Cryptographically Secure Pseudo-Random Number Generation using Generative Adversarial Networks

Student
Marcello DE BERNARDI

Supervisor
Dr. Arman KHOUZANI

Student email
m.e.debernardi@se15.qmul.ac.uk

Student number
150382405

April 26, 2018

Queen Mary
University of London

Abstract

Pseudo-random number generators are a fundamental element of many cryptographic systems such as *encryption algorithms* [38, p. 169] [31, p. 1]. As PRNGs are often a single point of failure for such systems, their design and analysis is an important field of investigation [31, p. 2] [21]. While *deep neural networks* have been tremendously successful in recent years [40, p. 24-29], little effort has gone into their application to the implementation of PRNGs. Some relatively obscure and complicated attempts have been made with little success [22] [23] [43] [44].

This investigation pursues a simple and elegant novel approach to the problem, by proposing the use of *generative adversarial networks* to train a neural network to behave as a cryptographically secure PRNG. This is a natural association, as a GAN closely resembles the adversarial nature of security problems. Furthermore, this work showcases a number of interesting modifications to the standard GAN architecture. The most significant is training the GAN's generator network to produce outputs that the adversary network cannot predict, rather than training the generator to mimic as reference distribution as is standard. Thus the pseudo-randomness property in the generator's output is formulated in terms of unpredictability by an improving opponent.

Using the NIST statistical test suite to evaluate the performance of the generator network, this work investigates the extent to which training the proposed models improves their performance as a PRNG, and discusses the possibility of using the models in a cryptographic context.

This report demonstrates that a generative adversarial network can effectively train the generator to produce pseudo-random number sequences with good statistical properties. At best, the trained generator could pass around 99% of test instances and 98% of different tests. These very strong results outperform most other results in the field, and are achieved with a much simpler and robust approach compared to previous attempts. While these metrics alone are not sufficient to justify use of the models in a cryptographic setting, there is a strong case for further investigation and improvements to the design.

Contents

List of Figures

List of Tables

Acknowledgements

1. Introduction

> From where we stand the rain seems random. If we could stand somewhere else, we would see the order in it.
>
> Tony Hillerman, Coyote Waits

A *random number* may informally be defined as a variable whose value is unpredictable, by virtue of the fact that all possible values are equally likely to appear [15, p. 7]. This notion of unpredictability, or *randomness*, is crucial to computer security, as it is a fundamental element of many cryptographic systems such as encryption algorithms, where security guarantees often rely on an adversary not knowing the value of some internal parameter [38, p. 169] [31, p. 1]. The task of obtaining unpredictable values for use in such applications is handled by systems known as *random number generators*, which may be implemented either as specialized hardware, software, or as a combination of both [38, p. 196, 172]. If the implementation is fundamentally deterministic, we refer to the system as a *pseudo-random number generator* [38, p. 169]. In many cryptographic systems, this "randomness source" is a single point of failure, making the generator's implementation a critical aspect of the overall design [31, p. 2]. Indeed, how to implement a good generator is a question considered by several books, from Donald Knuth's seminal *The Art of Computer Programming, Volume II: Seminumerical Algorithms* [24] to textbooks such as Katz and Lindell's *Introduction to Modern Cryptography* [30], as well as an active area of research.

Recent years have seen machine learning methods achieve a tremendous amount of success throughout all fields of human endeavor, including business, science, and engineering [40, p. 24-29]. This is particularly the case with *deep neural networks*, a parametric representation of a mathematical function consisting of computational units called *neurons* or *units*, usually arranged into layers [40, p. 731-732]. Major technology companies such as Google, Amazon, and Microsoft now provide plug-and-play machine learning solutions as part of their cloud platforms [35] [13] [19], and courses in machine learning are available at a vast number of universities as well as online. Indeed, public interest in machine learning is at an all-time high [6]. However, applications of machine learning to security have so far been limited [11].

1.1 Aims and Motivation

The precise aim of this research is to *determine whether a neural network can be trained to output sequences of numbers which appear to be randomly generated*, and, by extension, whether *such a neural network could be used as a pseudo-random number generator in a cryptographic context*.

The motivation for this work is two-fold. On one hand, it presents a significant challenge from the perspective of deep learning. While neural networks have been extremely successful in supervised learning of regression and classification models, as well as in producing new data mimicking an existing dataset [28], attempts to produce seemingly random outputs have shown poor results. Previous research efforts have often resorted to complex neural network architectures and contrived training procedures in an attempt to encourage chaotic behavior [22] [23] [43]. This work undertakes the task differently; the similarly adversarial nature of generating random numbers for cryptographic use and of training generative adversarial networks results in an approach that is conceptually both simple and elegant.

On the other hand, this investigation is also motivated by the needs of computer security. A hypothetical neural-network-based pseudo-random number generator would have several properties that may be desirable. This includes the ability to perform ad-hoc modifications to the generator by means of further training and the ability to arbitrarily extend the length of generated number sequences without necessarily decreasing their unpredictability by increasing the dimensionality of the network's input. Plausibly, these abilities could constitute the basis of strategies for dealing with the kind of non-statistical attacks described by Kelsey et al. in [31].

1.2 Approach, Scope, and Assumptions

This work unites the fields of machine learning and computer security by applying a recently formulated deep learning method known as *generative adversarial networks* [28] to the generation of pseudo-random numbers for use in cryptosystems. Two conceptually simple approaches are pursued and implemented; the implementations are evaluated for their strength as PRNGs using the NIST statistical test suite.

The scope of this investigation is confined to the statistical properties of a PRNG's outputs. Good statistical properties are a minimum requirement for a generator to be considered secure, though do not constitute a sufficient condition [38, p. 170]. *Cryptanalysis* of the PRNG is also required [39, Abstract][31, p. 3-6], but this is beyond the scope of the project.

This work hinges on the assumption that a neural network can represent a good pseudo-random generator function, and that discovering such a function by gradient descent is tractable.

1.3   Contributions and Findings

This work makes a number of novel contributions to the field by proposing several interesting modifications to the GAN framework. In summary, a simplification to the GAN framework that is applicable to this task is introduced, whereby the GAN does not include a reference dataset which the generator attempts to mimic. Furthermore, the network implementing the generator includes a custom activation function that computes a modulo operation over its outputs, and represents the internal state of a PRNG using a non-random external input. These concepts are explored in depth in section 3.1.

The overall product of these modifications is a system that is simple, conceptually elegant, and robust. The findings of this report show that the proposed approach is highly successful at training a neural network to behave as a PRNG with good statistical properties, and outperforms most previous results in neural network-based PRNGs. While the results are not strong enough to justify use in a cryptographic setting, this research makes a strong case for further investigation into the topic.

2. Background

Any one who considers arithmetical
methods of producing random
digits is, of course, in a state of sin.

John Von Neumann, 1951

This work lies at the intersection of machine learning and computer security. In particular, by exploring the applications of neural networks to computer security, it falls into the field of neural cryptography [33]. This section provides an overview of the required background for pseudo-random number generation, including a definition of random number sequences, some of their applications in cryptography, and guidelines on how to evaluate their suitability for such use. It also covers the basics of artificial neural networks before moving to the specific neural network techniques used in this work. Lastly, an overview of the most relevant literature is provided.

2.1 Random Numbers Sequences and Generators

Of primary concern to this research are the practical nature of random number sequences and the means by which such sequences may be generated. The nature of randomness is sidestepped, since, as remarked by Donald Knuth, a philosophical discussion almost invariably ensues [24, p. 2]. An interested reader may enjoy Deborah Bennett's *Randomness* [16].

2.1.1 Random Bits, Random Numbers, and Entropy

Basic concepts from probability theory recur throughout this report, such as sample spaces, random variables, and probability distribution functions. Since this work deals with electronically stored sequences of numbers, which by necessity are to be encoded in some fixed-digit format, it shall be implicit that all mentions of sample spaces and random variables refer to discrete sets and discrete probability distribution functions, respectively.

Definition 1. A *random number* x is a numeric value selected at random from an equiprobable sample space Ω. That is, the probability $\mathbb{P}(x)$ of the value being chosen from Ω is equal to that of all other possible values in Ω [15, p. 7] [39, s. 1.1.1]. It follows that a discrete random variable X defined as the outcome of a selection from Ω has the *discrete uniform distribution* (figure 2.1).
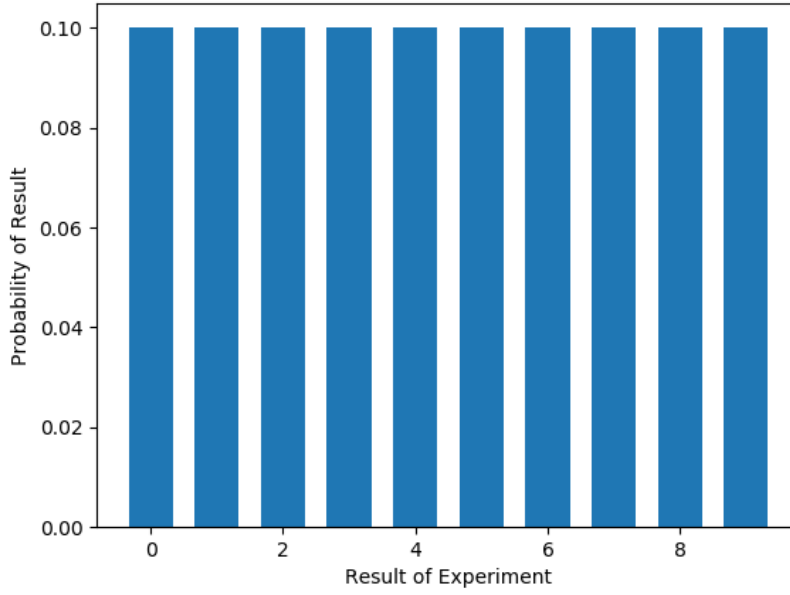
Figure 2.1: A discrete uniform probability distribution for an arbitrary experiment with 10 possible outcomes.

Definition 2. A *random bit b* is a special case of a random number, such that the equiprobable sample space is $\Omega = \{0, 1\}$ [39, s. 1.1.1].

Definition 3. A *random sequence* $s = (x_0, x_1, ..., x_n)$, as defined by Barker et al, is a sequence of random values x_i resulting from n independent selections. In other words, a random sequence consists of random numbers such that the probability distribution for any particular selection is entirely unaffected by previous selections. The random sequence has the same probability of being sampled as all other sequences of the same length [15, p. 7] [39, s. 1.1.1].

Every binary sequence represents, in some encoding scheme, a unique numerical value. For example, the binary sequence 101 is the unsigned integer representation of the number 5. Thus there is an equivalence between random binary sequences and random numbers, in that the probability of randomly sampling any particular binary sequence of length l from the set $\{0, 1\}$ is the same as that of selecting the number the sequence represents from the set of 2^l numbers that can be encoded with l bits. As observed by Menezes et al, we can therefore regard the task of producing a random number as equivalent to the task of producing a random binary sequence [38, p. 170].

A key property of random numbers and random sequences is *unpredictability*, for a sequence of random numbers, the probability of correctly guessing any value in the sequence is no greater than random chance [15, p. 7]. The degree of unpredictability of a number sequence is quantified by the concept of *entropy*, defined by Ferguson et al [20, p. 12-13] as follows:

Definition 4. Let X be a discrete random variable over a sample space Ω with a probability distribution function $p(x) = \mathbb{P}\{X = x\}, x \in \Omega$. The entropy $H(X)$, measured in

bits, of the discrete random variable X is defined as

$$H(X) = -\sum_{x \in \Omega} p(x) \lg p(x) \tag{2.1}$$

According to Ferguson et al., entropy is a subjective quantity, in the sense that it depends on the amount of knowledge available about a value of interest. The entropy of an arbitrary number is different for an observer that knows the number, and for one that only knows the value is one of 2^n possible values. The concept is not explored in depth in this report, where it is mentioned, it will suffice to informally understand that the more uncertain we are about a sequence, the higher its entropy [25, p. 137].

2.1.2 Random Number Generators

In order to obtain sequences of random numbers for use in applications such as those mentioned above, *random number generators* are used.

Definition 5. Menezes et al. define a random number generator as a software or hardware system that outputs random number (or bit) sequences. Key components of such a system are the *entropy source*, which gives rise to the randomness in the output, and the *entropy distillation* process, which is an algorithmic procedure applied to the produced values to improve the quality of the output sequence [38].

