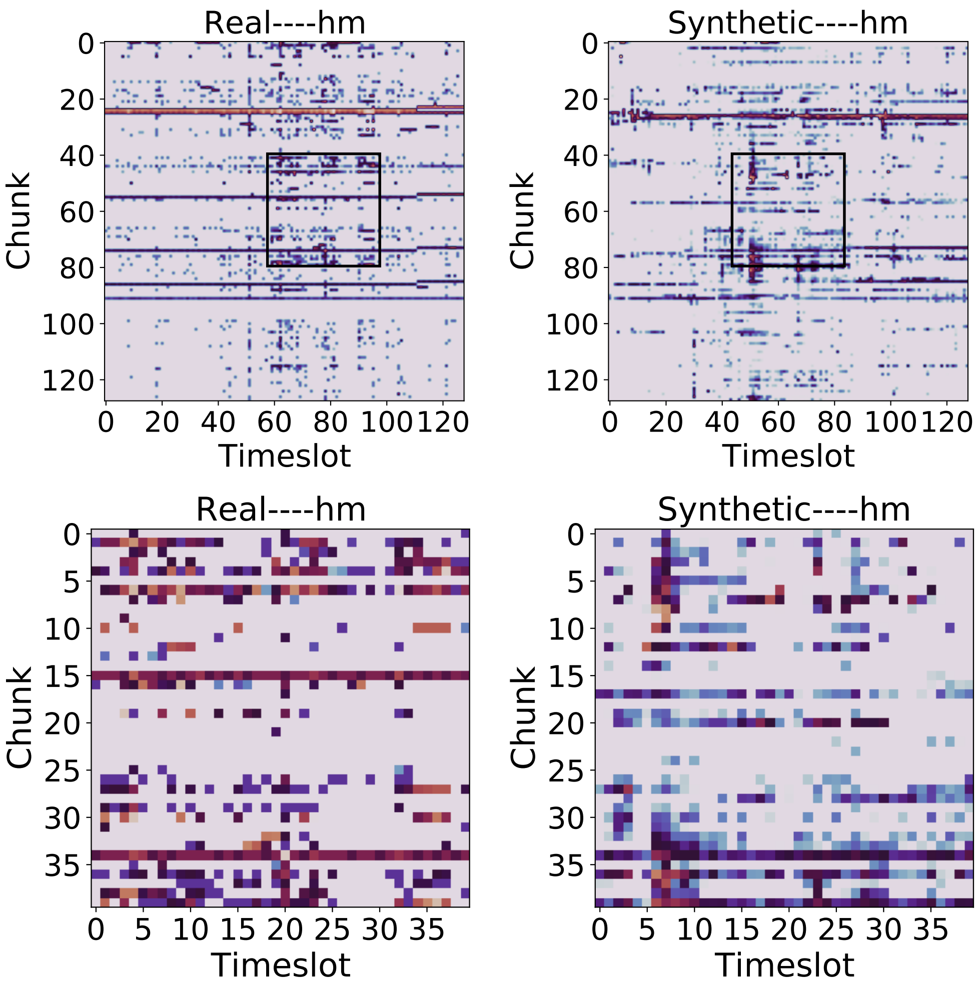


Fig. 7. Detail Heatmap of real(left) and synthetic (right) write accesses of hm MSR workload. For clarity, only the first couple of chunks are shown.

server is in charge of forwarding the I/O requests it receives from the client to the storage server. These highly dynamic traces are collected from multiple cloud virtual volumes (virtual disks). We evaluated several dozen of these traces and selected traces that represent the diversity of the dataset in the requested blocks and in the locality of requests. We randomly select two of such traces and call these two workloads *TC1* and *TC2*. We use the weekday traces of the first week in our evaluation.

4.2 Evaluation

Figs. 6-8 show the heatmap of synthetic traces and real traces for the MSR and Tencent CBS workloads. The generated traces show a similar behavior to the real traces. Also, when we take a closer look at both the generated and the real traces, for example, the detailed trace of hm workload is shown in figure 7, we notice that the generated traces are not duplicates of the original traces.

