PLOS ONE

RESEARCH ARTICLE

Learned pseudo-random number generator: WGAN-GP for generating statistically robust random numbers

Kiyoshiro Okada[1,3], **Katsuhiro Endo**[1], **Kenji Yasuoka**[1], **Shuichi Kurabayashi**[2,3]*

1 Department of Mechanical Engineering, Keio University, Yokohama, Japan, **2** Graduate School of Media and Governance, Keio University, Fujisawa, Kanagawa, Japan, **3** Cygames Research, Cygames Inc., Shibuya, Tokyo, Japan

* kurabayashi_shuichi@cygames.co.jp

Citation: Okada K, Endo K, Yasuoka K, Kurabayashi S (2023) Learned pseudo-random number generator: WGAN-GP for generating statistically robust random numbers. PLoS ONE 18(6): e0287025. https://doi.org/10.1371/journal.pone.0287025

Editor: Sheetal Kalyani, IIT Madras, INDIA

Received: August 30, 2022

Accepted: May 30, 2023

Published: June 14, 2023

Peer Review History: PLOS recognizes the benefits of transparency in the peer review process; therefore, we enable the publication of all of the content of peer review and author responses alongside final, published articles. The editorial history of this article is available here: https://doi.org/10.1371/journal.pone.0287025

Data Availability Statement: All relevant data are within the paper.

Funding: The authors received no specific funding for this work.

Abstract

Pseudo-random number generators (PRNGs) are software algorithms generating a sequence of numbers approximating the properties of random numbers. They are critical components in many information systems that require unpredictable and nonarbitrary behaviors, such as parameter configuration in machine learning, gaming, cryptography, and simulation. A PRNG is commonly validated through a statistical test suite, such as NIST SP 800-22rev1a (NIST test suite), to evaluate its robustness and the randomness of the numbers. In this paper, we propose a Wasserstein distance-based generative adversarial network (WGAN) approach to generating PRNGs that fully satisfy the NIST test suite. In this approach, the existing Mersenne Twister (MT) PRNG is learned without implementing any mathematical programming code. We remove the dropout layers from the conventional WGAN network to learn random numbers distributed in the entire feature space because the nearly infinite amount of data can suppress the overfitting problems that occur without dropout layers. We conduct experimental studies to evaluate our learned pseudo-random number generator (LPRNG) by adopting cosine-function-based numbers with poor random number properties according to the NIST test suite as seed numbers. The experimental results show that our LPRNG successfully converted the sequence of seed numbers to random numbers that fully satisfy the NIST test suite. This study opens the way for the "democratization" of PRNGs through the end-to-end learning of conventional PRNGs, which means that PRNGs can be generated without deep mathematical know-how. Such tailor-made PRNGs will effectively enhance the unpredictability and nonarbitrariness of a wide range of information systems, even if the seed numbers can be revealed by reverse engineering. The experimental results also show that overfitting was observed after about 450,000 trials of learning, suggesting that there is an upper limit to the number of learning counts for a fixed-size neural network, even when learning with unlimited data.

Competing interests: The authors have declared that no competing interests exist.

Introduction

With the increasingly unpredictable and nonarbitrary behaviors required by information systems in various fields, random numbers have played a crucial role in implementing unpredictable and dynamic behaviors [1]. For example, cryptography [2, 3], gaming [4], machine learning [5, 6], and a wide range of simulations such as molecular simulation [7–11] and phase field simulation [12, 13] have utilized random numbers to implement unpredictable and nonarbitrary behaviors. Random numbers are a fundamental tool for implementing fairness and an essential software component for implementation. There are two common tools used to generate random numbers: true random number generators (TRNGs) utilizing physical phenomena and pseudo-random number generators (PRNGs) implemented as software algorithms. TRNGs adopt physical phenomena with randomnesses, such as temporal properties of operating system user processes, thermal noise, shot noise, electronics noise, and the emission timing of radioactive decay, to generate random numbers. Many researchers have developed well-known TRNG implementations, starting with low speed rates up to 300 Gb/s random bit generation (RBG) [14–16], and high speed rates up to 2 Tb/s RBG [17–28], and the fastest one of 250 Tb/s RBG was developed by Kim et al. [29]. Such TRNGs are often utilized in high-risk domains where genuine unpredictability is required, including security and finance. The major disadvantage of TRNGs is the long time required to generate many random numbers compared with PRNGs, which is due to their dependence on physical phenomena and the need for specific hardware. PRNGs have been developed as general-purpose software modules for many years and have been widely adopted in domains where a vast array of random numbers are required in information systems, such as the security field. PRNGs are algorithms that rapidly generate uncorrelated and random sequences of numbers that appear to be sufficiently complex and random for ordinary purposes. A PRNG is validated through a statistical test suite, such as NIST SP 800-22rev1a (NIST test suite) [30, 31], which is commonly used to evaluate the robustness and fairness of the generated random numbers and consists of 15 statistical tests. PRNG algorithms, such as Mersenne Twister (MT) [32] and Xorshift [33], and cryptographic PRNGs are deterministic, producing the same sequence of numbers with a unique period when the same initial values, called "seeds", are input.

Because many PRNG algorithms are open and publicly available, applying PRNGs to online applications and services requires extensive care to hide the seeds and implementation code, thus preventing malicious users from predicting the next number to be generated. It is straightforward for well-trained engineers to predict the periodic behavior of a PRNG if they know the seed number and the algorithm through reverse engineering. Although many application developers adopt the encryption of seeds and code obfuscation as good practice when utilizing a PRNG, it is difficult to completely hide implementation details in modern smartphone applications. There is an urgent need to invent a new random number generation model to allow application software developers without advanced mathematical skills to create tailor-made PRNGs whose behavior is publicly unpredictable and fully satisfies the NIST test suite.

Machine learning (ML) technologies can be used to realize artificial intelligence for studying algorithms and statistical models that allow computer systems to perform tasks without explicit instructions. These ML technologies have been considered beneficial for creating new PRNGs. The statistical models currently widely used are a type of function approximator called neural networks. With the successful adoption of deep neural networks (DNNs) with multiple hidden layers in many fields [34], many researchers have attempted to use them to create PRNGs by approximating the mathematical functions of existing PRNGs. For example, the Elman neural network [35], recurrent layers network [36, 37], recurrent neural networks with

long short-term memory units (LSTMs) [38, 39], and Hopfield neural network [40–42] are well-known recurrent neural network models that have been applied to generate PRNGs. Over the last decade, PRNG methods using new deep learning models, such as reinforcement learning [39, 43] and generative adversarial networks (GANs) [44, 45], have emerged and attracted the attention of researchers.

Given the availability of many ML technologies, a recent key question has been how to create a new PRNG by "end-to-end" learning [44], where ML technologies directly learn the behavior of the existing PRNGs without any preprocesses or post-processes. End-to-end deep learning can replace a complex ML system that involves multiple stages of processing from the initial data input to the final result output with a single large neural network with multiple layers and modules that perform various processes. Such end-to-end and fully automatic generation of PRNGs will enable ordinary application developers to create their own PRNGs customized to their applications and systems. As examples of end-to-end neural network PRNGs, De Bernardi et al. [44] and Oak et al. [45] have applied GANs [46] to learn the behavior of existing PRNGs that allow end-to-end random number generation. However, these studies [44, 45] did not adopt the recommended NIST test suite configurations involving 1,000 test runs (e.g., 10 test runs in the previous studies). To our knowledge, there are no ML-based or DNN-based PRNGs that fully satisfy the NIST test suite under the recommended evaluation criteria. A novel method to realize a learned random number generator that fully satisfies the officially recommended settings of the NIST test suite is desired to achieve end-to-end learning for PRNGs having production-level credibility.

Thus, we propose a new end-to-end random number generation model rather than just brushing up on previous research. We call it as the learned PRNG (LPRNG), which adopts a GAN with the Wasserstein distance (WGAN [47]), which is already known as a sophisticated version of GAN. This model learns from inexhaustible random numbers obtained from MT [32], a well-known PRNG algorithm, and generates practical random numbers that pass all the tests in the NIST test suite under the NIST recommended settings, including over 1,000 test runs. To our knowledge, this is the first automatically generated PRNG having the sufficient quality to be deployed in production, owing to it fully satisfying the NIST test suite. Our key modification of the WGAN is to remove the dropout layers, which markedly improved the generation of random numbers. Fig 1 shows a schematic overview of this study. To verify the robustness of this approach, we conducted evaluations that adopt cosine function-based seed values with poor random number properties to avoid information leakage in the learning phase. We also confirmed that these seed values have poor randomness using the NIST test suite. We sequentially observed the learning processes of the LPRNG by which its capability to transform an input seed into a random number with better randomness is improved as the learning progresses up to 900,000 iterations. The experimental results show that our model can learn "randomness" from the nearly infinite amount of training data without early overfitting problems, despite our model not including dropout layers. Interestingly, we found that overfitting occurs after about 450,000 iterations even with an infinite amount of training data, which indicates the existence of an upper limit to the number of learning counts for a fixed-size neural network.