According to Ferguson et al and Menezes et al, entropy sources are commonly implemented in software, hardware, or both. Examples of entropy sources that are harnessed by software means include the timings of keystrokes on a computer user's keyboard, the current value of the system clock, the content of an I/O buffer, or any other measurable quantity in a computer system that is believed to exhibit random behavior. This conjecture does not always hold; for example, the time elapsed between keystrokes may not be accurately described by a uniform distribution, as an experienced typist may manage to keep their typing rate remarkably constant (with fluctuations on the order of several milliseconds). Care has to be taken to not overestimate the amount of entropy that can be derived from a source [25, p. 138-139] [38, p. 171-172].

Ferguson et al and Menezes et al also discuss hardware entropy sources. These rely on physical processes that behave randomly. Commonly cited examples are emission times of particles during radioactive decay or thermal noise in a resistor. While there are very many such processes (in particular in the "quantum realm" [17]), physically based entropy sources are not guaranteed to be flawless. Even if a process behaves randomly, the outputs may nonetheless be biased or correlated, possibly due to manipulation by a third party. Furthermore, a third party may be able to observe the physical entropy source; while still random, the outputs would no longer have any entropy from the their perspective [25, p. 138-139] [38, p. 172].

According to Ferguson, Schneier, and Kohno, there are several problems related to the practical use of truly random numbers. Real random data may not always be available, and even if available it is nonetheless always limited in quantity. For example, for an RNG relying on a user's keystrokes, it may be the case that the user has not been typing sufficiently. Waiting for more real random data to be acquired in order to receive random

6
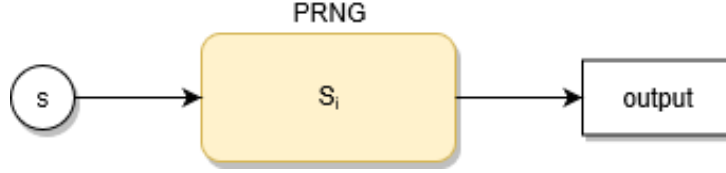
Figure 2.2: A high-level view of the operation of a PRNG [31].

numbers is not a viable option for a number of applications [25, p. 139]. Furthermore, as discussed above, it is difficult to ascertain how much entropy one is really getting from the source, not to mention that the source, in particular if implemented in hardware, may fail unexpectedly and become predictable [25].

2.1.3 Pseudo-Random Number Generators

A solution to many of the problems inherent to the use of truly random number generators is to use a *pseudo-random number generator* [25, p. 140].

Definition 6. A pseudo-random number generator (PRNG) is a deterministic algorithm with an internal state S_i [31, p. 2] which processes an input value s, known as the *seed*, to produce a number sequence that may not tractably be distinguished from a truly random sequence by statistical means [38, p. 170]. The current internal state and seed uniquely determine the PRNG's output (figure 2.2) [31, p. 2] [39, s 1.1.4].

Following from the above, this investigation defines a PRNG as an implementation of a function $prng(s) : \mathbb{R} \to \mathbb{R}^n$, where n is the length of the output sequence produced by the PRNG for some seed s. Alternatively, we can view each individual number in the output sequence as the product of some function $prng(s, S_i) : X \to \mathbb{R}$, where X is the set of all tuples (s, S_i).

Implementations of PRNGs range in complexity and efficacy from simple *linear congruential generators* [38, p. 170] to the more complex *Mersenne twister* [37]. The concepts behind their operation are not considered in this report. Note that the internal state of the PRNG may be arbitrarily complex. For example, the *ANSI X9.17 generator*, described by Menezes et al., takes further input parameters in addition to the random seed [38, p. 173]. This investigation takes the view that any such additional inputs, provided they are not random and thus do not constitute a source of entropy for the generator, can be seen as a component of the internal state.

With the limitations of truly random number generators in mind, according to Menezes at al. the purpose of a PRNG is to "take a small truly random sequence and expand it to a sequence of much larger length", in such a way that the output sequence fulfills some application-dependent randomness requirements [38, p. 170]. In the context of security, the minimum requirement is that the length l of the seed be such that it is unfeasible to carry out exhaustive search over all possible seeds. Randomness requirements for cryptographic use are considered in more detail in subsection 2.2.2.

2.2 Random Numbers and Cryptography

Random numbers are ubiquitous in computing, with applications in *simulation*, *sampling*, *numerical analysis*, *randomized algorithms*, *decision making*, *aesthetics*, and *games* [24, p. 1-2]. Perhaps most importantly, random numbers are widely used in cryptography to secure communications [25, p. 137].

The notion of randomness is crucial in cryptography, because the security guarantees of systems such as encryption and decryption algorithms often rely on an adversary not knowing the value of some internal parameter [38, p. 169] [31, p. 1]. It follows that the adversary should not have a good chance of being able to guess said parameters [31, p. 2], meaning that their values should be unpredictable.

2.2.1 Cryptographic Applications of Random Numbers

As stated earlier, pseudo-random numbers are widely applied to cryptosystems, for example in encryption and decryption algorithms [38, p. 169]. For example, the *RSA (Rivest-Shamir-Adleman) algorithm*, as explained by Anderson [14], is the canonical algorithm for performing public-key encryption and digital signatures. The RSA algorithm relies on two randomly chosen large prime numbers p and q, which act as the private keys used by the two communicating parties. As these values must be kept secret from any third parties, they need to be selected randomly in such a manner than an attacker cannot predict them [14, p. 171].

2.2.2 Cryptographically Secure Pseudo-Random Number Generators

As the outputs of a PRNG should be practically indistinguishable from truly random sequences, the basis for determining whether a PRNG is suitable for use in cryptographic applications is subjecting its outputs to a number of statistical tests [38, p. 170]. From Menezes et al. and Rukhin et al. we obtain the following definitions, formulated in terms of statistical tests.

Definition 7. A *statistical test* is a test which seeks to determine whether a sequences possesses some statistical property which is expected of a truly random sequence [38, p. 175]. A statistical test computes a specific *statistic* on the sample under evaluation [38, p. 179]. Since the properties of a truly random sequence are known a priori and can be formulated in probabilistic terms, it is possible to determine the probability of the computed statistic appearing under the assumption of randomness [39, s 1-3].

Definition 8. A *polynomial-time statistical test* is a test with a time complexity upper bound $O(l^k)$, where l is the length of the sequence being tested and k is a constant [38, p. 171].

Definition 9. A PRNG passes the *next-bit test* if no polynomial-time algorithm can predict the bit at position $n + 1$ in the output given the first n bits with probability significantly greater than 0.5. Furthermore, a PRNG passes the next-bit test if, and only if, it passes all polynomial-time statistical tests [38, p. 171].

Definition 10. A *cryptographically secure pseudo-random number generator* (CSPRNG) is a pseudo-random number generator that passes the next-bit test [38, p. 171].

Note that for real-world CSPRNGs, as with cryptosystems in general, security relies on the conjectured intractability of some underlying numerical problem [38, p. 185]. As an example, the RSA algorithm relies on the computational complexity of the integer factorization problem [14, p. 170-173]. This theoretical aspect of security is beyond the scope of this investigation and is not considered.

2.2.3 Testing Number Sequences for Randomness

Rukhin at al. point out that the degree of randomness of a number sequence can be evaluated by means of statistical hypothesis testing [39, s. 1-3].

Definition 11. According to Menezes et al., a *statistical hypothesis* is "an assertion about a distribution of one or more random variables", while a *test* is "a procedure [...] which leads to the acceptance or rejection of the hypothesis" [38, p. 179]. The probability of incorrectly rejecting a statistical hypothesis is known as the *significance level* α of the test [38, p. 179].

Rukhin et al explain that, in testing number sequences for randomness, statistical tests are formulated to test the *null hypothesis* H_0 that the sequence is random. Under the assumption of randomness, any statistical metric has a theoretical reference distribution [39, p. 1.3]. For this distribution, a *critical value* is determined, which acts as a threshold. For a measured test metric, a *p-value* is computed; this is the probability of observing the metric under the assumption of randomness. If the observed value exceeds the critical value, or equivalently, if the p-value is less than the significance level, we reject the null hypothesis. We consider a value above the critical value to be so unlikely to occur under the assumption of randomness, that we can reasonably (though not certainly) reject the randomness hypothesis [39, p. 1-3,1-4] (figure 2.3).

A simple example of a statistical randomness test, as explained by Menezes et al., is the *frequency test*, which determines the number of 1s and 0s in the sequence. For a truly random sequence, as the length of the sequence tends to infinity the ratio of the two values approaches 1. That is, we expect the two values to be approximately the same [38, p. 181]. Further to the results of any individual test, for a truly random sequence the p-values computed over all tests should be uniformly distributed. This is tested using the *Kolmogorov-Smirnov test* [39, s. 4-6].

The NIST Test Suite is the accepted standard for testing random and pseudo-random bit generators [34]. It consists of a battery of statistical tests performed on files containing large binary sequences (on the order of 10^6 bits with standard settings). Each test in the suite either accepts or rejects the null hypothesis [39]. Furthermore, The NIST suite was found to be used throughout the majority of reviewed papers on PRNGs, and is accordingly used in this investigation.
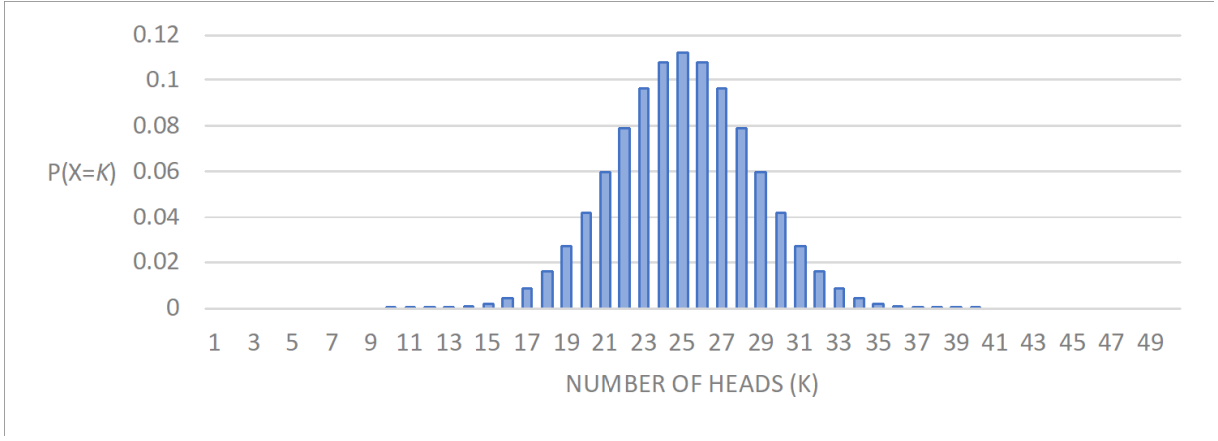
Figure 2.3: The probability distribution of a random variable representing the number of heads in 50 tosses of a fair coin. This is equivalent to the number of 1s in a truly random binary sequence of length 50. Values at the tails of the distribution, such as 2 and 48, could be chosen as critical values; farther results are so unlikely that, upon observing them, one could justifiably question whether the coin is fair.

2.3 Artificial Neural Networks

This section provides an introduction to neural networks, covering basic concepts such as the functioning of individual neurons in a network, how a neural network's parameters are updated to enable learning, and some common network topologies.

2.3.1 Introduction to Neural Networks

Definition 12. According to Russel and Norvig, an *artificial neural network* is a directed graph composed of *units* or *neurons* connected by directed *links*. Each unit computes an arbitrary function g, called the *activation function*, over a weighted sum of the unit's inputs. A link from unit i to a unit j carries the output of i's activation function to j. Each such link has an associated *weight* parameter w_{ij}, which is a coefficient applied to the activation value [40, p. 727-731]. Russel and Norvig define the function represented by a neuron j as

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{ij}a_i\right) \tag{2.2}$$

where a_j is the neuron's output, g is the neuron's activation function, and j has n input neurons i with outputs a_i and weights w_{ij} [40, p. 728].