We start with a comparison of the statistics of the generated data compared to the real data. We use sixteen real trace samples and sixteen generated trace samples in our evaluation. The trace

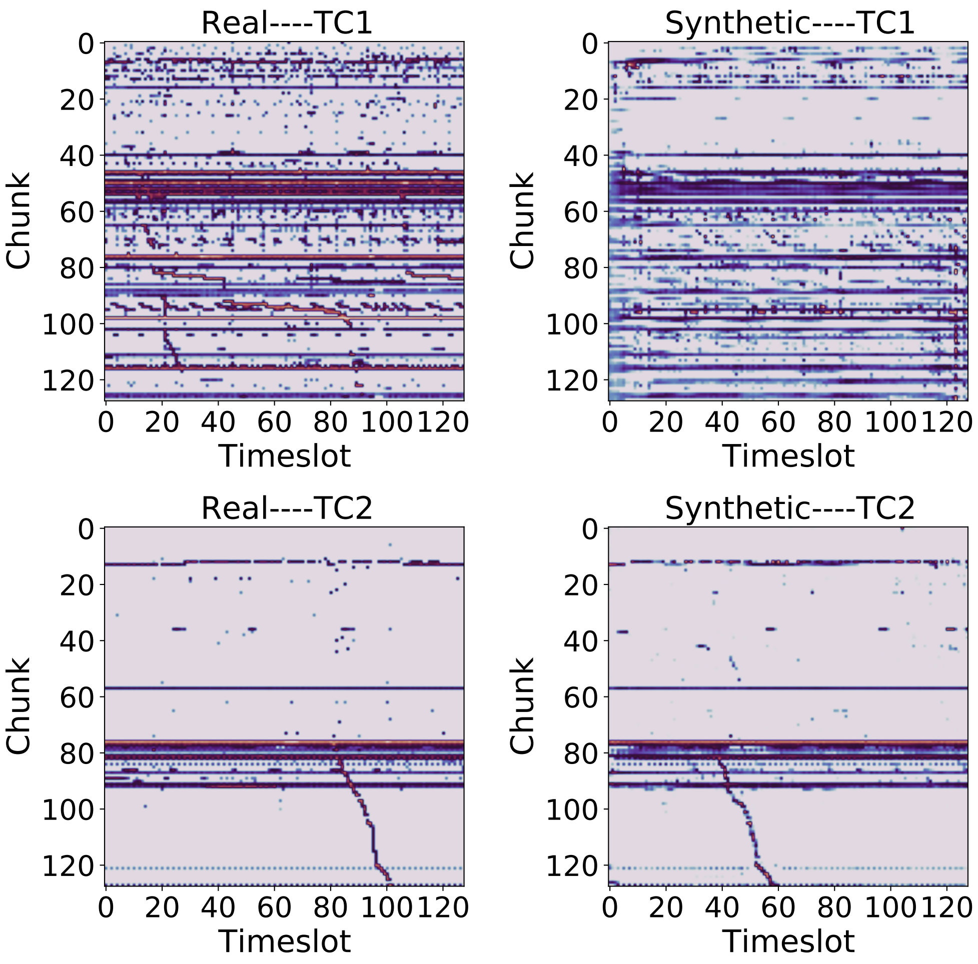


Fig. 8. Heatmap of real(left) and synthetic (right) write accesses of Tencent CBS workload. For clarity, only the first couple of chunks are shown.

Table 1. Mean and Standard Deviation (Real)

Day		**Workload**					
		usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Mean	***Max***	1.44	2.06	3.48	1.21	7.13	3.36
	Min	1.14	1.07	2.46	1.14	5.85	1.81
STD	***Max***	12.21	16.25	17.59	10.27	26.94	17.12
	Min	10.91	10.36	14.27	9.93	22.94	12.86

samples are randomly selected from the real and generated traces. The statistics are calculated for each of the sixteen trace samples. First, we compare both the mean and standard deviation (STD) of the real and generated traces. Thus, for each trace sample, we take the mean and standard deviation of the accesses over all the chunks and over the time range within the trace sample. The mean and standard deviation values of each sample may have different characteristics since the storage trace workload may have different patterns. Thus, we compare both the maximum and minimum values