Our proposed method using only end-to-end deep learning can democratize the random number generator, enabling ordinary engineers to develop their own PRNGs according to their needs. Our proposed method enables random number generation algorithms to be developed without deep mathematical knowledge or experience, which are traditionally considered indispensable, making it easy to create new and disposable random number generation algorithms. This means that a new publicly unknown and unpredictable random number generation algorithm can be used with confidence. In addition, the neural networks used in our

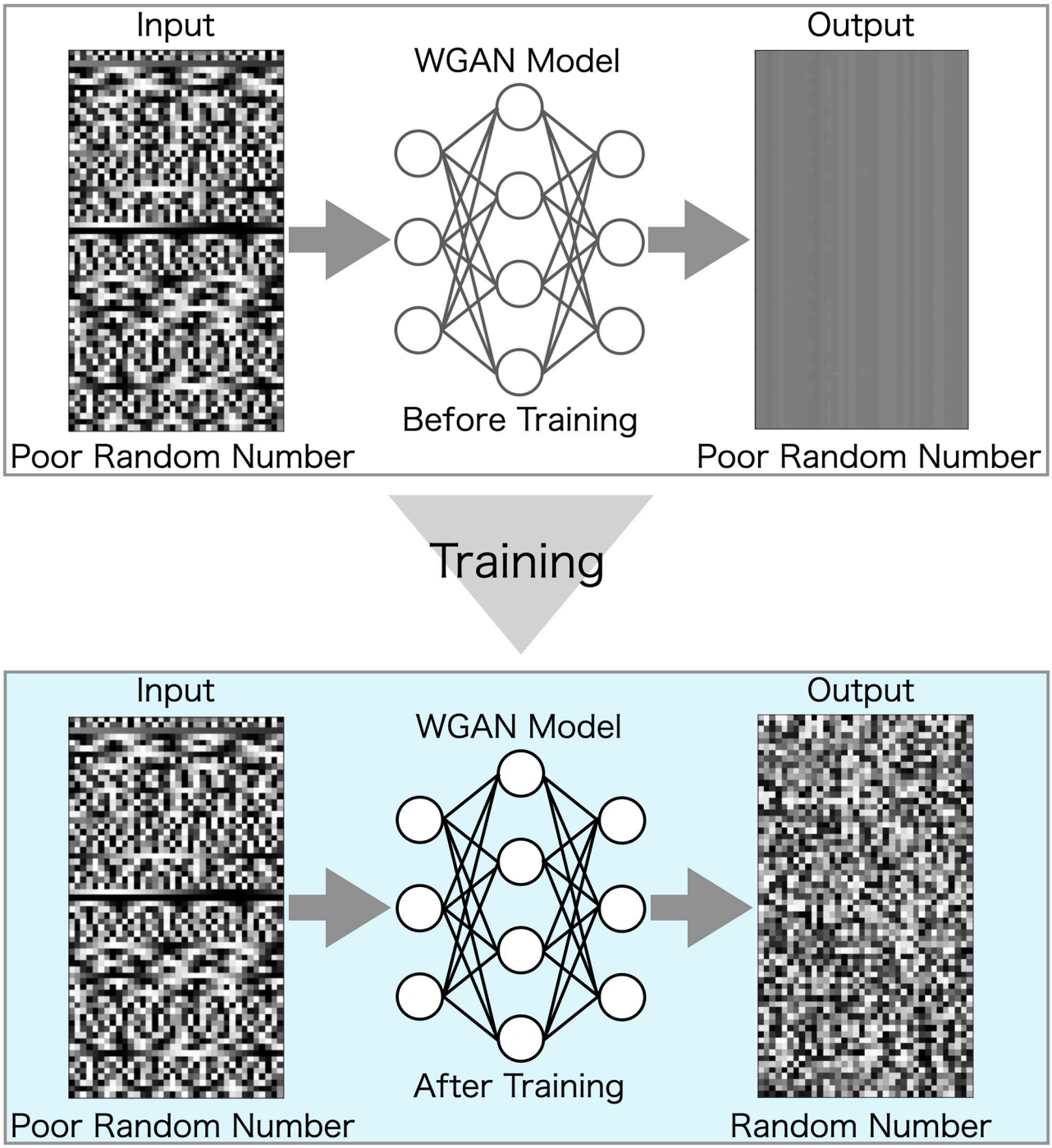

Fig 1. Schematic of this research. Upper image: WGAN model before training, which generates random numbers with poor properties. Lower image: WGAN model after training, which converts poor random numbers into random numbers with good properties.

https://doi.org/10.1371/journal.pone.0287025.g001

method are both portable and robust against reverse engineering. Here, portability means that they are independent of the computer architecture and can be operated on any computer, and robustness against reverse engineering means that the algorithms in the models are difficult to interpret by humans. DNNs are black-box approximators. Since our LPRNG use an algorithm

given by black-box approximators, they provide no human insights into the inner functioning of the PRNGs.

Our NN-based random number generator has two benefits: reverse engineering resistance and cross-platform portability among servers and clients. To clarify these benefits, we will first outline how commonly PRNGs are used in mobile applications such as games and social networking services. Pseudo-random numbers are widely used not only in high-risk areas like encryption and security but also in areas that require randomness and non-arbitrariness, such as the parameter for natural and smooth animation in user interfaces and the control parameters for computer-operated Non-Player Characters (NPCs) in games. PRNGs are particularly acknowledged as an essential technology in the gaming industry, which has a significant impact on determining the winner of a game. Yet, the number of behaviors that violate game regulations, such as reverse engineering [48] to the seed of a PRNG to anticipate the next random number, has increased exponentially due to the generalization of game development tools such as Unity [49]. It is crucial to provide a simple and portable method to prevent the reverse engineering of a PRNG that is used carelessly.

Thus, game industries have adopted encrypting random number generation processes and frequently changing the decryption key, approximately every few weeks, as the simplest and most practical way to prevent reverse engineering. Such a brute-force method is effective so long as the frequency of key changes exceeds the time and cost necessary to decrypt or to find the decryption key. However, due to the fact that the algorithm for the pseudo-random number generator is already known and limited in number such as Mersenne twister [32], y Multiply-With-Carry (MWC) [50], Xorshift [33], and The 64-bit Maximally Equidistributed F_2-Linear Generators with Mersenne Prime Period (MELG-64) [51], once a seed location in memory is decrypted, the next random number generation result may be simply anticipated. As new pseudorandom number generation algorithms are developed only every few years at least, it is totally impractical to try to develop a new PRNG algorithm every week to prevent guessing the random number generators.

Our Learned PRNG allows the random number generator to be automatically created weekly by ordinary engineers who are not experts in mathematics. This renders it impossible to predict the behavior of applications utilizing Learned PRNGs, even if the memory location of the seed is disclosed, because it is difficult to investigate the behavior of the learned PRNG using commonly available reverse engineering techniques. The Learned PRNG offers a novel counter-reverse-engineering strategy that enhances the update frequency of the random number generation period, which can be used with the usual approach that often updates the encryption key. Our Learned PRNG model allows us to generate a new PRNG periodically every week from a data set of random number sequences generated by existing methods. Our research will drastically reduce the cost of delivering a new PRNG to edge devices such as smartphones, allowing us to update PRNGs more frequently than it would cost to crack and identify the PRNG and its surrounding systems.