This can also be formulated in vector form as the inner product of inputs $\boldsymbol{a}_i$ and input weights $\boldsymbol{w}_{ij}$:

$$a_j = g(\boldsymbol{w}_{ij} \cdot \boldsymbol{a}_i) \tag{2.3}$$

A neural network as a whole, as explained by Russel and Norvig, computes a highly non-linear vector function $\boldsymbol{f_w}(\boldsymbol{x})$ parameterized by its weights $\boldsymbol{w}$ [40, p. 731, 732]. The
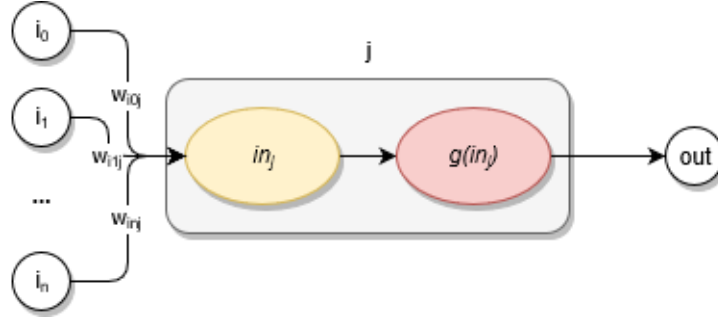
Figure 2.4: Schematic of the computation performed by a unit, drawn in accordance to the definitions given by Russel and Norvig [40, p.728]. A weighted sum in_j of the unit's inputs is computed, and the resulting value becomes the argument of the activation function g. The activation's output is the node's output. This schematic corresponds to equation 2.2.

function $\boldsymbol{f}$ is the composition of i functions $\boldsymbol{f}^{(i)}$ [27, p. 164], which can be expressed as

$$\boldsymbol{f_w}(\boldsymbol{x}) = \boldsymbol{f}_{\boldsymbol{w}_{i-1}}^{(i-1)}(\boldsymbol{f}_{\boldsymbol{w}_{i-2}}^{(i-2)}(\dots \boldsymbol{f}_{\boldsymbol{w}_0}^{(0)}(\boldsymbol{x}))) \tag{2.4}$$

where $\boldsymbol{w_i}$ is the output weight vector for the units computing the function $\boldsymbol{f}_{\boldsymbol{w}_i}^{(i)}$ [27].

According to Russel and Norvig, *a neural network of sufficient size can represent any continuous function to an arbitrary degree of accuracy*, and, under certain conditions, can even represent discontinuous functions. However, difficulties arise in determining, for any particular network architecture, the set of functions it can represent [40, p. 732]. The particular properties of the network are determined by the topology and behavior of its units, which form the basis for the naming and categorization of networks. In general, we distinguish between *feed-forward networks* and *recurrent networks*; the former are topologically acyclic graphs, while the latter contains loops [40, p. 729].

2.3.2 Learning in Neural Networks

Definition 13. *Training* an artificial neural network is an optimization problem which entails searching for a combination parameters $\boldsymbol{w}$ which results in the network approximating the desired function $f(x)$ as closely as possible [40, p. 718]. The most important related concepts are *loss functions*, *gradient descent*, and *backpropagation*.

Definition 14. Russel and Norvig define loss as "the amount of utility lost" by the network producing $f_{pred}(x) = y'$ as its output when the correct output on an input x is $f_{true}(x) = y$. A neural network's loss function $L(y, y')$ computes the loss for an output y' and its corresponding correct output y' [40]. Intuitively, a loss function provides a quantitative assessment of how closely the neural network approximates the desired function (a good model will have low loss). The objective of learning is to find the parameters which minimize the loss function [29, Linear classification: Support Vector Machine, Softmax].

Definition 15. Gradient descent is a numerical optimization algorithm used to train neural networks, in which the parameters of the network are iteratively updated into the direction opposite to the loss function's first-order gradient (computed with respect to the parameters) [29, Optimization: Stochastic Gradient Descent]. The magnitude of the gradient updates is scaled by a coefficient known as the *learning rate* (or step size)

[29, Optimization: Stochastic Gradient Descent]. Gradient descent is ubiquitous in the optimization of neural network loss functions. A high-level expression of gradient descent, from [29], is as follows:

```
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # parameter update
```

An important design choice, according to Karpathy, is the number of inputs over which the loss is computed in order to perform a single update to the network's parameters. In *batch gradient descent*, the loss is computed as an average of the loss for each input in the entire training dataset; in *mini-batch gradient descent* the dataset is split into subsets, and a single parameter update is performed using each subset [29, Optimization: Stochastic Gradient Descent]. In general, the larger the batch size, the more "stable" the loss function is over training iterations, as the impact of single network inputs is diminished [29, Neural Networks Part 3: Learning and Evaluation]. A single pass over the training dataset is referred to as an *epoch* [29, Neural Networks Part 3: Learning and Evaluation].

Definition 16. Lastly, backpropagation is an efficient algorithm for computing the partial derivative of a function of many variables by repeated application of the chain rule of derivation [29, Backpropagation, Intuitions]. It is crucial as it enables efficient computation of the gradient of the loss function with respect to each parameter in the neural network [29, Backpropagation, Intuitions]. Backpropagation is a core component of all modern machine learning software libraries.

These three components enable learning in neural networks: the loss function quantifies the quality of the current parameters, gradient descent is the general optimization algorithm for modifying the parameters, and backpropagation enables efficient computation of gradients, making gradient descent on large networks practically feasible [29, Optimization: Stochastic Gradient Descent]. A typical training session for a neural network is characterized by a loss function evolving over time as shown in figure 2.5.

2.3.3   Activation Functions

The concept of an *activation function* was briefly introduced in 2.3.1, as an arbitrary scalar function applied to the weighted sum of a unit's inputs to produce the unit's output. There are a number of standard activation functions that are commonly used [29, Neural Networks Part 1: Setting up the Architecture], of which two are considered here.

Definition 17. The *rectified linear unit* activation, or ReLU, is an activation function $ReLU(x) : \mathbb{R} \to \mathbb{R}^+$ with the following expression form:

$$ReLU(x) = max(0, x) \tag{2.5}$$

.
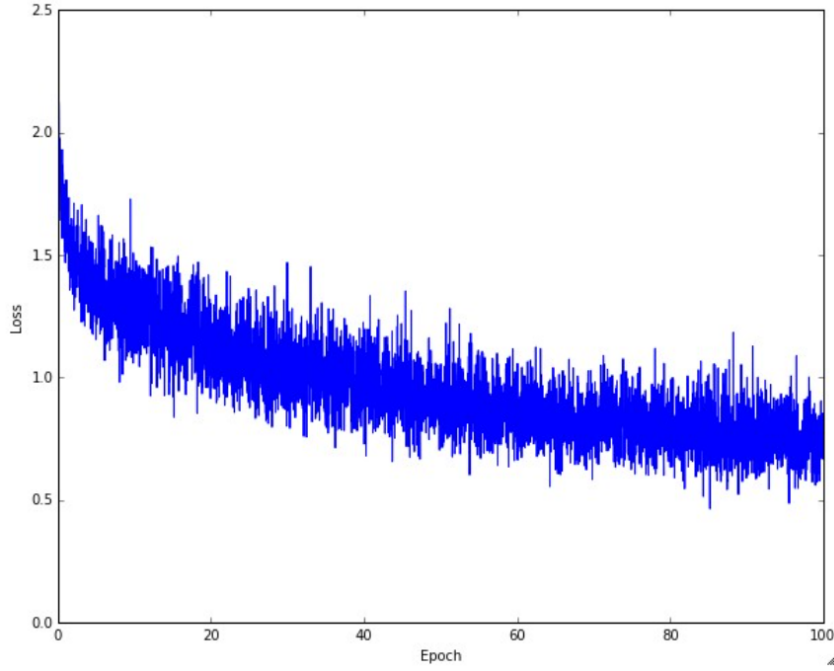
Figure 2.5: A typical neural network loss function evolution during training. Image courtesy of [29, Neural Networks Part 3: Learning and Evaluation].

The ReLU activation function works well in practice on a large number of problems [29, Neural Networks Part 1: Setting up the Architecture]. However, it is possible for a ReLU unit to have its weights updated in a way that causes them to never be updated again, due to the 0 gradient for all negative inputs [29, Neural Networks Part 1: Setting up the Architecture].

Definition 18. The *leaky ReLU* function is an activation function $LReLU(x) : \mathbb{R} \to \mathbb{R}$ with the following expression form:

$$LeakyReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.6}$$

where α is a (small) non-zero constant. For negative inputs close to 0, the outputs of the leaky ReLU are approximately equal to the outputs of ReLU. However, a leaky ReLU unit does not have a zero-gradient for any input, and thus cannot reach a state in which it stops updating, the way a ReLU unit would[29, Neural Networks Part 1: Setting up the Architecture].

2.3.4 Fully Connected Feed-Forward Networks

Definition 19. The simplest form of deep neural network is called a *fully connected feed-forward neural network*, or *multilayer perceptron*. A fully connected feed-forward (FCFF) network is a directed acyclic graph in which information strictly flows from the network's input units towards its output units [27, p. 164]. Feed-forward networks are typically arranged into fully connected *layers* of units, where the number of layers is called the *depth* of the network; the first layer is the *input layer*, followed by a number of *hidden layers* and finally the *output layer* [27, p. 164-165]. The number of units in the largest layer is referred to as the *width* of the network (figure 2.7) [27, p. 164-165].
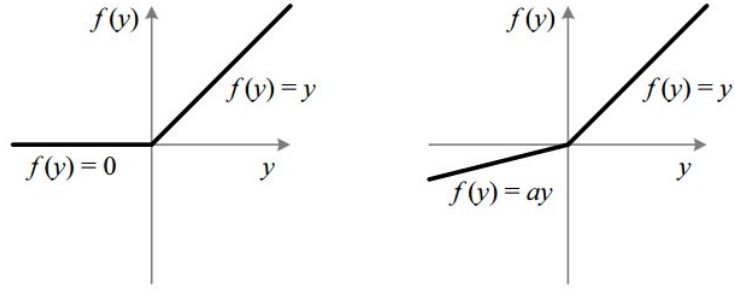
Figure 2.6: Left: the ReLU function. Right: the LeakyReLU function. Note the zero gradient for negative inputs on the left, and how this problem is addressed on the right. Image courtesy of [41].
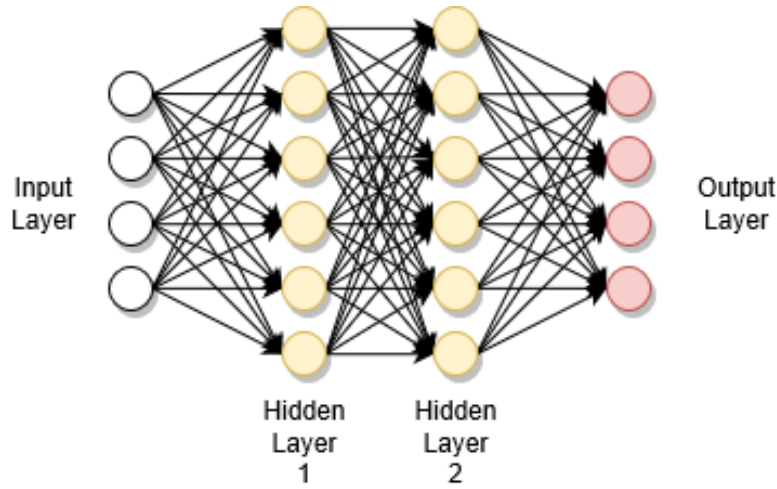
Figure 2.7: Simple example of the structure of a feedforward network. The network has two hidden layers, for a total depth of 4. The network's width is 6, as given by the size of the hidden layers. Image drawn using draw.io [36]

The number of units in the input and output layers are bound by the dimensionality of the input data and the dimensionality of the expected outputs. In a fully connected feed-forward network, the operation of each layer can be characterized as a simple matrix operation, whereby a layer's output vector is matrix multiplied with the layer's output weight matrix, and then passed to the next layer where the activation function is applied [27, p. 170-171]. It follows that the network as a whole is a sequence of matrix multiplications and element-wise applications of non-linearity.

2.3.5 Convolutional Neural Networks

Definition 20. A *convolutional neural network* is a more specialized neural network architecture characterized by the fact that at least one pair of layers is connected in such a manner as to perform a *convolution* rather than a general matrix multiplication [27, p. 326].