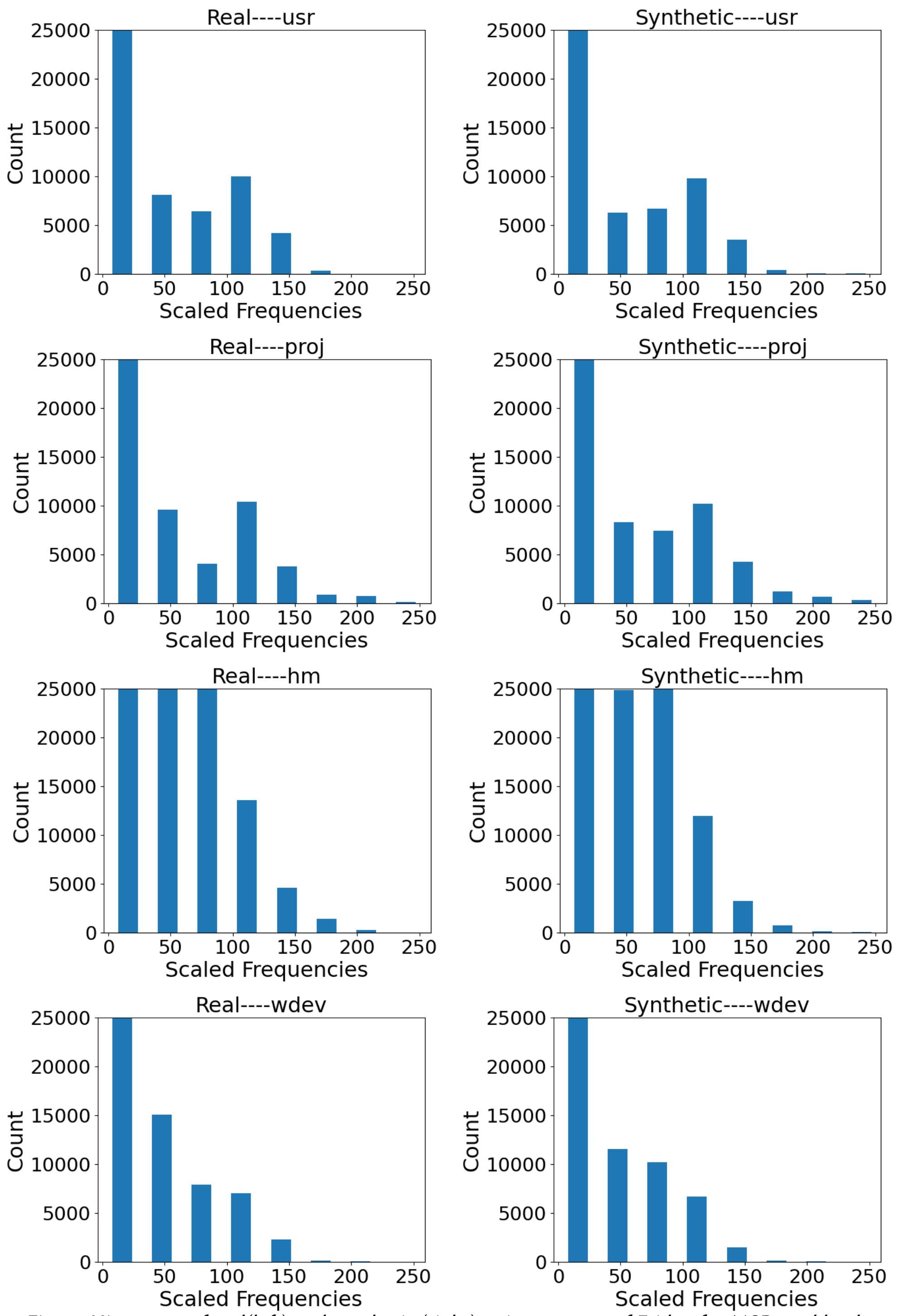


Fig. 9. Histogram of real(left) and synthetic (right) write accesses of Friday for MSR workload.