As for the portability of the Learned PRNG model, it is a "machine/language neutral" data structure that is independent of a particular programming language or CPU architecture. The services of today must be compatible with multiple platforms, such as Apple's iOS, Google's Android, Microsoft's Windows, Apple's macOS, and Linux. In addition, we employ multiple CPUs, including x86-64, ARM, and RISV-V, on both the client-side and server-side. Numerous programming languages, such as C/C+, Python, Java, JavaScript, C#, and Swift, are used to implement services. Consequently, the cost of adapting a new PRNG to all platforms, CPUs, and programming languages is exceptionally high. Our Learned PRNG model, on the other hand, can be represented as open neural network exchange (ONNX) [52], a machine-neutral and language-neutral model that is independent of these platforms, operating systems, and

programming languages. Dedicated runtimes [53] and compilers [54] enable our Learned PRNG model to effectively infer random numbers on a variety of platforms and hardware. Smartphones of the present day are typically equipped with specialized chips for the rapid execution of neural networks, such as the Apple Neural Engine, which can efficiently execute NN models. Therefore, the delivery of Learned PRNGs to edge devices is practical.

The proposed random number generation method using ML has the following applications:

1. Our LPRNG method realizes the one-off creation of new statistically valid PRNGs. Such one-off PRNG prevents mathematical or software-engineering experts from exploiting their knowledge by taking unfair profit from PRNG. Our LPRNG provides unpredictable behavior for everyone, including ordinary people and experts, because it includes no human insights into the inner functioning of the PRNG. This feature makes it impossible for anyone to use corruptly optimized and tuned random numbers to achieve desirable results.
2. Our LPRNG provides high-level anti-tampers for software developers who use PRNGs on the client-side such as PCs and smartphones. It is very difficult for malicious users to predict the next random numbers generated by our LPRNG, even if the seed number might be disclosed through the reverse-engineering because users cannot know the learning process and parameters essential to investigate the behavior of LPRNGs.

In the following, we show how to create an input seed, and we discuss the process by which our ML model outputs random numbers with sufficient properties in terms that pass statistical tests.

A significant difficulty with the Learning PRNG is that it requires a seed sequence equal to the quantity of random numbers to be generated. Most known PRNG algorithms use an asymptotic formula to produce random numbers constantly by recursively using the output as input to generate subsequent random numbers. Further research will validate that the Learned PRNG passes the NIST test suite even when the output of WGAN is utilized as input to WGAN to create continuous random numbers.

Methods

Wasserstein Generative Adversarial Network (WGAN)

Goodfellow et al. [46] initially proposed Generative adversarial networks (GANs), which are a type of deep generative model, and they are commonly used for generating images and predicting simulation results [55].

In this algorithm, a generator network G competes with a discriminator network D. After learning process, the generator will generate samples x that deceive the discriminator whereas the discriminator will correctly discriminate whether the samples are generated by the generator or not.

$$V(D, G) = \mathbb{E}_{x\sim p_r}[\log D(x)] + \mathbb{E}_{x\sim p_g}[\log(1 - D(x))]. \tag{1}$$

Where $D(x)$ is the probability that the discriminator decides a sample x is a real sample ($x \sim p_g$: probability distribution of false samples, $x \sim p_r$: probability distribution of real samples). GAN learning follows a mini-max game, represented by

$$\min_G \max_D V(D, G). \tag{2}$$

Where $V(D, G)$ is the objective function of the GAN.

Conventional GANs often pose significant challenges in their training process, primarily attributable to the instability of the learning processes [56, 57]. Two of these challenges stand out: one is the vanishing gradient problem, a well-known issue where the gradient of the loss function shrinks to an infinitesimally small value, inhibiting effective weight updates during backpropagation. The second, referred to as mode collapse, describes a phenomenon where the generator persistently yields identical, remarkably similar, or a restricted set of outputs, remaining unaffected by changes in the input noise vector. To address these predicaments that contribute to the learning instability, a variation of GANs known as Wasserstein GAN (WGAN) was proposed by Arjovsky et al. [47].

W denotes the cost of transporting earth from one location to another. If the earth is a probability distribution function, W is the minimum total work involved in converting one distribution to another. The GAN objective function $V(D, G)$ can be expressed using the Wasserstein distance as

$$\begin{aligned} V(D,G) &= W(p_r, p_g) \\ &= \max_{\phi} \mathbb{E}_{x\sim p_r}[D(x;\phi)] - \mathbb{E}_{z\sim p_z}[D(G(z;\theta);\phi)], \end{aligned} \tag{3}$$

where $D(x; \phi)$ is a function satisfying the Lipschitz continuity $||D(x_1) - D(x_2)|| \leq ||x_1 - x_2||$ and here, we used both spectral normalization [58] and gradient penalty [59] (Wasserstein GAN with gradient penalty: WGAN-GP).

Training of Wasserstein Generative Adversarial Network

We built the machine learning model to predict the structure of the random numbers generated by MT using the WGAN [46, 47]. We trained the WGAN with the MT random numbers used as training data and used the WGAN as a model to output the random numbers. The WGAN trained with various random images is expected to generate images with many types of random structures.

Fig 2 shows the architecture of the WGAN. For each layer, the Leaky ReLU function is used as the activation function of the input and hidden layers. The output of the generator is scaled to the range of (0,1) using the sigmoid function. A schematic view of the training process is shown in Fig 3. Poor-quality random numbers generated by the cosine function (Eq 4) were used for the input seed of the generator. Input seed has 36 dimensions, and we used 64 batches. This input seed has low-quality randomness in terms of the NIST test results. Then, the discriminator decides whether the generated numbers are random numbers or low-quality random numbers. In this study, MT random numbers were used as training data. We set the dimension of the latent variable to 36 and defined each latent variable as a floating-point number in the range from 0 to 1 generated by the cosine function. The discriminator was trained five times for each training of the generator. One update of the parameters of the generator after updating the parameters of the discriminator five times was considered as one iteration. The RMSprop optimizer was used for the discriminator and generator. Spectral normalization [58] was implemented in all the linear layers in the discriminator. We also applied a gradient penalty [59].

WGAN without dropout layers

In this study, we used the WGAN model, which excludes the dropout layers through tuning for random number generation by machine learning. We originally used a model with dropout layers, but the removal of the dropout layers markedly improved the quality of the generated random numbers. In general, as the structure of a neural network becomes increasingly