Definition 21. According to Karpathy [29], a *convolutional layer* is a collection of units further subdivided into equally sized groups referred to as *filters*. Each unit in a filter receives inputs from k consecutive units in the previous layer, where k is referred to as
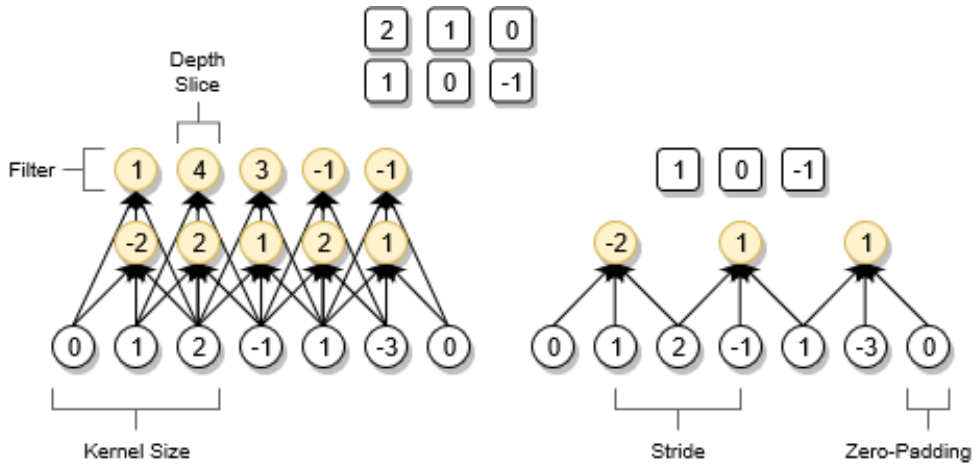
Figure 2.8: A representation of the connectivity between the units of a 1-dimensional input layer (white) and the filters in a convolutional layer (yellow), with each convolutional unit's input weights shown in the squares. In both images, the kernel size is 3; on the left the stride of the layer is 1, while on the right it is 2. Furthermore, the convolutional layer on the left has two filters. Note how both filters are connected to the input in the same way, and how each filter has its own set of input weights. The image is based on the CS231n lecture notes by Andrej Karpathy [29, Convolutional Neural Networks: Architectures, Convolution / Pooling Layers] and was drawn with draw.io [36].

the *kernel size* or *receptive field* of the convolutional layer. The *stride s* of the layer determines the sparsity of the units in each filter (figure 2.8). A convolutional layer may have multiple filters, each with the same connection topology to the previous layer. The number of filters is referred to as the *depth* of the layer, while the dimensions of each individual filter are referred to as the *width* and *height* of the layer [29, Convolutional Neural Networks: Architectures, Convolution / Pooling Layers].

All units in a filter share the same input weights (an optimization called *parameter sharing*). Thus every unit in the filter computes the same function on a specific subset of the previous layer's outputs. Intuitively, the filter as a whole "looks for" specific instances of some pattern in the input; a stack of n filters looks for instances of n different patterns (figure 2.8) [29, Convolutional Neural Networks: Architectures, Convolution / Pooling Layers].

Karpathy explains that the dimensionality of a convolutional layer's output is $d + 1$, where d is the dimensionality of the output of the previous layer: with N filters, the convolutional layer outputs N matrices with the same dimensionality as the layer's input. Because of this, convolutional layers are usually followed by *pooling layers*, which are non-parametric layers that down-sample the convolutional layer's outputs [29, Convolutional Neural Networks: Architectures, Convolution / Pooling Layers]. This expansion along the depth-dimension, followed by down-sampling in the width and height dimensions, is shown in figure 2.9.
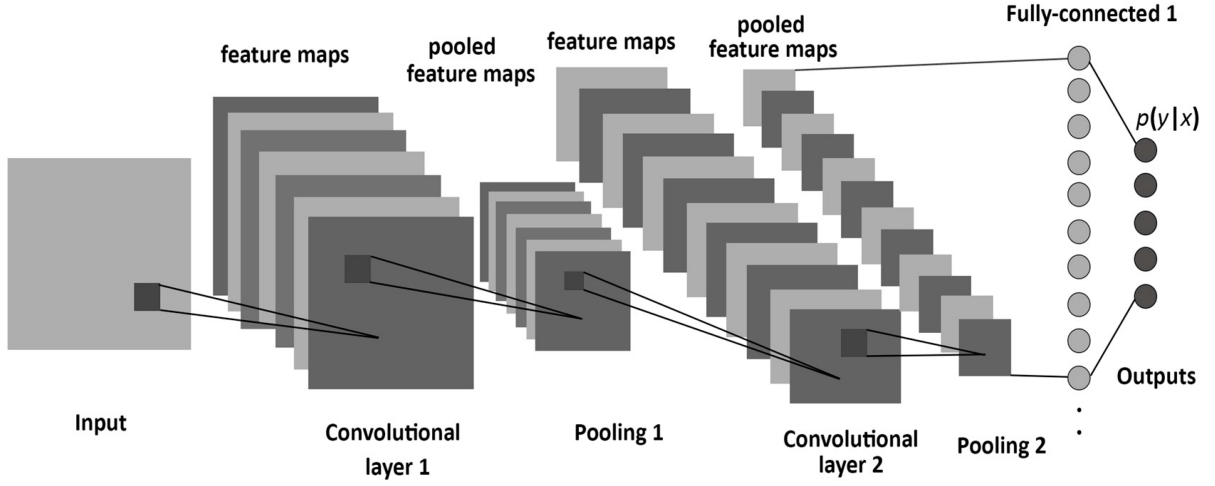
Figure 2.9: A convolutional layer preserves the dimensions of its input matrix, but produces an output with a larger depth dimensions depending on the number of filters. This is followed by pooling layers, which down-sample the data in the width and height dimensions. Image courtesy of [12].

2.3.6 Generative Adversarial Networks

Machine learning models may be classified as being either *discriminative* or *generative*. An example of discriminative models is neural networks that map images to content labels [28, p. 1]. Generative models, on the other hand, learn to mimic the training set. Goodfellow et al introduced *generative adversarial networks* (GANs) in 2014, succinctly defining them as a "framework for estimating generative models via an adversarial process", where a discriminative model is used to train a generative model by scrutinizing its outputs [28, p. 1].

Definition 22. Let Z be a random variable with any distribution, and let T be a random variable representing a dataset with a probability distribution p_{data}. A *generative model* learns a mapping $gen(z) : Z \to M$, where M is a random variable with a distribution p_{model} that approximates p_{data} [26].

Definition 23. Let Z be a random variable with any distribution, and let T be a random variable representing a dataset with a probability distribution p_{data}. A generative adversarial network (GAN) consists of two artificial neural networks, a *generator* G and a *discriminator* D. The generator represents a function $G_{\theta_g}(\boldsymbol{z}) : Z \to M$ parameterized by weights θ_g, which maps inputs $\boldsymbol{z} \in Z$ to an output $\boldsymbol{x}$ in M p_{model}. The discriminator represents a function $D_{\theta_d}(\boldsymbol{x}) : T \cap M \to \mathbb{R}$ that outputs a scalar representing the probability that $\boldsymbol{x}$ was sampled from T rather than produced by the generator (figure 2.10) [28, p. 2-3].

The discriminator is trained to maximize the probability of assigning the correct label to inputs sampled from T and generated by G. In turn, the generator is trained to minimize $\log 1 - D(G(\boldsymbol{z}))$. Goodfellow et all [28, p. 3] showed that this is equivalent to saying that, during training, D and G engage in a two-player mini-max game with value function $V(G, D)$, formulated as follows:

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \ p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \ p_{\boldsymbol{z}}(\boldsymbol{z})}[\log 1 - D(G(\boldsymbol{z}))] \qquad (2.7)$$
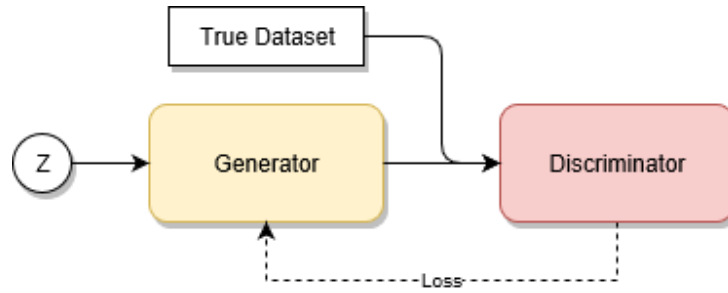
Figure 2.10: The conceptual structure of a generative adversarial network. The outputs of the generator are fed into the discriminator along with samples from the original dataset; the performance of the discriminator is factored into the loss function of the generator. Image drawn with draw.io [36].

A mini-batch stochastic gradient descent training algorithm for a GAN, as given by Goodfellow et al but slightly simplified, is shown below.

```
for i in range(training_iterations):
  for k in range(steps):
    sample mini-batch from data distribution
    sample mini-batch from generator
    update discriminator by gradient descent
  sample mini-batch of noise from noise distribution
  update generator by gradient descent
```

Goodfellow et all also showed that, provided G and D have sufficient capacity, and at each step of the above algorithm D is allowed to reach its optimum given the current G, then p_{model} converges to p_{data} [28, p. 5].

2.4 Related Work

Literature review found that few efforts have been made to train neural networks to act as pseudo-random number generators. In this section, the findings of some key relevant papers are reviewed.

2.4.1 Papers on Using Neural Networks as Pseudo-Random Number Generators

Overall, the literature review carried out for this investigation showed that little research has been carried out on the implementation of PRNGs using neural networks. A small number of such papers was identified, but all were relatively obscure, possibly due to reliance on complex neural network architectures and contrived training procedures in an attempt to encourage chaotic behavior [22] [23] [43] [44].

For example, a 2012 paper by Veena Desai et al from the Gogte Institute of Technology, *Pseudo random number generator using time delay neural network*, failed to produce

a viable neural network-based PRNG, and identified the computational complexity of training networks with thousands of neurons as one of the key challenges [23].

Other publications include Desai's earlier 2011 paper *Pseudo random number generator using Elman neural network* and the 2010 paper *Hopfield Neural Networks as Pseudo-Random Number Generators* by Tirdad and Sadeghian. In all cases the conclusions were mixed. Desai's paper reported "satisfactory results" on a subset of the performed statistical tests, with unexplained particularly poor performance on one of the tests [22]. Tirdad and Sadeghian achieved a strong performance on the NIST test suite in some of the reported experiments [43]. However, the opinion of the author of this report is that their implementation is contrived and exceedingly complex.

2.4.2 Learning to Protect Communications with Adversarial Neural Cryptography

The 2016 paper by Google Brain researches Martin Abadi and David Andersen investigates the ability of neural networks to learn some form of symmetric-key encryption scheme in a multi-agent environment, enabling some agents to communicate securely. The paper's abstract states:

"We ask whether neural networks can learn to use secret keys to protect information from other neural networks. Specially, we focus on ensuring condentiality properties in a multiagent system, and we specify those properties in terms of an adversary. Thus, a system may consist of neural networks named Alice and Bob, and we aim to limit what a third neural network named Eve learns from eavesdropping on the communication between Alice and Bob. We do not prescribe specic cryptographic algorithms to these neural networks; instead, we train end-to-end, adversarially. We demonstrate that the neural networks can learn how to perform forms of encryption and decryption, and also how to apply these operations selectively in order to meet condentiality goals." [11]

The paper's conclusion mentions pseudo-random number generation as a possible avenue of further investigation, providing the original inspiration behind this work.

3. Design and Implementation

This section provides an in-depth explanation of the conceptual design of the system and how it relates to the research hypothesis, the technologies used to implement it, and the details of the implementation.

3.1 Conceptual Design

As stated, the aim of this investigation is to determine whether it is possible to train a neural network to output pseudo-random number sequences. Previous work has attempted to answer this question by using recurrent architectures to encourage chaotic behavior (see section 2.4). In contrast, this investigation takes a more intuitive approach, conjecturing that a simple fully-connected feed-forward network of sufficient size can learn a function whose inputs appear random.

For simplicity, we view a pseudo-random number generator as a system implementing some function

$$prng(s) : \mathbb{R} \to \mathbb{R}^n \tag{3.1}$$

where s is a truly random seed value, n is a very large value, and the outputs of $prng$ fulfill a set of criteria for randomness. In regards to each individual output value, we can characterize a PRNG as a function

$$prng(s, S_i) : X \to \mathbb{R} \tag{3.2}$$

where S_i is the current state of the generator, and X is the set of all tuples (s, S_i). That is, a PRNG is fundamentally a function which maps a single seed value to a very large (unique) output sequence, such each element of the sequence is determined by the generator's state. A generator neural network, G, should learn a function which approximates $prng$.

We can abstract the complexity of the internal state of PRNGs by using a stateless neural network, and equivalently representing the generator's "state" as a component of the network's input instead. This conceptual difference is demonstrated in figure 3.1. Thus the neural network should learn a function

$$G_{\theta_G}(s, o_t) : \mathbb{R}^i \to \mathbb{R}^n \tag{3.3}$$

where s is a truly random seed value, o_t model's a PRNG's internal state and can be seen as an "offset" into the full output sequence for s, and i is the dimensionality of the network input.
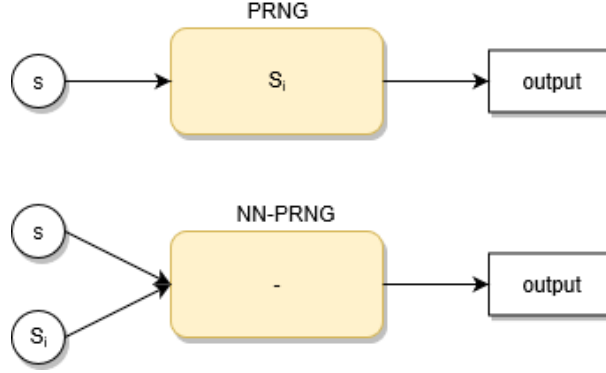
Figure 3.1: The conceptual difference in the common implementation of PRNGs (top) and the implementation using GANs in this work (bottom). Image produced using draw.io [36].