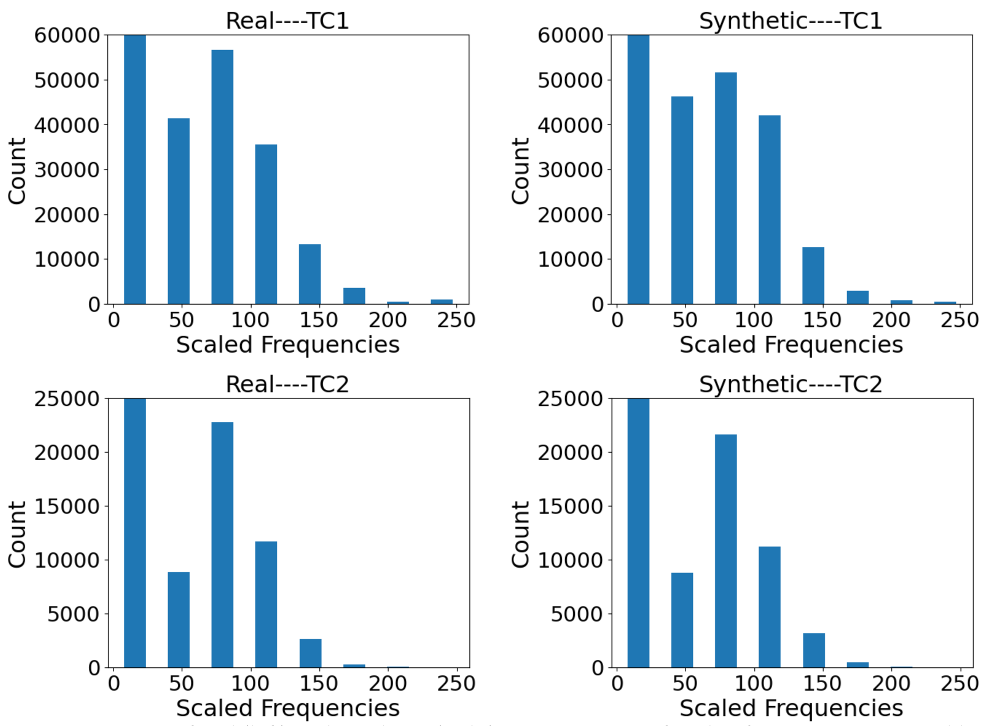


Fig. 10. Histogram of real (left) and synthetic (right) write accesses of Friday for Tencent CBS workload.

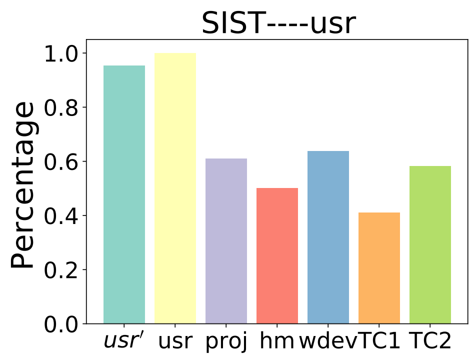
Table 2. Mean and Standard Deviation (Synthetic)

Day		**Workload**					
		usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Mean	***Max***	1.39	2.01	3.40	1.21	7.46	3.59
	Min	1.07	1.05	2.58	1.09	6.30	1.82
STD	***Max***	11.61	16.00	16.62	10.04	26.10	17.43
	Min	10.38	10.11	14.32	9.53	23.44	12.75

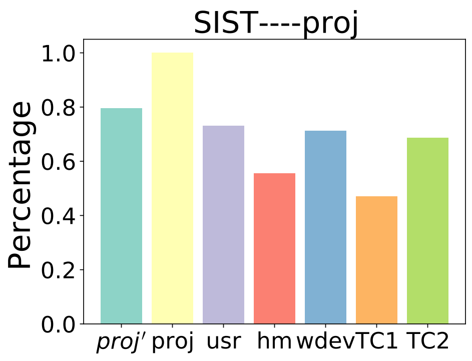
of these samples. As we can see from tables 1 and 2, for both MSR trace and Tencent CBS trace, the max/min mean and the STD values of the generated trace samples are very close to the max/min mean and the STD values of the real traces. The reason that the mean is a low value is that the accesses in the trace across time and chunks are very sparse. Generating overlapping means is in fact non-trivial because the tail distributions of the accesses are long.