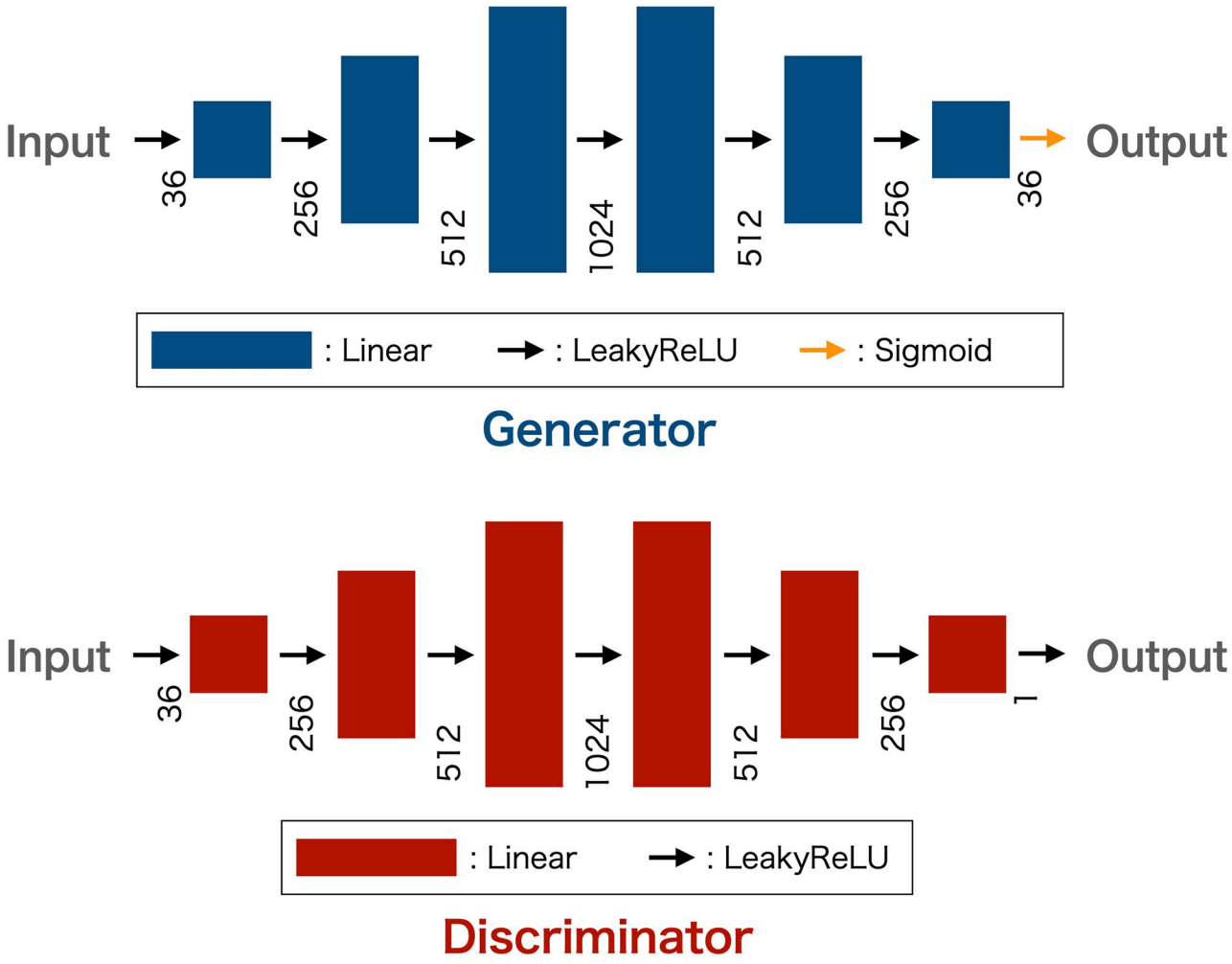

Fig 2. WGAN model. Upper figure: Generator model. Lower figure: discriminator model.

https://doi.org/10.1371/journal.pone.0287025.g002

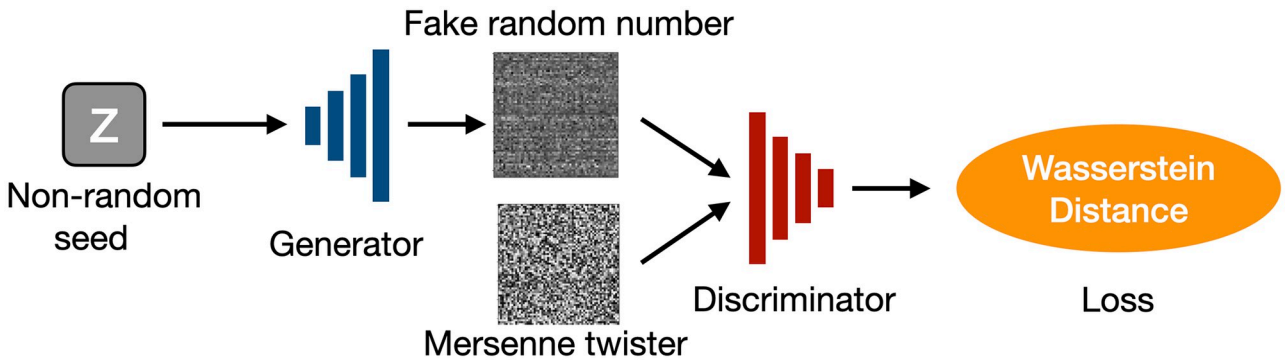

Fig 3. Schematic representation of the learning process of the Wasserstein GAN. At each iteration, the generator receives the latent variable z and outputs a false image, and the discriminator receives the false and real images and calculates the loss using the Wasserstein distance.

https://doi.org/10.1371/journal.pone.0287025.g003

complex, the weights of the neurons become optimized for the training data set. As it is, the model lacks generalization performance and becomes poor and can only be used for the training data set as if memorizing the data one by one. The percentage of correct answers in the training data will gradually increase as the learning proceeds, but the error in the test data will stop decreasing and start increasing again. Such a situation is called overfitting. The dropout layer is used to prevent this phenomenon. In our case, we have an infinite number of random numbers as the training data. Moreover, the learning process was accelerated by removing the dropout layers.

NIST tests

To evaluate the randomness of the numbers generated from the machine learning model, we employed the NIST random number test suite, which consists of the following 15 tests, Frequency Test, Block Frequency Test, Cumulative Sums Test, Runs Test, Longest Run Test, Rank Test, FFT Test, Non-Overlapping Template Test, Overlapping Template Test, Universal Test, Approximate Entropy Test, Random Excursions Test, Random Excursions Variant Test, Serial Test, Linear Complexity Test. Please refer to the specification [30] for details.

We used each series of 1048576 bits for each test method, and the results of the 1000 series were used to make comprehensive results. As these tests were based on a statistical analysis of a random stream, a p-value was reported for each test or subtest, indicating the probability that the result is due to randomness. A significance level of 0.01 was used for all tests.

The method of bit window extraction

Binary data must be prepared for evaluation using the NIST random number test. On the other hand, in machine learning, a decimal floating-point representation is generally used. In our experiments, we extracted the binary data from the floating-point representation, consisting of the sign, exponent, and mantissa parts. When considering a random number between 0 and 1, the sign part and the exponent part are almost unchanged. This means that the randomness of the bit sequence is low. On the other hand, some of the mantissa part may contain randomness. To use the randomness, we extracted several bits from the mantissa part that were found to have sufficient randomness in terms of NIST test results in the MT case, as shown in Fig 4. The results of the NIST tests are for these several bits from the mantissa part of the generated random numbers.

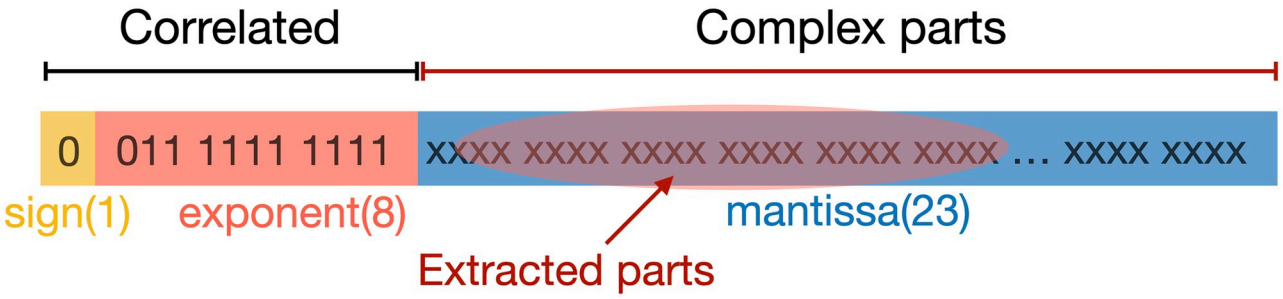

Fig 4. Bit window extraction method. We extracted several bits from the mantissa part that were found to have sufficient randomness in the case of MT-generated random numbers.

https://doi.org/10.1371/journal.pone.0287025.g004

Results and discussion

Creating an input seed

The main objective of this study is to generate random numbers using machine learning techniques, specifically by using GANs. When attempting to generate random numbers using GANs, it is important to note that the input seed should not possess random properties. Although random numbers are commonly used as input seeds, in this study, using random numbers as input seeds would not allow us to determine whether the GAN learned the random properties to generate output or simply output the input seed's properties without learning any random properties. If a sequence of numbers without randomness is used as the input seed and the output has randomness through transformation by GAN, it can be considered that GAN has learned randomness and is able to transform it into a sequence of numbers with randomness. Therefore, we devised poor random input seeds that are suitable for GAN learning. There are many possibilities for poor random number sequences, but it is important to provide various numbers to the GAN during learning. we considered number sequences with regularity or periodicity as poor random number sequences and introduced trigonometric functions. Even if a sequence of numbers can be converted to random numbers in a GAN conversion, the relationship between batches of numbers cannot be controlled by the GAN. To address this issue, we used different initial phases (R_2) and periods (R_1) of trigonometric functions for each batch to reduce the similarity between number sequences in advance. This is also important in providing diverse input seeds for GAN learning. The finally selected input seed was represented by

$$\text{input} = \cos(2R_1\pi t + 2R_2\pi) \times \frac{1}{2} + 1, \tag{4}$$

where $0 \leq t < 36$ and R_1 and R_2 are random numbers in the range of $0 \leq t \leq 1$ which are introduced to reduce the similarity of the numerical sequences and to provide a variety of input seeds. The input was also adjusted to be between 0 and 1. A sequence of numbers generated by the cosine function with 64 unique periods is shown in Fig 5. Periodicity in the horizontal direction can be seen. The results of evaluating this numerical sequence by the NIST test in Table 1 show that this numerical sequence does not have sufficient randomness. This sequence of numbers was used as the initial input seed for machine learning. The use of these input seeds is also different from the previous studies.