This work views the generation of pseudo-random numbers in a cryptographic setting as a fundamentally adversarial task: the goal of good CSPRNG design is to minimize the probability of an intelligent adversary correctly guessing future outputs. This is analogous to the generative adversarial network framework, where a discriminator attempts to find patterns in the generator's output, and the generator minimizes its probability of doing so correctly. Thus it is natural to formulate the task of generating pseudo-random number sequences using a GAN. Two distinct approaches are considered, termed the *discriminative approach* and the *predictive approach*, respectively (figure 3.2).

In the discriminative approach, the adversary is a standard discriminator network, which receives number sequences as inputs both from the generator and a source of true randomness, and outputs the probability that a sequence is truly random. The input sequences are labeled as truly random or not random, and the discriminator is trained to better discern the two classes. In order to prevent the discriminator from performing better than it would by making random guesses, the generator has to learn to mimic the truly random sequences.

The predictive approach is loosely based on the idea of the theoretical next bit test, outlined in section 2.2.2. Each sequence of length n produced by the generator is split, such that the first $n - 1$ values are passed to the adversary as input, and the nth value is used as the corresponding label; the predictor is trained to output the nth value in the generator's output sequence based on all previous values. Again, the generator's goal is to modify its behavior in order to minimize the probability of the predictor making a correct guess.

Both approaches are elegantly intuitive. The former is a direct application of the standard GAN framework, which requires the generator to learn the uniform probability distribution characteristic of truly random number sequences. The latter models closely the actual goals of a PRNG and its adversary in a cryptographic setting. Unlike previous work, this investigation follows an important guideline given by Russel and Norvig: *"As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave"* [40, p. 37].
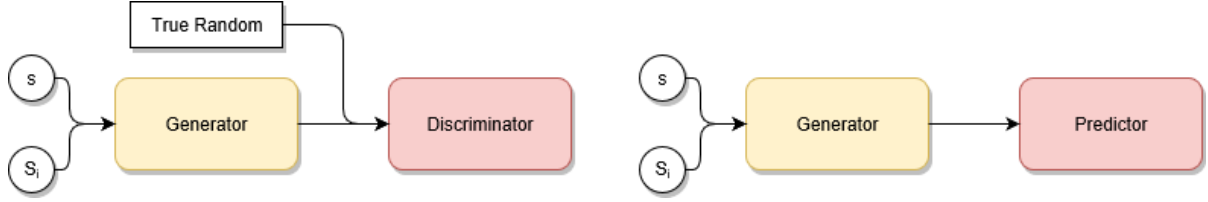
Figure 3.2: The two approaches differ at the conceptual level in the the discriminative approach (left) requires a source of true randomness which it attempts to emulate, while the predictive approach (right) is an even purer "game" between the two networks, with no side inputs. Image produced using draw.io [36].
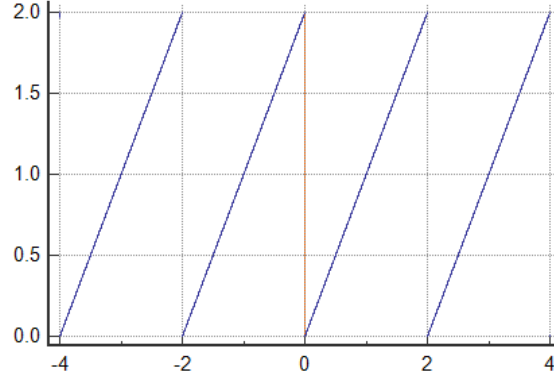
Figure 3.3: Example of the graph of the function $f(x) = x \bmod 2$. Image courtesy of [18].

3.1.1 Generative Model

The generator is a fully connected feed-forward neural network implementing the function $G_{\theta_G}(s, o_t) : \mathbb{R}^2 \to \mathbb{R}^m$. It takes two input values, where the first is a truly random seed, and the second is a non-random input representing the PRNG state; the output is a 1D real vector of length m. For any specific seed s, the complete pseudo-random sequence produced by the generator is given by the concatenation of all the output sequences $\forall o_t \; G_{\theta_G}(s, o_t)$, where s is fixed.

The generator consists of five fully connected feed-forward layers (figure 3.4). The input layer, as well as the hidden layers, use the leaky ReLU activation function which is currently the general recommendation for generic application [29, Neural Networks Part 1: Setting up the Architecture]. The output layer uses an activation function which computes a modulo operation on every element in the output vector, squeezing the values into a desired range. This is not a standard activation function, and constitutes one of the key innovations of this work. Intuitively, the modulo activation confounds any patterns otherwise present in the network's output (see figure 3.3). A popular variant of the stochastic gradient descent algorithm called Adam is used, which adaptively computes separate learning rates for each parameter [32] [29, Optimization: Stochastic Gradient Descent].

Two slightly different variants of the generator are implemented. The former uses a "squared difference" loss function and is trained using the discriminative approach, while the latter uses an "absolute difference" loss function and is trained using the predictive approach. The two implementations are identical otherwise.
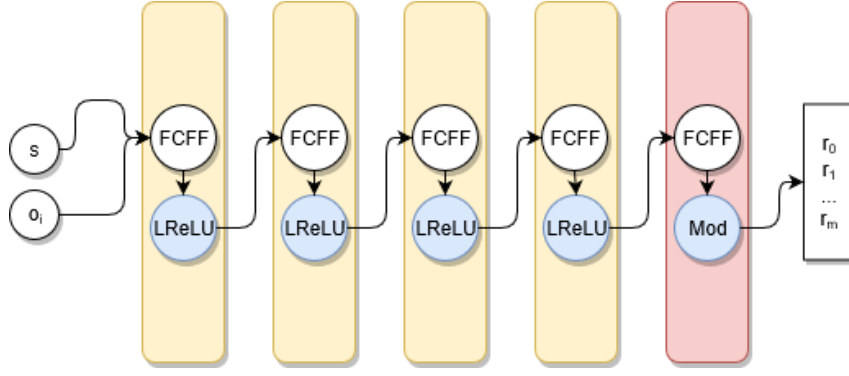
Figure 3.4: Architecture of the generator. Each fully connected feed-forward layer's activation function (blue) is shown; the output layer (red) differs from the other layers in that it does not use the leaky ReLU activation, but rather a modulus function. Image produced using draw.io [36].

3.1.2 Discriminative Model

The discriminator (figure 3.5) is convolutional neural network implementing the function

$$disc(\boldsymbol{r}) : \mathbb{R}^m \to [0, 1] \tag{3.4}$$

where $\boldsymbol{r}$ is the generator's output vector and m is the output vector's size. The discriminator takes as inputs sequences of length m and outputs a scalar $p(true)$ in the range $[0, 1]$ representing the probability the sequence is truly random.

The discriminator consists of four stacked convolutional layers, followed by a pooling layer and two fully connected feed-forward layers. As explained in section 2.3.5, a single convolutional layer with multiple filters "searches" the previous layer's output for occurrences of specific patterns. Multiple stacked convolutional layers allow the network to search for "patterns of patterns", giving the discriminator the ability to detect complex correlations in its inputs, if there are any. According to Karpathy, stacking multiple convolutional layers is preferable to increasing the number of filters or the kernel size of a single layer [Convolutional Neural Networks: Architectures, Convolution / Pooling Layers][29].

The convolutional layers are followed by a pooling layer to reduce the dimensionality of the convolutional layers' outputs, and fully connected layers to gradually squeeze the data down to a single scalar. Overall, the discriminator's architecture is designed to enable it to learn to detect patterns in its input, and to compute a function of that detection information that reduces it to a single probability value.

3.1.3 Predictive Model

The predictor (figure 3.6) is a convolutional neural network implementing the function

$$pred(\boldsymbol{r}_{split}) : \mathbb{R}^{m-1} \to \mathbb{R} \tag{3.5}$$

where $\boldsymbol{r}$ is the generator's output vector, $\boldsymbol{r}_{split}$ is the output vector without the last element, and m is the output vector's size. The predictor's inputs are sequences of length $m-1$ produced by the generator, which the predictor attempts to map to the mth value in the sequence.
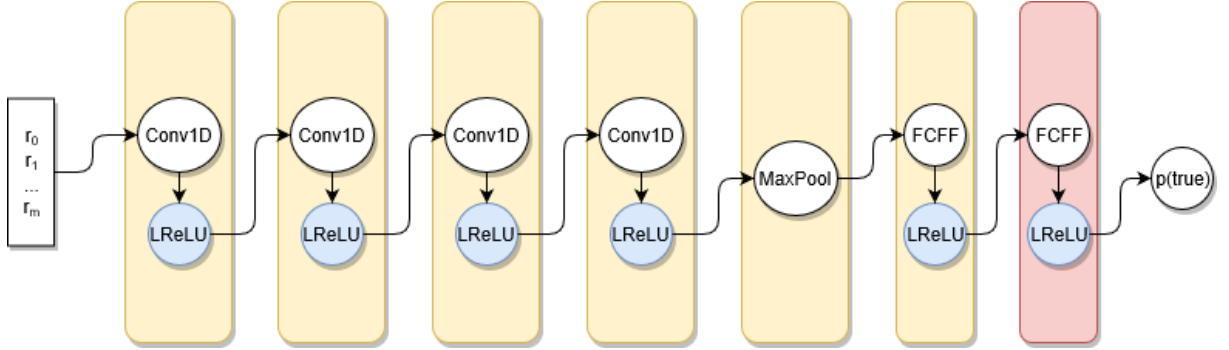
Figure 3.5: Convolutional discriminator architecture. The output of the generator is convolved multiple times in order to extract higher-level features from the sequence; this is followed by pooling to reduce the output size, and fully connected feed-forward layers to produce the final classification output. Image produced using draw.io [36].
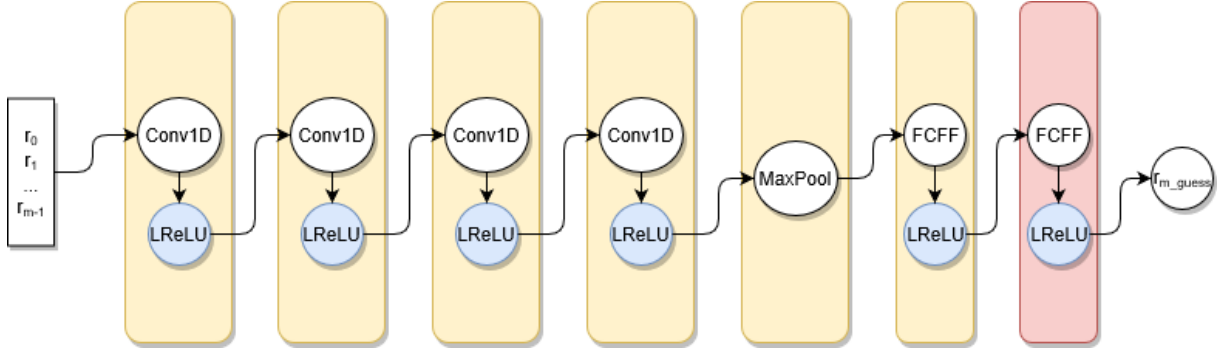
Figure 3.6: Convolutional predictor architecture. The design is shared with the discriminator, but the size of the input sequence and the meaning of the output scalar are different. The predictor produces a guess for the last value in the generator's output based on the previous values. Image produced using draw.io [36].

Apart from the input size and the meaning of the output, the discriminator and the predictor share the same architecture. The rationale behind the design choices for the predictor are the same as those for the discriminator.

3.2   Implementation Technologies

The system was developed using version 3.6 of the Python programming language, which is popular in the machine learning field due to its conciseness and the large ecosystem of software libraries for numerical computing [5]. In addition to NumPy, which is ubiquitous in numerical computation using Python [5], the main software library used for this project is TensorFlow. The latter is an "open source library for numerical computation" released by Google, which is commonly used for machine learning [7]. Most of the code in the implementation deals with abstractions defined in the TensorFlow API.

While Python and the TensorFlow library are popular in the machine learning community, alternatives abound. Programming languages considered for the project included Java and C++, both of which have popular deep learning libraries. Furthermore, TensorFlow is not the only machine learning library available for Python: options include Theano, PyTorch, CNTK, and scikit-learn.