For the reason stated previously, synthetic storage traces should also capture the distribution of the workload being targeted. Fig. 9 shows the histogram of the accesses for the MSR write workload and Fig. 10 shows the histogram of the accesses for the Tencent CBS write workload. The closeness of the distribution shape between the real and synthetic traces validates the inclusion of the differential histogram loss as a way to control the distribution of accesses.

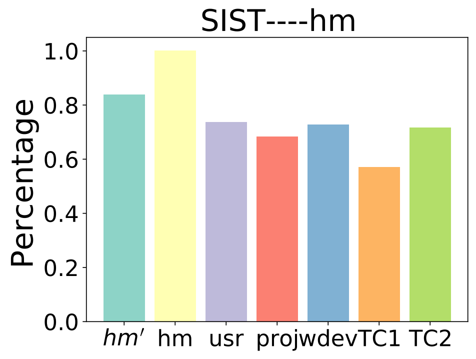
To evaluate the behavior of the generated workload compared to the real data, we use a storage trace similarity metric that we have introduced in our prior work [30]. We call this as Similarity Index for Storage Traffic (SIST). SIST is a similarity metric that is more directly related to storage access issues and performs better compared to other commonly used similarity measures when dealing with storage traces.



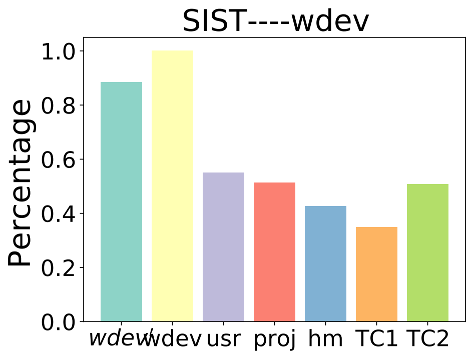
(a)



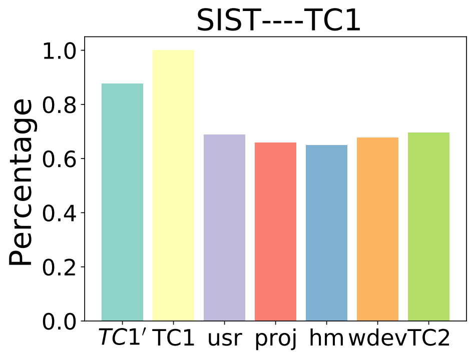
(b)



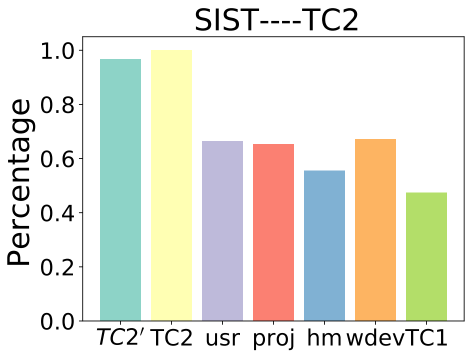
(c)



(d)



(e)



(f)

Fig. 11. Histogram of SIST for MSR and Tencent CBS workloads.

Fig. 11 shows 6 charts, one for each workload as indicated. Each chart shows different SIST measures of one workload. These SIST measures include the SIST measure between the original trace samples and the synthetic samples of that workload, the SIST measure between the original trace samples of that workload, and the SIST measure of the original samples of that workload against original samples of other workloads. For example, in Fig. 11(a), we show the SIST measure between the original usr trace samples (marked as usr), and compare it against the SIST measure of