Transition of learning

Fig 6 shows the transition of the loss function. During the initial phase of training (approximately 250,000 iterations), it can be observed that the loss decreases as the learning progresses. However, in the latter half of the training, the loss starts to increase again. Fig 7(a) shows the NIST test results of the random numbers outputted at each learning step. We ran the NIST test after each set of 3000 iterations by connecting multiple batches of random numbers generated at each step. We divided the learning process into four regions denoted W, X, Y, and Z. In region W, the NIST score improved as the learning progressed. In region X, the NIST score fluctuated around 11, always failing to pass the Frequency, CumulativeSums, and Runs tests, often failing NonOverlappingTemplate test and rarely failing to pass the OverlappingTemplate, Universal, ApproximateEntropy, and Serial tests. In region Y, the generated random numbers finally passed the NIST test completely but often failed to pass the Frequency, CumulativeSums, Runs, and NonOverlappingTemplate tests. It was difficult for our model to pass the Frequency and CumulativeSums tests. Note that just by learning MT random numbers, it becomes possible to pass the NIST 15-item statistical test. Tables 2 and 3 respectively show the

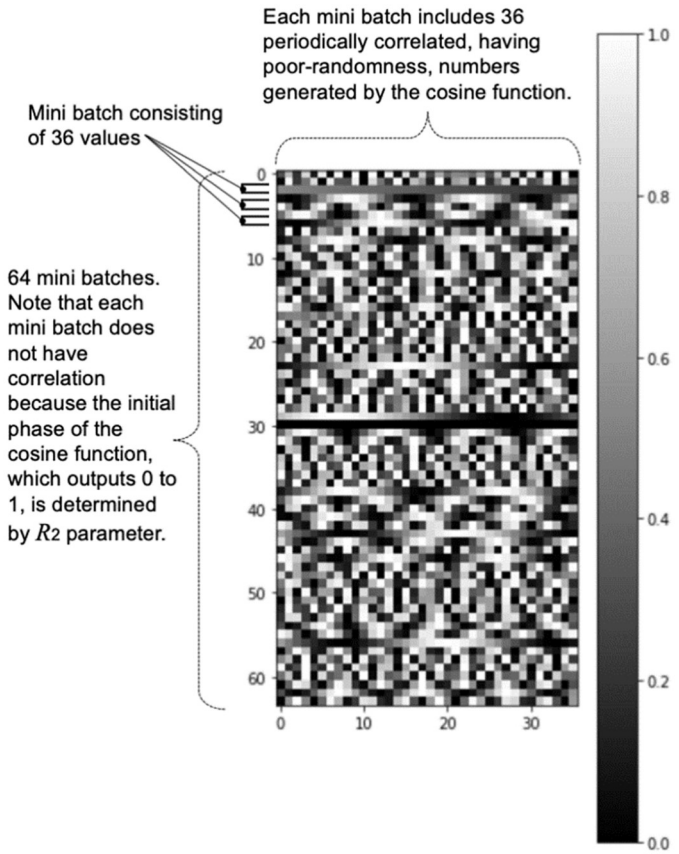

Fig 5. Simple sequence of numbers generated by the cos function. The horizontal axis represents a sequence of poor random numbers representing a single input seed, and the vertical axis represents the data from each of the 64 batches.

https://doi.org/10.1371/journal.pone.0287025.g005

Table 1. NIST statistical test results for input seed.

Statistical test	Pass rate	Uniformity of *p*-values	Pass/fail judgment
Frequency	0.000	Fail	Fail
BlockFrequency	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
Runs	0.000	Fail	Fail
LongestRun	0.000	Fail	Fail
Rank	0.995	Pass	Pass
FFT	0.983	Pass	Pass
NonOverlappingTemplate	0.784	36/148 Pass	Fail
OverlappingTemplate	0.000	Fail	Fail
Universal	0.541	Fail	Fail
ApproximateEntropy	0.000	Fail	Fail
RandomExcursions	0.000	0/8 Pass	Fail
RandomExcursionsVariant	0.000	0/18 Pass	Fail
Serial	0.000	Fail	Fail
Serial	0.800	Fail	Fail
LinearComplexity	0.992	Pass	Pass

https://doi.org/10.1371/journal.pone.0287025.t001

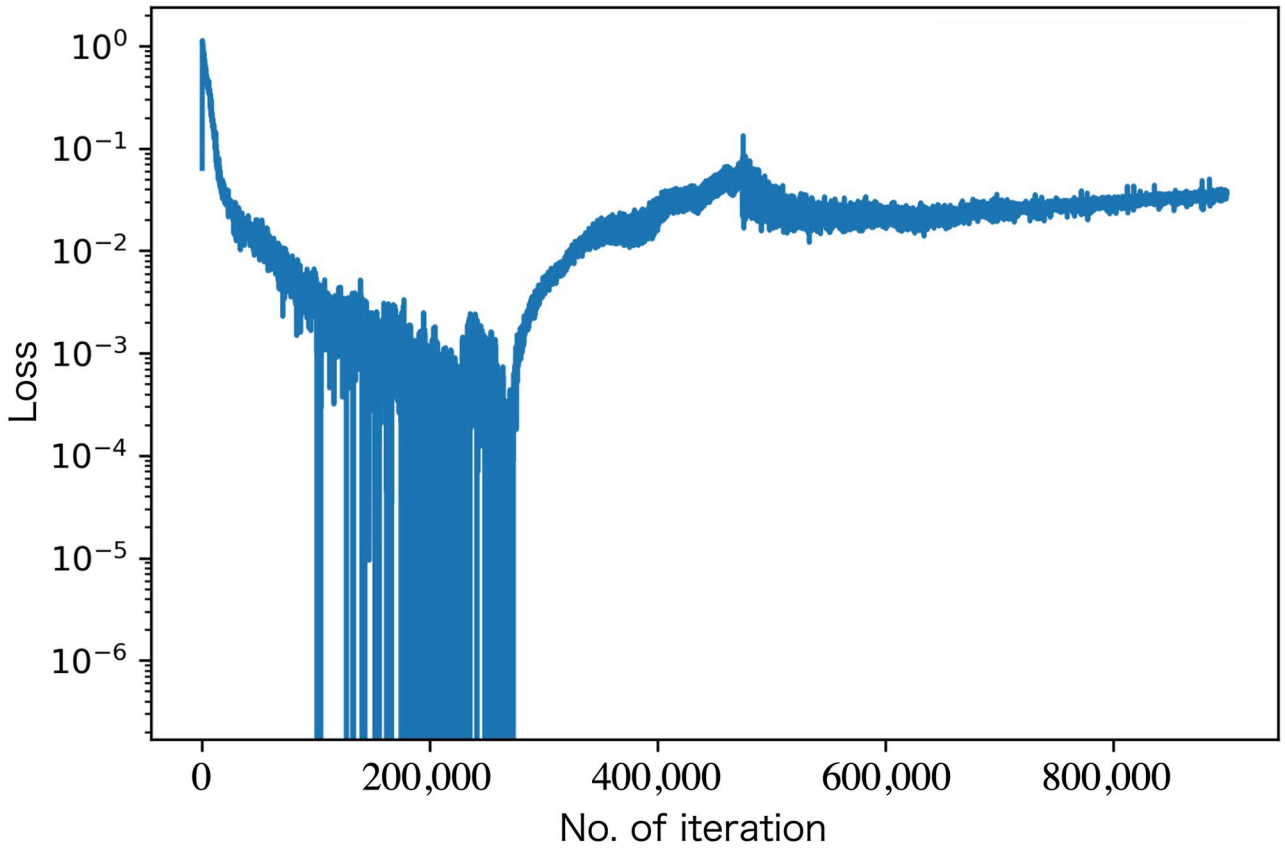

Fig 6. Loss function as a function of a number of iterations.

https://doi.org/10.1371/journal.pone.0287025.g006

scores of the NIST tests for the random numbers outputted before and after learning in which a perfect score was obtained (the data for A and C in Fig 7). In images A, B, and C in Fig 7(b), the growth of randomness can be seen. Note that our model has difficulty in stably passing all NIST tests during continuous learning. However, we found that we can generate good random numbers even if our model learns to output a small amount of random numbers (but there are many batches), because the random numbers created by connecting multiple outputs consisting of 36 float type bitstreams pass the NIST test. Fig 8 shows the results when a drop-apt layer is added. As can be seen, the learning results are better when the Dropout layer is removed.