Choosing a language, a software library, or a software framework for a particular task is not always straightforward. In this case, Python and TensorFlow were selected due to Python's legibility and simplicity, which supports rapid prototyping, as well as Tensor-Flow's popularity in the machine learning field, which ensures the availability of abundant documentation, tutorials, and more [3]. The use of these particular technologies is not a critical aspect of this research.

3.2.1   Introduction to TensorFlow

An introductory guide to the central concepts and functionality of TensorFlow is provided on the software library's website [9]. The TensorFlow version used in this project is 1.6.

The central concept is that of the *tensor*, an n-dimensional generalization of vectors and matrices. Tensors have properties such as *rank* and *shape*, which specify the number of dimensions of a tensor as well as its size in each dimension [9]. TensorFlow programs perform a series of operations on tensors.

At the highest level of abstraction, a TensorFlow program consists of a *Computational Graph* and a *Session*. The former is a directed graph consisting of tensor operations applied to a set of inputs, and defines the topology of a machine learning model. The operations specified in the graph are not computed by the Python interpreter, but rather are first compiled and then executed in TensorFlow's C++ runtime. The Python application interface to the runtime is defined by a TensorFlow Session [8]. When interacting with a session, the act of passing specific inputs to a model for a particular session is referred to as *feeding*, while retrieving the value of a tensor in the graph at a particular time during execution is referred to as *fetching* [9].

The package `tf.contrib.gan` in TensorFlow's Python API enables the programmer to easily define generative adversarial networks.

3.2.2 Supporting Technologies

Development and deployment required a mix of free as well as proprietary supporting tools. Most notable are the distributed software version control system `git` [2], and the popular containerization tool `Docker` [1] for deployment.

Software used also includes the Jetbrains PyCharm integrated development environment [4], and Microsoft's free code editor VSCode. Early tests of the implementation's performance were executed on virtual machines in Google's, Amazon's, and Digital Ocean's cloud services, with a mix of CPU-focused and GPU-focused instance types.

Training of the model was carried out on an HPC cluster, access to which was provided by the School of Electronic Engineering and Computer Science at Queen Mary University of London. The cluster consists of 5 machines, each with 64 CPU cores, 256 GB RAM, and no GPUs. The code was executed within a `Docker` container.

3.3 Software Design

This section outlines the structure of the software implementing the concepts from section 3.1. The code broadly follows the examples provided in TensorFlow's tutorials [10] [42]. The structure of the implementation is as follows:

```
> adversarial-csprng
    > src
        > components
            __init__.py
            activations.py
            inputs.py
            operations.py
        > utils
            __init__.py
            debug.py
            files.py
            operations.py
            visualize.py
        decode_nist.py
        main.py
    .dockerignore
    .gitignore
    Dockerfile
    LICENSE.md
    nist.sh
    readme.md
```

```
requirements.txt
transfer.sh
```

The files in the root directory include a `.gitignore` specifying which files in the project should not be version-controlled by `git`, a `Dockerfile` specifying the build procedure for the Docker container used to run the system, a `.dockerignore` specifying which files should be ignored by the Docker build, a convenient `readme`, a `requirements` file listing the project's Python dependencies, and two shell scripts to automate the retrieval of evaluation results from a remote compute cluster.

The `src` package contains all Python software modules. At its root are the `main.py` module, which is the software's intended entry point and contains the implementation's core, and the `decode_nist.py` utility for converting text files to a format appropriate for the NIST test suite. The `components` package contains modules defining custom TensorFlow functions used as components of the TensorFlow computational graph. Lastly, the `utils` package contains a number of modules defining utility functions broadly categorized by functionality. These are not considered in the report, but are available in the supporting documentation.

The code is structured in a procedural manner (as opposed to object-oriented). The rationale for this design choice is based on a number of factors. Firstly, localizing all of the core logic into `main.py` improves flexibility in prototyping and experimenting: as most of the core logic is prone to frequent change, bundling it into a single module improves productivity by reducing the amount of time spent looking for particular implementation details. In other words, in a project such as this, *low-level details matter at the high level*, and abstracting them into object-oriented modules is inconvenient. Furthermore, the procedural approach is the convention of the field. Following convention facilitates communication.

3.3.1   System Parameterization

Several aspects of the systems are parameterized to provide ease of modification in the prototyping phase. In some cases simple command-line arguments are also supported, simplifying the running of experiments with different training setups.

The most important parameter specifiable via command-line arguments is `HPC_TRAIN`, which specifies whether the model is being trained on the HPC cluster or tested during development. In testing mode, the scale of the model and the duration of training are drastically reduced, allowing a rapid test of correct execution of the system. Other parameters specifiable via the command line are the networks' learning rate and the number of training iterations.

```python
# main settings
# set to true when training on HPC to collect data
HPC_TRAIN = '-t' not in sys.argv
LEARN_LEVEL = 2 if '-highlr' in sys.argv else 0 if '-lowlr' in sys.argv else 1

# hyper-parameters
OUTPUT_SIZE = 8
```

```
MAX_VAL = 65535
OUTPUT_BITS = 16
BATCH_SIZE = 2046 if HPC_TRAIN else 10
LEARNING_RATE = {2: 0.1, 1: 0.02, 0: 0.008}[LEARN_LEVEL]
GEN_WIDTH = 30 if HPC_TRAIN else 10
DATA_TYPE = tf.float64

# training settings
TRAIN = ['-nodisc' not in sys.argv, '-nopred' not in sys.argv]
STEPS = 1000000 if '-long' in sys.argv else 150000 if HPC_TRAIN else 40
PRE_STEPS = 100 if HPC_TRAIN else 5
ADV_MULT = 3
```

3.3.2 Generative Model

The generator G has the same implementation for both the discriminative approach,
where it is referred to as `jerry`, and the predictive approach, where it is referred to as
`janice`. The model's architecture is defined in `main.py` using TensorFlow as follows:

```
input_layer = tf.reshape(noise, [-1, 2])
outputs = fully_connected(input_layer, GEN_WIDTH, activation=leaky_relu)
outputs = fully_connected(outputs, GEN_WIDTH, activation=leaky_relu)
outputs = fully_connected(outputs, GEN_WIDTH, activation=leaky_relu)
outputs = fully_connected(outputs, GEN_WIDTH, activation=leaky_relu)
outputs = fully_connected(outputs, OUTPUT_SIZE, activation=modulo(MAX_VAL))
return outputs
```

`GEN_WIDTH` parameterizes the number of units in the hidden layers. The `reshape` opera-
tion is used to cast input tensors to an appropriate size. The second argument specifies
the shape to which input tensors are cast, where the first value is the tensor's size in
the *batch dimension*, and any following values represent the dimensionality of the input
samples. The batch dimension refers to the number of input samples in a single batch.
In TensorFlow `-1` is a special flag signifying dynamic batch size, which allows the model
to process its inputs in batches of any size, rather than a fixed size. The output layer's
`modulo` activation function is detailed in subsection 3.3.4.

3.3.3 Discriminative and Predictive Models

The discriminator D, `diego`, and the predictor P, `priya`, share the same architecture with
the exception of the width of the input layer (`OUTPUT_SIZE` for `diego`, `OUTPUT_SIZE - 1`
for `priya`). This is explained in section 3.1. The convolutional architecture of the adver-
saries is implemented in `main.py` as follows:

```
input_layer = tf.reshape(inputs, [-1, size])
outputs = tf.expand_dims(input_layer, 2)
outputs = conv1d(outputs, filters=4, kernel_size=2, strides=1,
                 padding='same', activation=leaky_relu)
```

```
outputs = conv1d(outputs, filters=4, kernel_size=2, strides=1,
                 padding='same', activation=leaky_relu)
outputs = conv1d(outputs, filters=4, kernel_size=2, strides=1,
                 padding='same', activation=leaky_relu)
outputs = conv1d(outputs, filters=4, kernel_size=2, strides=1,
                 padding='same', activation=leaky_relu)
outputs = max_pooling1d(outputs, pool_size=2, strides=1)
outputs = flatten(outputs)
outputs = fully_connected(outputs, 4, activation=leaky_relu)
outputs = fully_connected(outputs, 1, activation=leaky_relu)
return outputs
```

The `flatten` and `reshape` operations are used to modify the shape of tensors in order to connect layers with different shapes. Unlike the generator, where a custom activation function is used in order to introduce greater non-linearity, all layers in the adversary models use the popular leaky rectified linear unit activation.

3.3.4   Custom Tensor Operations

The `components` package contains modules implementing neural network functionality that is not a part of the TensorFlow API, including the custom activation function `modulo`.

`modulo(divisor, withActivation=None)` is a closure returning a custom tensor function that computes the element-wise modulus   mod *divisor* of its input tensor. Using a closure allows the creation of a function object with the desired parameters, which can then be passed to the TensorFlow API. The `modulo` function can additionally wrap another activation function, though this feature is not used.

3.3.5   Obtaining Training Data

Input data for training and evaluating the networks is generated by functions defined in the `components.input` module. This includes two functions for producing a training dataset which return equivalent data structures, with the difference that one returns it as a NumPy array and the other as a TensorFlow tensor. The module also defines a function that produces an input dataset as a NumPy array which is used to evaluate the models.

A training dataset is produced as follows (the NumPy version is identical in structure as the TensorFlow API and NumPy API share many design aspects):

```
def get_input_tensor(batch_size, max_val) -> tf.Tensor:
    return tf.transpose(
        tf.stack(
            [tf.fill([batch_size], tf.random_uniform(shape=[], minval=0,
                    maxval=max_val)),
             tf.random_uniform(shape=[batch_size], minval=0,
                    maxval=batch_size)],
        ))
```

The evaluation dataset is generated as follows:

```python
def get_eval_input_numpy(seed, length, batch_size) -> np.ndarray:
    data = []
    offset = 0

    for batch_num in range(length):
        batch = []
        for item in range(batch_size):
            batch.append([seed, offset])
            offset = offset + 1
        data.append(batch)

    return np.array(data)
```

Documentation strings were removed for brevity.

3.3.6 Defining and Training the Discriminative GAN

The discriminative GAN is defined using TensorFlow's `tf.contrib.gan` package, which simplifies the implementation of a standard GAN consisting of a generator and discriminator. The GAN is structured as follows:

```python
# build the GAN model
discgan = tfgan.gan_model(
    generator_fn=generator,
    discriminator_fn=adversary_conv(OUTPUT_SIZE),
    real_data=tf.random_uniform(shape=[BATCH_SIZE, OUTPUT_SIZE]),
    generator_inputs=input.get_input_tensor(BATCH_SIZE, MAX_VAL)
)

# Build the GAN loss.
discgan_loss = tfgan.gan_loss(
    discgan,
    generator_loss_fn=tfgan.losses.least_squares_generator_loss,
    discriminator_loss_fn=tfgan.losses.least_squares_discriminator_loss)

# Create the train ops, which calculate gradients and apply updates to weights.
train_ops = tfgan.gan_train_ops(
    discgan,
    discgan_loss,
    generator_optimizer=GEN_OPT,
    discriminator_optimizer=OPP_OPT)
```

The core of the main training loop is as follows:

```python
for step in range(STEPS):
    train_steps_fn(sess, train_ops, global_step, train_step_kwargs={})
```

```python
        # if performed right number of steps, log
        if step % LOG_EVERY_N == 0:
            sess.run([])
            gen_l = discgan_loss.generator_loss.eval(session=sess)
            disc_l = discgan_loss.discriminator_loss.eval(session=sess)

            debug.print_step(step, gen_l, disc_l)
            losses_jerry.append(gen_l)
            losses_diego.append(disc_l)
```

3.3.7   Defining and Training for the Predictive GAN

The predictive approach does not fit the standard GAN framework, and thus was not easily implementable with `tf.contrib.gan`. It was implemented using more general low-level TensorFlow functionality. The GAN is defined as follows:

```python
# janice tensor graph
janice_input_t = tf.placeholder(shape=[BATCH_SIZE, 2], dtype=tf.float32)
janice_output_t = generator(janice_input_t)
janice_true_t = tf.strided_slice(janice_output_t, [0, -0],
                                 [BATCH_SIZE, 1], [1, 1])
priya_pred_t = tf.placeholder(shape=[BATCH_SIZE, 1], dtype=tf.float32)
# priya tensor graph
priya_input_t = tf.placeholder(shape=[BATCH_SIZE, OUTPUT_SIZE - 1],
                               dtype=tf.float32)
priya_label_t = tf.placeholder(shape=[BATCH_SIZE, 1], dtype=tf.float32)
priya_output_t = adversary_conv(OUTPUT_SIZE - 1)(priya_input_t)

# losses and optimizers
priya_loss = tf.losses.absolute_difference(priya_label_t, priya_output_t)
janice_loss = -tf.losses.absolute_difference(janice_true_t, priya_pred_t)
janice_optimizer = GEN_OPT.minimize(janice_loss)
priya_optimizer = OPP_OPT.minimize(priya_loss)
```

The main training loop is as follows:

```python
for step in range(STEPS):
    batch_inputs = input.get_input_numpy(BATCH_SIZE, MAX_VAL)
    # generate
    janice_output_n = sess.run([janice_output_t],
                               feed_dict={janice_input_t: batch_inputs})
    priya_input_n, priya_label_n = operations.slice_gen_out(janice_output_n[0])
    # update priya
    priya_output_n = None
    priya_loss_epoch = None
    for adv in range(ADV_MULT):
        _, priya_loss_epoch, priya_output_n
            = sess.run([priya_optimizer, priya_loss, priya_output_t],
```

```
                    feed_dict={priya_input_t: priya_input_n,
                               priya_label_t: priya_label_n})
# update janice
_, janice_loss_epoch = sess.run([janice_optimizer, janice_loss],
                                feed_dict={priya_pred_t: priya_output_n,
                                           janice_input_t: batch_inputs})

# log and evaluate
if step % LOG_EVERY_N == 0:
    debug.print_step(step, janice_loss_epoch, priya_loss_epoch)
    losses_janice.append(janice_loss_epoch)
    losses_priya.append(priya_loss_epoch)
```

```
# log and evaluate
```

4. Experiments

This section outlines the experimentation phase of the project, including the experimentation plan, the precise procedure followed, the experiment parameters, and the results obtained.