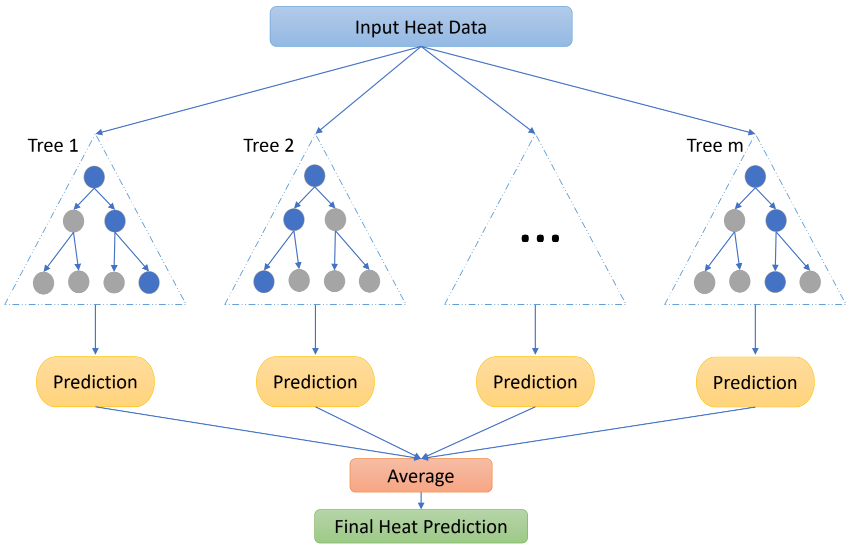


Fig. 12. Architecture of the heat prediction model.

generated and original usr trace samples (marked as usr'), and also show the SIST measure of the original usr trace samples and the original proj, hm, wdev, TC1 and TC2 trace samples. It is seen that SIST measure of usr' is closest to that of usr. The same behavior holds in Figs. 11(b)-(f). For example, in Fig. 11(f) TC2' is closest to TC2 as compared to others.

To further evaluate the behavior of the generated workload compared to the real data, we evaluate the synthetic trace on a prediction model. The model is a simplified version of a heat prediction model that we developed for storage traces [29]. The aspect of the model that we use in our evaluation is the heat prediction of the next time step based on the current activity. The heat prediction model predicts how many accesses each of the chucks gets for the next time step.

Table 3. Active Hit Rate

Day	**Workload**					
	usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Real	0.77	0.72	0.56	0.73	0.66	0.71
Synthetic	0.77	0.73	0.62	0.73	0.67	0.70

Fig. 12 shows the architecture of our heat prediction model. We use the Random Forest (RF) [5] algorithm to generate the prediction model. The prediction model is composed of a collection of many Decision Tree (DT) [31] regression models. We use this collection of DT models to train over the input data and average the results of each model to produce the final heat prediction. We run the prediction model with the mixed training dataset in that one out of five weekday data are randomly selected and replaced by the generated data. As we can see from table 3, we achieve similar results in heat prediction evaluation.

5 RELATED WORK

Synthetic data generation and data augmentation using generative models have a huge amount of interest in the research community. The works range from methods that allow the choosing of classes for the outputs to augmentation by transforming the current dataset. The emphasis of the synthetic data generation and data augmentation works shown below differ from the goal that our work tries to accomplish.

Data Augmentation Generative Adversarial Networks (DAGAN) [1] is a model designed to generate class-specific data. It does this by learning the class parameters as part of its generator. This allows it to use few-shot learning and generate unique data from the distribution. DAGAN infers the class information from the example input and is able to generate output similar to the class.

Trabucco et. al. [37] studies diffusion models [15] to generate synthetic data to augment real data for downstream tasks. Their aim is to ensure the diversity of the generated data. For example, they noted the issue of generating different species of animals. They propose an image-to-image transformation parameterized by pre-trained text-to-image diffusion models.

T-CGAN [33] uses the Conditional GAN (CGAN) [25] model to generate irregularly sampled time series data for data augmentation. CGAN introduces a way to condition the generator and discriminator by feeding the extra information, such as class labels or other domain knowledge, to both the discriminator and generator. CGAN is trained in this way so that data for specific classes can be generated on demand. T-CGAN generates time series, that are irregularly sampled, by conditioning the generator and discriminator with the timestamps of the time series.