After achieving a perfect score on the NIST test in region Y, the score deteriorates in region Z. In this region, only the Rank, FFT, RandomExcursions, RandomExcursionsVariant, and LinearComplexity tests are usually passed, and ultimately the RandomExcursions and RandomExcursionsVariant tests are failed. Throughout the learning, Rank, FFT, and LinearComplexity tests are easily passed, whereas it is difficult to pass the Frequency and CumulativeSums tests. Fig 7(b) also shows that the random images after long-term learning (D) are not random, and the score of the NIST test is shown in Table 4. In this machine learning process, over fitting occurs even though an infinite random dataset is used. Over fitting is generally caused by the use of finite datasets, but our results show that it can also occur for infinite datasets. We believe that the early stopping of the algorithm is effective even when infinite datasets are used. We additionally discussed two cases where the random property of the input seed was better or worse.

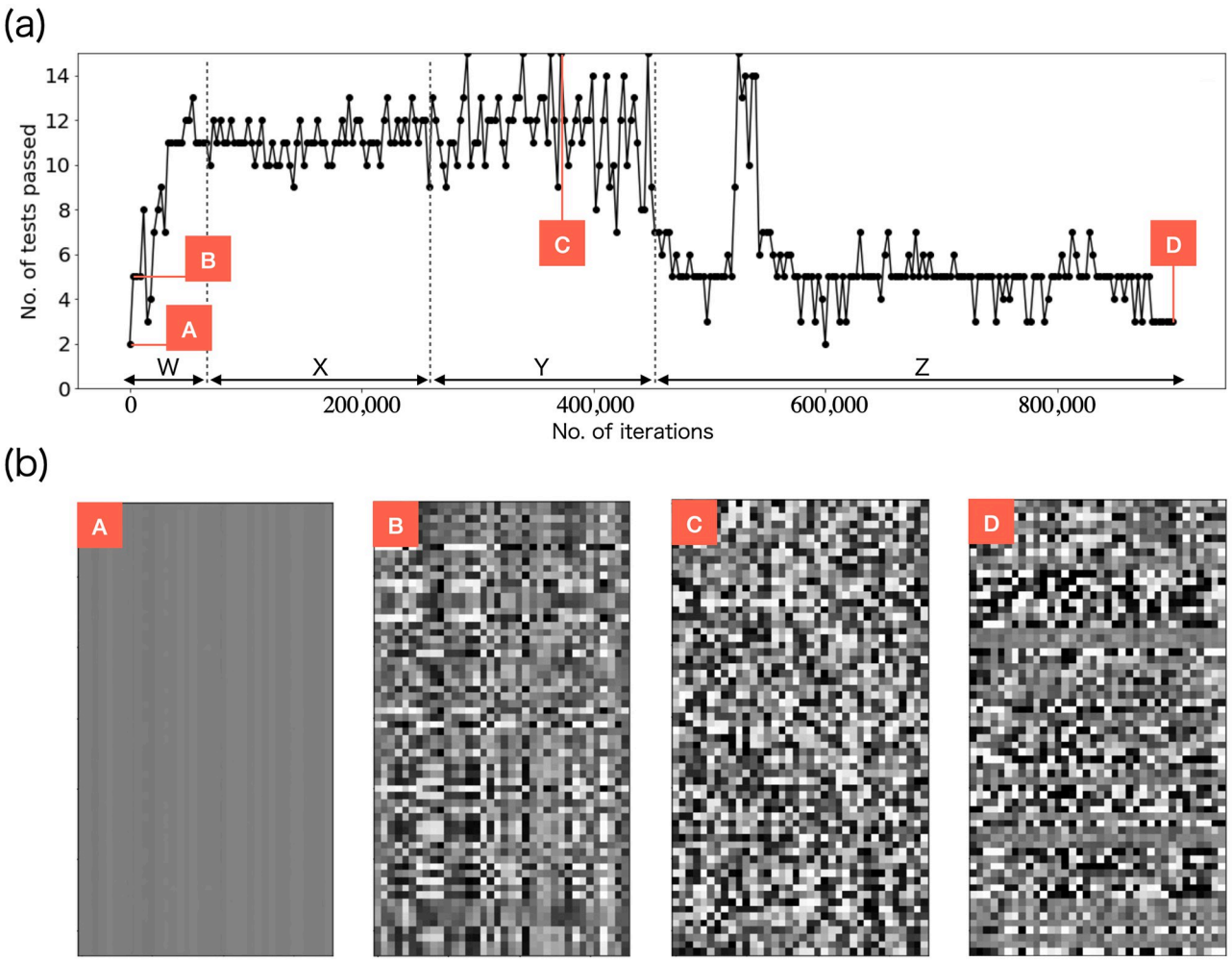

Fig 7. NIST test results and images of generated random numbers at different numbers of iterations. The NIST scores of the random numbers improve in the order A < B < C. The random number images show that the randomness increases. The random number performance degrades from C to D, indicating that over fitting occurs despite the use of infinite training data. One update of the parameters of the generator after updating the parameters of the discriminator five times was considered as one iteration.

https://doi.org/10.1371/journal.pone.0287025.g007

1. What happens if we use input seeds that have random properties? In this case, the input seed will be the same as those widely used in general, which is more suitable for machine learning. Consequently, the output will also produce random numbers that clear the NIST tests. However, it is difficult to determine whether these random numbers are superior to those generated using input seeds made with trigonometric functions. This is because our study used the NIST test for random verification and we cannot compare the quality of the two outputs beyond the range of a perfect score. Therefore, in this case, it is also possible that the machine learning model does not learn the randomness property, but learns to output the property of the input seed as it is, making it difficult to determine whether the machine learning model has successfully imitated the random properties.
2. What happens if we use input seeds that have worse properties as random numbers? Although we developed input seeds using trigonometric functions, there are several methods to create input seeds with worse properties than these. For example, consider a

Table 2. NIST statistical test results for data A in Fig 7 (before learning).

Statistical test	Pass rate	Uniformity of *p*-values	Pass/fail judgment
Frequency	0.000	Fail	Fail
BlockFrequency	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
Runs	0.000	Fail	Fail
LongestRun	0.000	Fail	Fail
Rank	0.990	Pass	Pass
FFT	0.000	Fail	Fail
NonOverlappingTemplate	0.113	0/148 Pass	Fail
OverlappingTemplate	0.000	Fail	Fail
Universal	0.000	Fail	Fail
ApproximateEntropy	0.000	Fail	Fail
RandomExcursions	0.000	0/8 Pass	Fail
RandomExcursionsVariant	0.000	0/18 Pass	Fail
Serial	0.000	Fail	Fail
Serial	0.000	Fail	Fail
LinearComplexity	0.987	Pass	Pass

https://doi.org/10.1371/journal.pone.0287025.t002

numerical sequence where 1 always appears continuously. In this case, the value of the input seed becomes limited, and the learning process becomes extremely difficult, which results in the difficulty in obtaining random numbers with good properties as outputs. As one of our contributions to this study, we can emphasize that we have developed a way to create numerical sequences without random properties that do not significantly affect the machine learning process.

Table 3. NIST statistical test results for data C in Fig 7 (successful area).

Statistical test	Pass rate	Uniformity of *p*-values	Pass/fail judgment
Frequency	0.992	Pass	Pass
BlockFrequency	0.988	Pass	Pass
CumulativeSums	0.992	Pass	Pass
CumulativeSums	0.995	Pass	Pass
Runs	0.991	Pass	Pass
LongestRun	0.988	Pass	Pass
Rank	0.982	Pass	Pass
FFT	0.989	Pass	Pass
NonOverlappingTemplate	0.990	148/148 Pass	Pass
OverlappingTemplate	0.989	Pass	Pass
Universal	0.991	Pass	Pass
ApproximateEntropy	0.987	Pass	Pass
RandomExcursions	0.990	8/8 Pass	Pass
RandomExcursionsVariant	0.990	18/18 Pass	Pass
Serial	0.988	Pass	Pass
Serial	0.987	Pass	Pass
LinearComplexity	0.991	Pass	Pass