4.1 Experimentation Plan

The aim of the experiments is to determine quantitatively the extent to which training the implemented GANs improves the randomness properties of the generators' outputs. These are measured by producing large files of generator output values to be analyzed using the NIST statistical test suite, both before and after a generator is trained.

Independent variable: whether the generative adversarial network has been trained or not. In other words, the generator's weight parameters are the independent variable.

Dependent variable: the result produced by the NIST test suite on a file containing a large number sequence produced by the generator on a fixed input.

Controlled variables: the following variables are unchanged throughout the entire experimental procedure:

1. Contents of input dataset used to produce the sequences to be evaluated using NIST
2. Number of training epochs for the GANs
3. Number of adversary epochs per each generator epoch
4. Architecture of the generator
5. Architecture of the discriminator
6. Architecture of the predictor
7. Learning rate of the networks
8. Range of output values
9. Mini-batch size used in processing inputs

In summary, in each experiment, the randomness properties of the generator's outputs are assessed before and after training; the only changed quantity is the network's parameters that have been optimized during training.

4.2 Experimental Procedure

The precise procedure for each experiment is as follows:

1. Start the system on the HPC machine.
2. Create the pre-defined and unchanging evaluation dataset.
3. Generate output number sequences by giving evaluation dataset as input to generator.

   (a) Compute output vector for each input vector.
   (b) Round real numbers to nearest integer. If the outputs are randomly distributed over a range $[a, b]$ where $a, b \in \mathbb{R}^+$, then they will also be randomly distributed over the range $[a, b]$ where $a, b \in \mathbb{Z}^+$.
   (c) Store integers as hex strings in a text file prefixed with 0_. Hex is used to reduce the number of characters in the file.

4. Train the generative adversarial network. The generator and the adversary are trained in turn, with the adversary performing multiple parameter updates for each update made by the generator.
5. Generate output number sequences by giving evaluation dataset as input to generator.

   (a) Compute output vector for each input vector.
   (b) Round real numbers to nearest integer. If the outputs are randomly distributed over a range $[a, b]$ where $a, b \in \mathbb{R}^+$, then they will also be randomly distributed over the range $[a, b]$ where $a, b \in \mathbb{Z}^+$.
   (c) Store integers as hex strings in a text file prefixed with 1_. Hex is used to reduce the number of characters in the file.

6. Retrieve files containing generated number sequences using `scp`.
7. Convert the text file's contents from hex characters to binary characters using the `decode_nist.py` utility.
8. Run the NIST test suite on the converted file, performing all tests with default/recommended settings.
9. Store the testing result files produced by the NIST test suite.

For both approaches 10 experiments are carried out. In each experiment the GAN is trained for 200000 epochs, with the adversary being trained 3 times at each epoch (see subsection 2.3.6 for the rationale behind this choice).

Training is performed with mini-batches of 2046 data points, while the learning rate of the networks is set to 0.02. Both values were manually selected during prototyping as they resulted in good performance. The generator outputs real numbers constrained to the range $[0, 2^{16} - 1]$, which are rounded to the nearest integer and converted to unsigned 16-bit integers for evaluation. The evaluation dataset consists of 400 batches of 2046 inputs, for a total of 818400 input samples. The generator produces 8 real numbers for

each input, and each real number yields 16 bits for the output sequence. In total, each evaluation output consists of 104755200 bits.

The NIST test suite is applied with the recommended setting of using 1000000 bits for each test. The test suite includes 188 separate tests, referred to from here onward as a *test*, each repeated 10 times. Each repetition will be referred to as a *test instance*. For each test, NIST reports the number of individual instances that passed, as well as a p-value for the distribution of p-values of the 10 test instances. A test can fail in one of two ways: either the number of passed instances is insufficient, or the p-value for the p-value distribution is below a critical value. See appendix A for a sample of the test suite's output.

4.3 Results

The statistical test results from the 20 training instances are collated here. Tables 4.1 and 4.2 show the performance of the generator neural networks before and after training. Table 4.3 shows the average performance after training for the two approaches, while table 4.4 shows, for each experiment, the change in performance resulting from training. Lastly, table 4.5 shows the average improvement for the discriminative and predictive approaches, across all 10 experiments.

The symbols in the tables have the following meanings: i identifies the experiment, with D_i and P_i referring to discriminative and predictive experiments, respectively. T refers to the overall number of separate tests carried out by the NIST test suite, while T_I refers to the number of total test instances. F_I and $F_{I\%}$ refers to the number of failed test instances and the percentage of failed test instances, respectively. F_p represents the number of tests failed due to an abnormal distribution of the p-values of the underlying test instances. F_T and $F_\%$ refer to the absolute number and percentage of separate tests failed, respectively. In tables 4.3, 4.4 and 4.5, the convention followed is that Δ refers to the change in some quantity, and the angle brackets $\langle \rangle$ refer to the average value of the enclosed quantity.

Figures 4.1 and 4.2 showcase the recorded outputs of the loss functions of both generative adversarial networks during the 9th discriminative experiment and the 9th predictive experiment, respectively.

i	T	T_I	F_I	$F_{I\%}/\%$	F_p	F_T	$F_\%/\%$
D_1	188	1802	1800	99.9	188	188	100
D_2	188	1802	1791	99.4	188	188	100
D_3	188	1802	1798	99.8	188	188	100
D_4	188	1802	1802	100.0	188	188	100
D_5	188	1802	1788	99.2	188	188	100
D_6	188	1802	1792	99.4	188	188	100
D_7	188	1802	1799	99.8	188	188	100
D_8	188	1802	1801	99.9	188	188	100
D_9	188	1802	1802	100.0	188	188	100
D_{10}	188	1802	1791	99.4	188	188	100
P_1	188	1802	1789	99.3	188	188	100
P_2	188	1802	1802	100.0	188	188	100
P_3	188	1802	1799	99.8	188	188	100
P_4	188	1802	1802	100.0	188	188	100
P_5	188	1802	1792	99.4	188	188	100
P_6	188	1802	1798	99.8	188	188	100
P_7	188	1802	1792	99.4	188	188	100
P_8	188	1802	1801	99.9	188	188	100
P_9	188	1802	1802	100.0	188	188	100
P_{10}	188	1802	1801	99.9	188	188	100

Table 4.1: Test results for untrained generators.

i	T	T_I	F_I	$F_{I\%}/\%$	F_p	F_T	$F_\%/\%$
D_1	188	1802	29	1.6	3	3	2
D_2	188	1724	109	6.3	5	11	6
D_3	188	1724	123	7.1	4	11	6
D_4	188	1828	168	9.2	9	18	10
D_5	188	1854	49	2.7	3	5	3
D_6	188	1776	30	1.7	5	5	3
D_7	188	1854	21	1.1	4	4	2
D_8	188	1776	46	2.6	5	5	3
D_9	188	1802	25	1.4	3	3	2
D_{10}	188	1854	14	0.8	2	4	2
P_1	188	1802	39	2.2	2	4	2
P_2	188	1828	27	1.5	3	3	2
P_3	188	1828	35	1.9	3	4	2
P_4	188	1828	100	5.5	2	8	4
P_5	188	1802	52	2.9	2	3	2
P_6	188	1880	111	5.9	4	9	5
P_7	188	1802	25	1.4	3	3	2
P_8	188	1880	96	5.1	2	4	2
P_9	188	1854	36	1.9	2	3	2
P_{10}	188	1802	37	2.1	4	4	2

Table 4.2: Test results for trained generators.

i	T	$\langle T_I \rangle$	$\langle F_I \rangle$	$\langle F_{I\%} \rangle/\%$	$\langle F_p \rangle$	$\langle F_T \rangle$	$\langle F_\% \rangle/\%$
D	188	1800	61	3.5	4.3	6.9	3.9
P	188	1830	56	3.0	2.7	4.5	2.5

Table 4.3: Average test results for trained generators.

i	$\Delta F_{I\%}/\%$	ΔF_p	ΔF_T	$\Delta F_\%/\%$
D_1	-98.3	-185	-185	-98
D_2	-93.1	-183	-177	-94
D_3	-92.7	-184	-177	-94
D_4	-90.8	-179	-160	-90
D_5	-96.5	-185	-183	-97
D_6	-97.7	-183	-183	-97
D_7	-98.7	-184	-184	-98
D_8	-97.3	-183	-183	-97
D_9	-98.6	-185	-185	-98
D_{10}	-98.6	-186	-184	-98
P_1	-97.1	-186	-184	-98
P_2	-98.5	-185	-185	-98
P_3	-97.9	-185	-184	-98
P_4	-94.5	-186	-180	-96
P_5	-96.5	-186	-186	-98
P_6	-93.9	-184	-179	-95
P_7	-98.0	-185	-185	-98
P_8	-94.8	-186	-184	-98
P_9	-98.1	-186	-185	-98
P_{10}	-97.8	-184	-184	-98

Table 4.4: Improvements in the measured quantities as a result of training the networks.

i	$\langle \Delta F_{I\%} \rangle/\%$	$\langle \Delta F_p \rangle$	$\langle \Delta F_T \rangle$	$\langle \Delta F_\% \rangle/\%$
D	-96.2	-183.7	-180.1	-96.1
P	-96.7	-185.3	-183.6	-97.5