Auxiliary Classifier GAN (ACGAN) [28] extends and improves on CGAN by introducing an auxiliary classifier to predict the class label of the generated data. Chen et. al. [7] adopt a data augmentation scheme based on ACGAN to directly generate different features of the desired acoustic scene with input scene conditions.

InfoGAN [8] learns interpretable representations to generate synthetic data with specific characteristics. InfoGAN introduces a classifier to maximize the mutual information between conditional variables and the generated data. This allows it to associate the conditional variable with the characteristics of the generated data. It does this in a completely unsupervised manner. This means that some features of the generated data can be controlled by changing a latent vector. But what features the individual element of the factor corresponds to are not known a priori. Wan et. al.[40] propose an InfoGAN-based model to learn the coupling relations among bridge monitoring factors and then generate synthetic bridge monitoring data with various characteristics to augment the existing monitoring data.

6 CONCLUSIONS AND FUTURE WORK

Traditionally, in machine learning, data augmentation has been performed for computer vision applications to improve the performance of the models. Storage traces present a unique challenge to augmentation including the fact that storage traces are sparse and rather irregular, so the usual methods used in computer vision applications do not apply. We discussed how synthetic data can be used to design and evaluate storage systems when access to real traces is not available.

Our model is based on PGGAN so that we are able to grow the traces to dimension sizes that are usable in storage applications. We add a differential histogram to capture not only the point statistics but also the distribution of accesses. We show that indeed the model is able to capture the essence of the trace. We also show that, through evaluating the generated data on previous prediction models, the synthetic data provides an alternative to evaluating storage systems when we are not able to use a real workload.

One potentially useful aspect in generating storage traces is its specialization to certain categories such as heavy traffic, highly variable traffic, etc. We have not pursued this angle in this paper. Some simple post-processing techniques (e.g., those that scale the mean or variance or cause other systematic perturbations to the traffic) can accomplish such tasks without disrupting the correlation structure of the time series. Nevertheless, it is possible to train a generator that explicitly takes a class designator as input and generates traffic according to those characteristics. Many image GAN models have explored the generation of class-specific images such as the CGAN, ACGAN, and InfoGAN models discussed in the Related Work section. These methods could also be adapted to generate class-specific storage traces.

In the future, we plan to extend our technique to other spatio-temporal data such as data concerning vegetation, spread of tree and crop pathogens, or spatio-temporal variations in various socio-economic factors.

ACKNOWLEDGMENTS

We are grateful to Dr. Jeremy Swift from Dell Corporation for a long-term collaboration on this work. His expertise and insights into the enterprise storage systems, gained from his 20+ of work at Dell, were immensely valuable in ensuring that this work stayed in tune with the needs of designing and improving real enterprise storage systems. The initial part of this work was also funded by Dell, and are grateful for the funding.

REFERENCES

[1] Anthreas Antoniou, Amos Storkey, and Harrison Edwards. 2018. Data Augmentation Generative Adversarial Networks. https://openreview.net/forum?id=S1Auv-WRZ

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein Generative Adversarial Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 214–223.

[3] Jens Axboe. 2022. Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[4] Philip Bachman, R Devon Hjelm, and William Buchwalter. 2019. Learning Representations by Maximizing Mutual Information Across Views. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/ddf354219aac374f1d40b7e760ee5bb7-Paper.pdf

[5] Leo Breiman. 2001. Random Forests. *Machine Learning* (2001). https://doi.org/10.1023/A:1010933404324

[6] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2019. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations*.

[7] Hangting Chen, Zuozhen Liu, Zongming Liu, and Pengyuan Zhang. 2020. ACGAN-based data augmentation integrated with long-term scalogram for acoustic scene classification. *arXiv preprint arXiv:2005.13146* (2020).

[8] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *Advances in neural information processing systems* 29 (2016).

[9] Adriel Cheng. 2019. PAC-GAN: Packet generation of network traffic using generative adversarial networks. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 0728–0734.

[10] Intel Corporation. 2023. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html.