https://doi.org/10.1371/journal.pone.0287025.t003

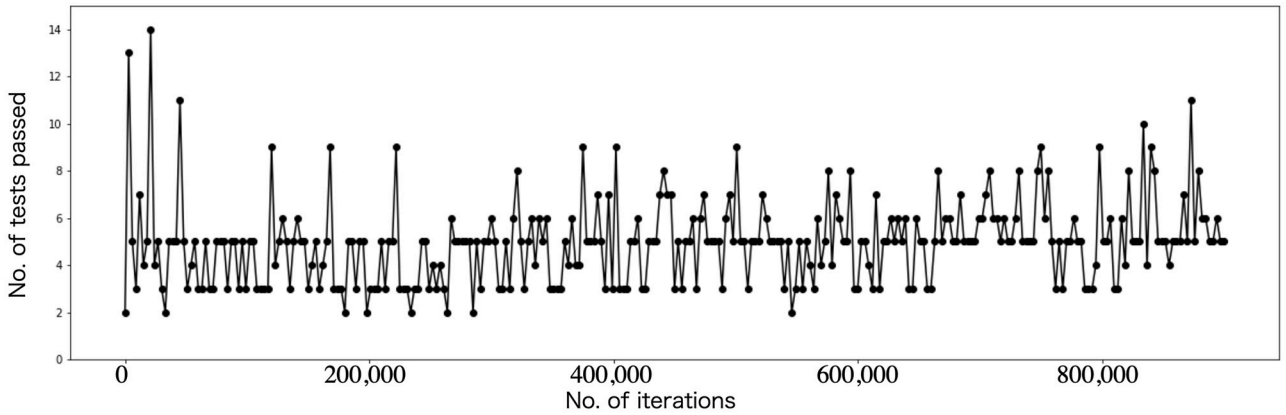

Fig 8. NIST test results when WGAN learned with dropout layer.

https://doi.org/10.1371/journal.pone.0287025.g008

Conclusion and future work

In this study, we succeeded in generating random numbers using a WGAN with poor random number seeds. The obtained random numbers were evaluated by the NIST random number tests and found to pass all the tests. We also observed over fitting even though infinite training data were used. Through training, our model was found to easily pass the Rank, FFT, and LinearComplexity tests, but not the Frequency and CumulativeSums tests. The mechanism of deep learning is still not fully understood. In this study, we clarified how the generator learns by examining the tests that were passed. The evaluation of learning by statistical tests, such as random number tests, may bring new insights into our understanding of deep learning. This research has various applications. 1. By using random numbers that follow an arbitrary distribution as the training data for learning, it may become possible to design a random number

Table 4. NIST statistical test results for data D in Fig 7 (over fitting area).

Statistical test	Pass rate	Uniformity of *p*-values	Pass/fail judgment
Frequency	0.000	Fail	Fail
BlockFrequency	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
CumulativeSums	0.000	Fail	Fail
Runs	0.000	Fail	Fail
LongestRun	0.000	Fail	Fail
Rank	0.992	Pass	Pass
FFT	0.984	Pass	Pass
NonOverlappingTemplate	0.900	75/148 Pass	Fail
OverlappingTemplate	0.000	Fail	Fail
Universal	0.752	Fail	Fail
ApproximateEntropy	0.000	Fail	Fail
RandomExcursions	1.000	0/8 Pass	Fail
RandomExcursionsVariant	1.000	0/18 Pass	Fail
Serial	0.370	Fail	Fail
Serial	0.989	Fail	Fail
LinearComplexity	0.991	Pass	Pass

https://doi.org/10.1371/journal.pone.0287025.t004

generation method that follows an arbitrary distribution. 2. Training machine learning to take computational efficiency into account will lead to the development of computationally efficient and fast random number generation methods. 3. Random number testing methods using machine learning have been investigated in previous studies [60]. The discriminator used in GAN training also can be used as a random number test as it distinguishes between random and poor random numbers. In addition, unlike the NIST test, there is no limit to the amount of input data, so a discriminator can be used to test random numbers for arbitrary input data sizes. Our results provide useful information for the use of machine learning and the future development of random number generation methods.

Author Contributions

Conceptualization: Kiyoshiro Okada, Shuichi Kurabayashi.

Data curation: Kiyoshiro Okada.

Formal analysis: Kiyoshiro Okada.

Investigation: Kiyoshiro Okada, Shuichi Kurabayashi.

Methodology: Kiyoshiro Okada, Katsuhiro Endo, Kenji Yasuoka, Shuichi Kurabayashi.

Project administration: Shuichi Kurabayashi.

Software: Kiyoshiro Okada, Katsuhiro Endo.

Supervision: Katsuhiro Endo, Kenji Yasuoka, Shuichi Kurabayashi.

Validation: Kiyoshiro Okada, Katsuhiro Endo.

Writing – original draft: Kiyoshiro Okada.

Writing – review & editing: Kiyoshiro Okada, Kenji Yasuoka, Shuichi Kurabayashi.

References

1. Motwani R, Raghavan P. Randomized algorithms. Cambridge University Press; 1995.
2. Xu H, Tong X, Meng X. An efficient chaos pseudo-random number generator applied to video encryption. Optik. 2016; 127(20):9305–9319. https://doi.org/10.1016/j.ijleo.2016.07.024
3. Li B, Liao X, Jiang Y. A novel image encryption scheme based on improved random number generator and its implementation. Nonlinear Dynamics. 2019; 95(3):1781–1805. https://doi.org/10.1007/s11071-018-4659-2
4. Sato Y, Brückner S, Kurabayashi S, Waragai I. An Empirical Taxonomy of Monetized Random Reward Mechanisms in Games. Proceedings of DiGRA 2020. 2020;21.
5. Hou S, Wang Z. Weighted channel dropout for regularization of deep convolutional neural network. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33; 2019. p. 8425–8432.
6. Bird JJ, Ekárt A, Faria DR. On the effects of pseudorandom and quantum-random number generators in soft computing. Soft Computing. 2020; 24(12):9243–9256. https://doi.org/10.1007/s00500-019-04450-0
7. Groot RD, Warren PB. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. The Journal of Chemical Physics. 1997; 107(11):4423–4435. https://doi.org/10.1063/1.474784
8. Hoogerbrugge P, Koelman J. Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. EPL (Europhysics Letters). 1992; 19(3):155. https://doi.org/10.1209/0295-5075/19/3/001
9. Espanol P, Warren P. Statistical mechanics of dissipative particle dynamics. EPL (Europhysics Letters). 1995; 30(4):191. https://doi.org/10.1209/0295-5075/30/4/001
10. Okada K, Brumby PE, Yasuoka K. The influence of random number generation in dissipative particle dynamics simulations using a cryptographic hash function. PloS One. 2021; 16(4):e0250593. https://doi.org/10.1371/journal.pone.0250593 PMID: 33905444

11. Okada K, Brumby PE, Yasuoka K. An Efficient Random Number Generation Method for Molecular Simulation. Journal of Chemical Information and Modeling. 2021; 62(1):71–78. https://doi.org/10.1021/acs.jcim.1c01206 PMID: 34951306

12. Kobayashi R. Modeling and numerical simulations of dendritic crystal growth. Physica D: Nonlinear Phenomena. 1993; 63(3-4):410–423. https://doi.org/10.1016/0167-2789(93)90120-P

13. Kobayashi R. A numerical approach to three-dimensional dendritic solidification. Experimental mathematics. 1994; 3(1):59–81. https://doi.org/10.1080/10586458.1994.10504577

14. Uchida A, Amano K, Inoue M, Hirano K, Naito S, Someya H, et al. Fast physical random bit generation with chaotic semiconductor lasers. Nature Photonics. 2008; 2(12):728–732. https://doi.org/10.1038/nphoton.2008.227

15. Reidler I, Aviad Y, Rosenbluh M, Kanter I. Ultrahigh-speed random number generation based on a chaotic semiconductor laser. Physical Review Letters. 2009; 103(2):024102. https://doi.org/10.1103/PhysRevLett.103.024102 PMID: 19659208

16. Kanter I, Aviad Y, Reidler I, Cohen E, Rosenbluh M. An optical ultrafast random bit generator. Nature Photonics. 2010; 4(1):58–61. https://doi.org/10.1038/nphoton.2009.235

17. Argyris A, Deligiannidis S, Pikasis E, Bogris A, Syvridis D. Implementation of 140 Gb/s true random bit generator based on a chaotic photonic integrated circuit. Optics express. 2010; 18(18):18763–18768. https://doi.org/10.1364/OE.18.018763 PMID: 20940769

18. Hirano K, Yamazaki T, Morikatsu S, Okumura H, Aida H, Uchida A, et al. Fast random bit generation with bandwidth-enhanced chaos in semiconductor lasers. Optics express. 2010; 18(6):5512–5524. https://doi.org/10.1364/OE.18.005512 PMID: 20389568