[11] Cristóbal Esteban, Stephanie L Hyland, and Gunnar Rätsch. 2017. Real-valued (medical) time series generation with recurrent conditional gans. *arXiv preprint arXiv:1706.02633* (2017).

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[13] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. 2017. Improved Training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf

[14] Kay Gregor Hartmann, Robin Tibor Schirrmeister, and Tonio Ball. 2018. EEG-GAN: Generative adversarial networks for electroencephalograhic (EEG) brain signals. *arXiv preprint arXiv:1806.01875* (2018).

[15] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 6840–6851. https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf

[16] Everspin Technologies Inc. 2023. Storage Solutions. https://www.everspin.com/storage-solutions.

[17] KIOXIA America Inc. 2023. XL-FLASH | Storage Class Memory (SCM). https://americas.kioxia.com/en-us/business/memory/xlflash.html.

[18] Barry L Kalman and Stan C Kwasny. 1992. Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, Vol. 4. IEEE, 578–581.

[19] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *International Conference on Learning Representations*.

[20] Diederik P. Kingma et al. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[22] Seagate Technology LLC. 2023. Everything You Want to Know About Hard Drives. https://www.seagate.com/blog/everything-you-wanted-to-know-about-hard-drives-master-dm/.

[23] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. Atlanta, Georgia, USA, 3.

[24] Inc. Micron Technology. 2023. What is a hard disk drive (HDD)? https://www.crucial.com/articles/pc-builders/what-is-a-hard-drive.

[25] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. *CoRR* abs/1411.1784 (2014). arXiv:1411.1784 http://arxiv.org/abs/1411.1784

[26] Dmytro Mishkin and Jiri Matas. 2015. All you need is a good init. *CoRR* (2015).

[27] Dushyanth Narayanan et al. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*.

[28] Augustus Odena, Christopher Olah, and Jonathon Shlens. 2017. Conditional Image Synthesis with Auxiliary Classifier GANs. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 2642–2651. https://proceedings.mlr.press/v70/odena17a.html

[29] Lu Pang et al. 2019. Data Heat Prediction in Storage Systems Using Behavior Specific Prediction Models. In *38th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE.

[30] Lu Pang and Krishna Kant. 2022. SIST: A Similarity Index for Storage Traffic. *Proc. of NAS confernce* (Oct 2022).

[31] J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning* (1986). https://doi.org/10.1007/BF00116251

[32] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[33] Giorgia Ramponi, Pavlos Protopapas, Marco Brambilla, and Ryan Janssen. 2018. T-cgan: Conditional generative adversarial network for data augmentation in noisy time series with irregular sampling. *arXiv preprint arXiv:1811.08295* (2018).

[34] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. 2019. Flow-based network traffic generation using generative adversarial networks. *Computers & Security* 82 (2019), 156–172.

[35] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. 2016. Improved Techniques for Training GANs. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf

[36] Patrice Y Simard, David Steinkraus, John C Platt, et al. 2003. Best practices for convolutional neural networks applied to visual document analysis.. In *Icdar*, Vol. 3. Edinburgh.

[37] Brandon Trabucco, Kyle Doherty, Max Gurinas, and Ruslan Salakhutdinov. [n. d.]. Effective Data Augmentation With Diffusion Models. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

[38] Evgeniya Ustinova and Victor Lempitsky. 2016. Learning Deep Embeddings with Histogram Loss. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4177–4185.

[39] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. 2016. Generating videos with scene dynamics. *Advances in neural information processing systems* 29 (2016).

[40] Ping Wan, Hongli He, Ling Guo, Jiancheng Yang, and Jie Li. 2021. InfoGAN-MSF: a data augmentation approach for correlative bridge monitoring factors. *Measurement Science and Technology* 32, 11 (2021), 114008.

[41] Jinsung Yoon, Daniel Jarrett, and Mihaela Van der Schaar. 2019. Time-series generative adversarial networks (TimeGAN). *Advances in neural information processing systems* 32 (2019).

[42] Yu Zhang et al. [n. d.]. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *USENIX ATC (2020)*.

Received 20 February 2023; revised 12 March 2023; accepted 5 June 2023