19. Zhang J, Wang Y, Liu M, Xue L, Li P, Wang A, et al. A robust random number generator based on differential comparison of chaotic laser signals. Optics express. 2012; 20(7):7496–7506. https://doi.org/10.1364/OE.20.007496 PMID: 22453429

20. Li XZ, Chan SC. Heterodyne random bit generation using an optically injected semiconductor laser in chaos. IEEE Journal of Quantum Electronics. 2013; 49(10):829–838. https://doi.org/10.1109/JQE.2013.2279261

21. Oliver N, Soriano MC, Sukow DW, Fischer I. Fast random bit generation using a chaotic laser: approaching the information theoretic limit. IEEE Journal of Quantum Electronics. 2013; 49(11):910–918. https://doi.org/10.1109/JQE.2013.2280917

22. Virte M, Mercier E, Thienpont H, Panajotov K, Sciamanna M. Physical random bit generation from chaotic solitary laser diode. Optics express. 2014; 22(14):17271–17280. https://doi.org/10.1364/OE.22.017271 PMID: 25090541

23. Sakuraba R, Iwakawa K, Kanno K, Uchida A. Tb/s physical random bit generation with bandwidth-enhanced chaos in three-cascaded semiconductor lasers. Optics express. 2015; 23(2):1470–1490. https://doi.org/10.1364/OE.23.001470 PMID: 25835904

24. Tang X, Wu ZM, Wu JG, Deng T, Chen JJ, Fan L, et al. Tbits/s physical random bit generation based on mutually coupled semiconductor laser chaotic entropy source. Optics Express. 2015; 23(26):33130–33141. https://doi.org/10.1364/OE.23.033130 PMID: 26831980

25. Butler T, Durkan C, Goulding D, Slepneva S, Kelleher B, Hegarty S, et al. Optical ultrafast random number generation at 1 Tb/s using a turbulent semiconductor ring cavity laser. Optics letters. 2016; 41(2):388–391. https://doi.org/10.1364/OL.41.000388 PMID: 26766721

26. Shinohara S, Arai K, Davis P, Sunada S, Harayama T. Chaotic laser based physical random bit streaming system with a computer application interface. Optics express. 2017; 25(6):6461–6474. https://doi.org/10.1364/OE.25.006461 PMID: 28380996

27. Ugajin K, Terashima Y, Iwakawa K, Uchida A, Harayama T, Yoshimura K, et al. Real-time fast physical random number generator with a photonic integrated circuit. Optics express. 2017; 25(6):6511–6523. https://doi.org/10.1364/OE.25.006511 PMID: 28380999

28. Xiang S, Wang B, Wang Y, Han Y, Wen A, Hao Y. 2.24-Tb/s physical random bit generation with minimal post-processing based on chaotic semiconductor lasers network. Journal of Lightwave Technology. 2019; 37(16):3987–3993. https://doi.org/10.1109/JLT.2019.2920476

29. Kim K, Bittner S, Zeng Y, Guazzotti S, Hess O, Wang QJ, et al. Massively parallel ultrafast random bit generation with a chip-scale laser. Science. 2021; 371(6532):948–952. https://doi.org/10.1126/science.abc2666 PMID: 33632847

30. Rukhin A, Soto J, Nechvatal J, Smid M, Barker E. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Booz-allen and hamilton inc mclean va; 2001.

31. Bassham III LE, Rukhin AL, Soto J, Nechvatal JR, Smid ME, Barker EB, et al. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications; 2010.

32. Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS). 1998; 8 (1):3–30. https://doi.org/10.1145/272991.272995

33. Marsaglia G, et al. Xorshift rngs. Journal of Statistical Software. 2003; 8(14):1–6. https://doi.org/10.18637/jss.v008.i14

34. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015; 521(7553):436–444. https://doi.org/10.1038/nature14539 PMID: 26017442

35. Desai V, Deshmukh V, Rao D. Pseudo random number generator using Elman neural network. In: 2011 IEEE Recent Advances in Intelligent Computational Systems. IEEE; 2011. p. 251–254.

36. Desai V, Patil RT, Deshmukh V, Rao D. Pseudo random number generator using time delay neural network. World. 2012; 2(10):165–169.

37. Desai V, Patil R, Rao D. Using layer recurrent neural network to generate pseudo random number sequences. International Journal of Computer Science Issues. 2012; 9(2):324–334.

38. Jeong YS, Oh K, Cho CK, Choi HJ. Pseudo random number generation using LSTMs and irrational numbers. In: 2018 IEEE International Conference on Big Data and Smart Computing (BigComp). IEEE; 2018. p. 541–544.

39. Pasqualini L, Parton M. Pseudo Random Number Generation through Reinforcement Learning and Recurrent Neural Networks. Algorithms. 2020; 13(11):307. https://doi.org/10.3390/a13110307

40. Tirdad K, Sadeghian A. Hopfield neural networks as pseudo random number generators. In: 2010 Annual Meeting of the North American Fuzzy Information Processing Society. IEEE; 2010. p. 1–6.

41. Hameed SM, Ali LMM. Utilizing Hopfield neural network for pseudo-random number generator. In: 2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA). IEEE; 2018. p. 1–5.

42. Wang YH, Shen ZD, Zhang HG. Pseudo random number generator based on hopfield neural network. In: 2006 International Conference on Machine Learning and Cybernetics. IEEE; 2006. p. 2810–2813.

43. Pasqualini L, Parton M. Pseudo random number generation: A reinforcement learning approach. Procedia Computer Science. 2020; 170:1122–1127. https://doi.org/10.1016/j.procs.2020.03.057

44. De Bernardi M, Khouzani M, Malacaria P. Pseudo-random number generation using generative adversarial networks. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer; 2018. p. 191–200.

45. Oak R, Rahalkar C, Gujar D. Poster: Using Generative Adversarial Networks for Secure Pseudorandom Number Generation. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security; 2019. p. 2597–2599.

46. Goodfellow IJ, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, et al. Generative adversarial networks. arXiv preprint arXiv:14062661. 2014;.

47. Arjovsky M, Chintala S, Bottou L. Wasserstein generative adversarial networks. In: International conference on machine learning. PMLR; 2017. p. 214–223.

48. Tomičić I, Grd P, Schatten M. Reverse engineering of the MMORPG client protocol. In: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE; 2019. p. 1099–1104.

49. Haas JK. A history of the unity game engine. Diss Worcester Polytechnic Institute. 2014; 483(2014):484.

50. Marsaglia G. Random number generators. Journal of Modern Applied Statistical Methods. 2003; 2(1):2. https://doi.org/10.22237/jmasm/1051747320

51. Harase S, Kimoto T. Implementing 64-bit maximally equidistributed F2-linear generators with Mersenne prime period. ACM Transactions on Mathematical Software (TOMS). 2018; 44(3):1–11. https://doi.org/10.1145/3159444

52. Bai J, Lu F, Zhang K, et al. Onnx: Open neural network exchange. GitHub repository. 2019; p. 54.

53. ONNX Runtime developers: ONNX Runtime; accessed 2023-02-21.

54. Jin T, Bercea GT, Le TD, Chen T, Su G, Imai H, et al. Compiling onnx neural network models using mlir. arXiv preprint arXiv:200808272. 2020;.

55. Hiraide K, Hirayama K, Endo K, Muramatsu M. Application of deep learning to inverse design of phase separation structure in polymer alloy. Computational Materials Science. 2021; 190:110278. https://doi.org/10.1016/j.commatsci.2021.110278

56. Mao X, Li Q, Xie H, Lau RY, Wang Z, Paul Smolley S. Least squares generative adversarial networks. In: Proceedings of the IEEE international conference on computer vision; 2017. p. 2794–2802.

57. Metz L, Poole B, Pfau D, Sohl-Dickstein J. Unrolled generative adversarial networks. arXiv preprint arXiv:161102163. 2016;.

58. Miyato T, Kataoka T, Koyama M, Yoshida Y. Spectral normalization for generative adversarial networks. arXiv preprint arXiv:180205957. 2018;.

59. Gulrajani I, Ahmed F, Arjovsky M, Dumoulin V, Courville A. Improved training of wasserstein gans. arXiv preprint arXiv:170400028. 2017;.

60. Fischer T. Testing cryptographically secure pseudo random number generators with artificial neural networks. In: 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE). IEEE; 2018. p. 1214–1223.