Table 4.5: Average performance change for the discriminative and predictive approaches across all tests.
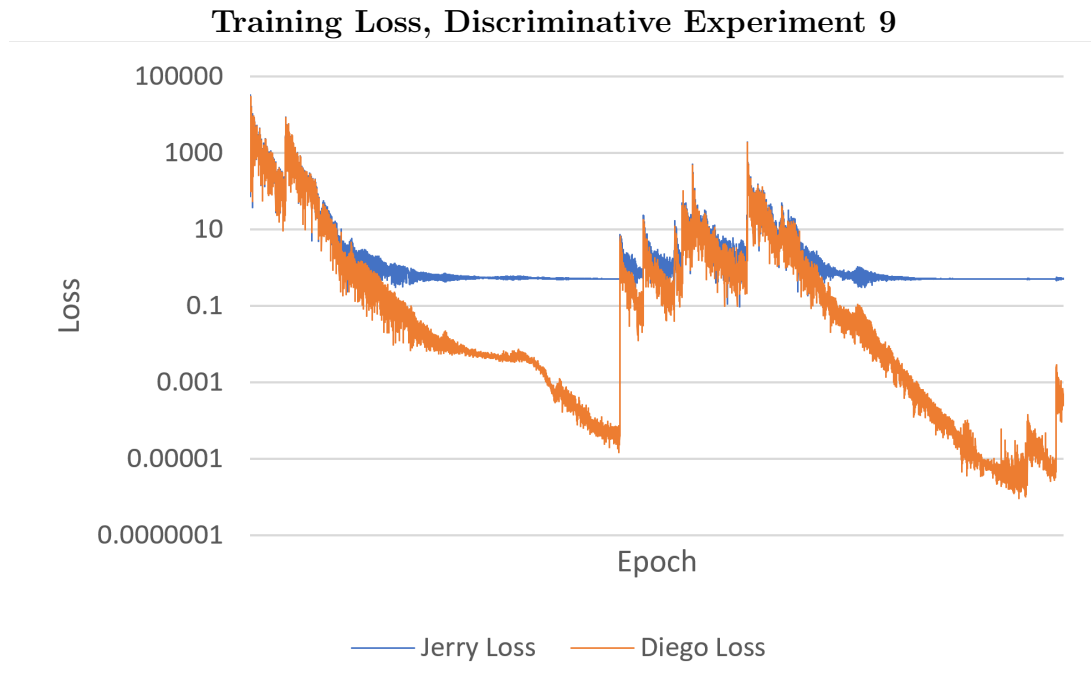
Figure 4.1: A plot of the training loss during training of the discriminative model. The discriminator (orange) has a tendency to gradually keep improving its performance (decreasing loss) while the generator (blue) plateaus. Occasionally, however, the generator "discovers" something which destabilizes the learning process. This pattern repeats multiple times.
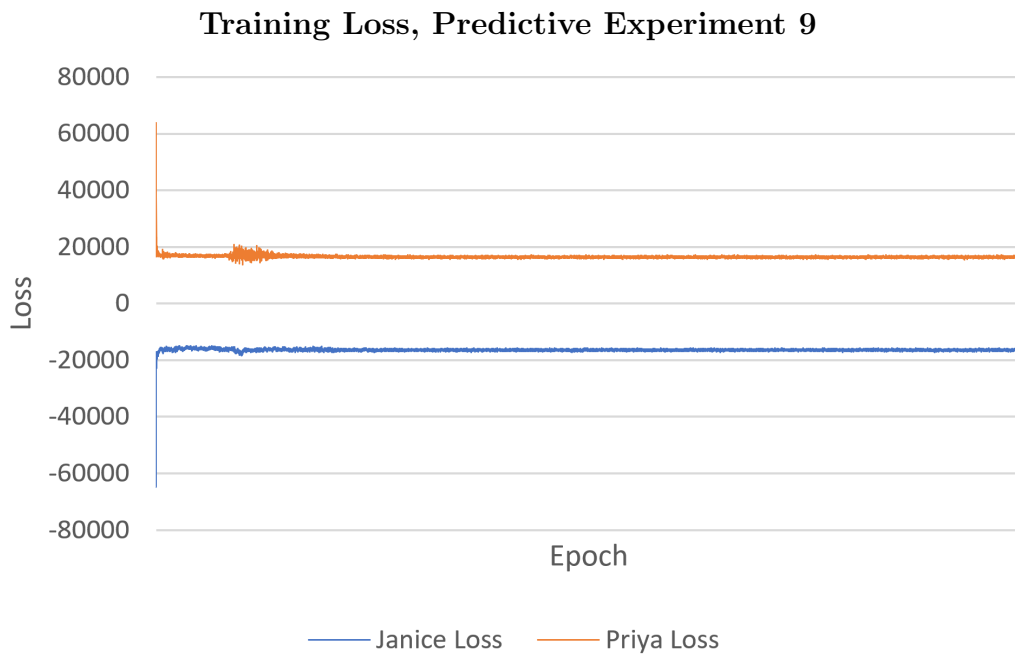
Figure 4.2: A plot of the training loss during training of the predictive model. The predictor and generator both rapidly improve their performance in the initial phases of training. For the entire remaining duration of the training, the loss values remain fairly constant.

4.4 Evaluation

The results show that, prior to training, the generator networks are not suitable for the production of pseudo-random numbers. As seen in table 4.1, the generators' outputs fail to pass any statistical tests. Indeed, visualizations of the generator's outputs before training (figures 4.3 and 4.5) shows that the outputs of the networks follow extremely predictable patterns. It is worthy of note that a very small percentage of the total test instances $F_{I\%}$ does pass. This behavior is not surprising, due to the probabilistic nature of statistical randomness testing: unlikely results do occasionally occur, and even a non-random sequence may occasionally appear to be at least somewhat random. Passing a small proportion of test instances does not suffice as a condition for passing any of the overall tests, where the failure rate is 100% across all experiments.

Table 4.2 shows the performance of the generators after training, which is consistently very strong across both approaches. The evaluated number sequences achieved below 10% failure rate in all but one of the experiments, with the majority of experiments resulting in a failure rate below 5%. As seen in figures 4.4 and 4.6, the previously observed patterns in the generator's output are no longer visible. Table 4.4 more clearly quantifies the change in performance achieved in each experiment. In most experiments the absolute change in failure percentage points was greater than 95%,

From table 4.2 we observe that there is higher variance in the performance observed in the discriminative experiments; however, a larger number of experiments would be required to verify whether this is a trend. An overall comparison of the two approaches may be carried out by looking at tables 4.3 and 4.5. In both tables we can see that the predictive approach shows somewhat better performance, with the predictively trained generators producing approximately 2/3 of the number of failures produced by the discriminatively trained generator. The difference is more noticeable in the overall test failures and in the number of tests failed due to anomalous p-value distributions. This suggests that the discriminative approach's weaker overall performance is explained by failure in a small number of specific tests, rather than by a generally weaker performance in test instances across all tests.

The training loss of the network may also be examined from figures 4.1 and 4.2. Neither plot looks like a typical neural network loss function (figure 2.5). In the discriminative case, we observe that the discriminator keeps steadily improving its performance relative to the generator, until something destabilizes the learning process and causes both loss functions to spike. The generator, in turn, plateaus as the discriminator improves.

In the second plot, we observe that both the generator and the predictor rapidly improve initially, until both plateau. It seems implausible that the generator's very strong pseudo-randomness properties at the end of training could be explained by the first few hundred training epochs. Rather, the apparent plateau in both loss functions might be explained by an extremely balanced learning process, in which both the generator and the predictor "learn at the same pace"; both are improving, but their performance relative to each other is largely unchanged.

Overall, the data shows an extremely strong improvement in the generator's performance as a result of training, and clearly demonstrates the validity of the generative adversarial network approach. The peculiar loss plots raise questions about the behavior of the

networks and suggest that the learning task at hand is a difficult one, with gradient descent encountering frequent local minima or flat regions.

Output Sample, Before Discriminative Training
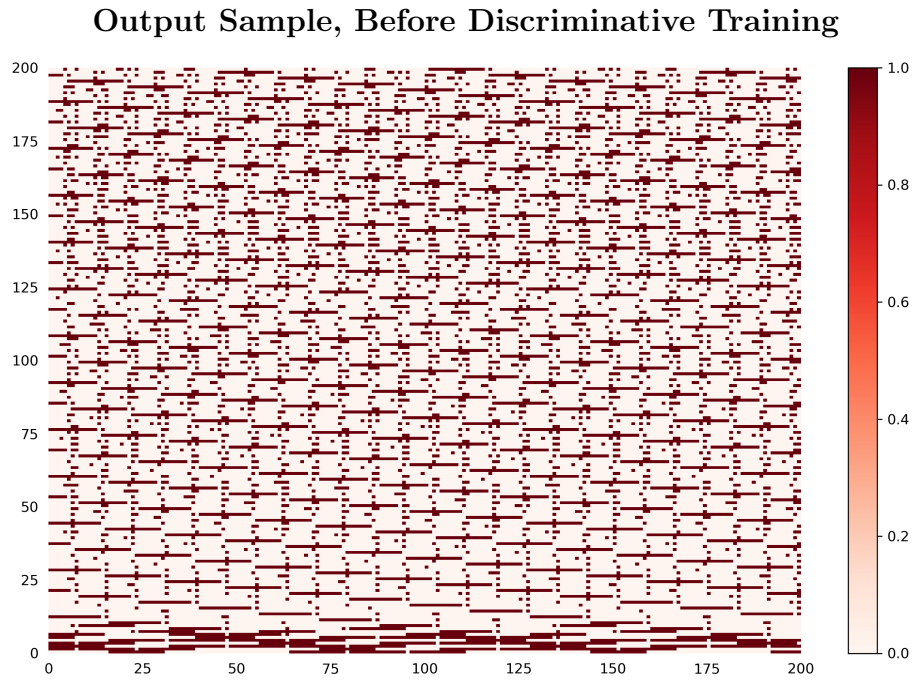
Figure 4.3: Visualization of the generator output as produced in the 9th discriminative training instance, before training. The first 40000 bits are presented as a 200 by 200 grid. The non-randomness of the output is obvious due to the visible patterns.

Output Sample, After Discriminative Training
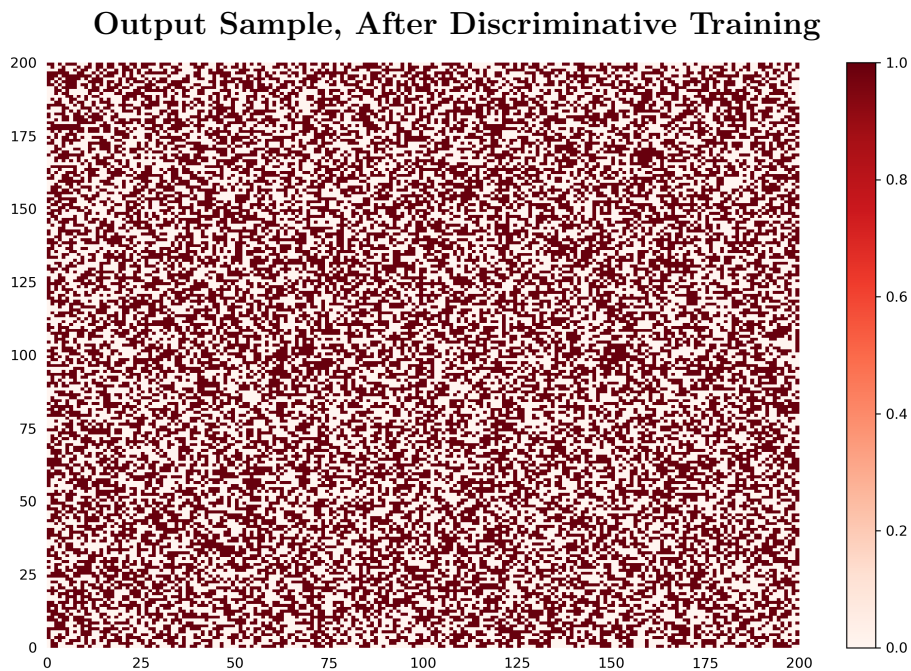
Figure 4.4: Visualization of the generator output as produced in the 9th discriminative training instance, after 200000 training iterations. The first 40000 bits are presented as a 200 by 200 grid. Clearly the generator neural network has learned to produce more randomness, as no patterns can reasonably be discerned by a human observer.

Output Sample, Before Predictive Training
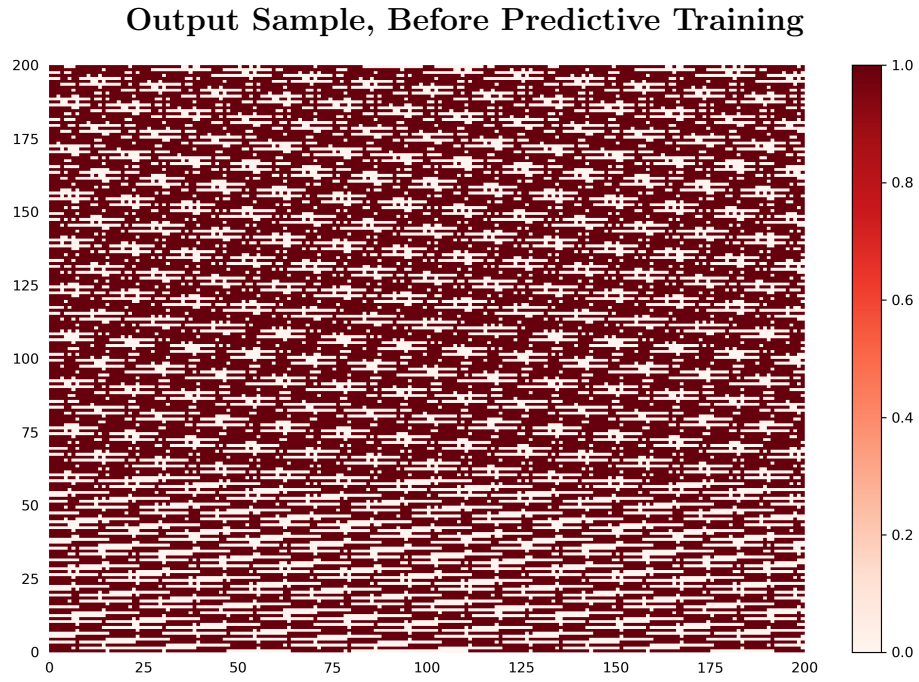
Figure 4.5: Visualization of the generator output as produced in the 9th predictive training instance, before training. The first 40000 bits are presented as a 200 by 200 grid. The non-randomness of the output is obvious due to the visible patterns.

Output Sample, After Predictive Training
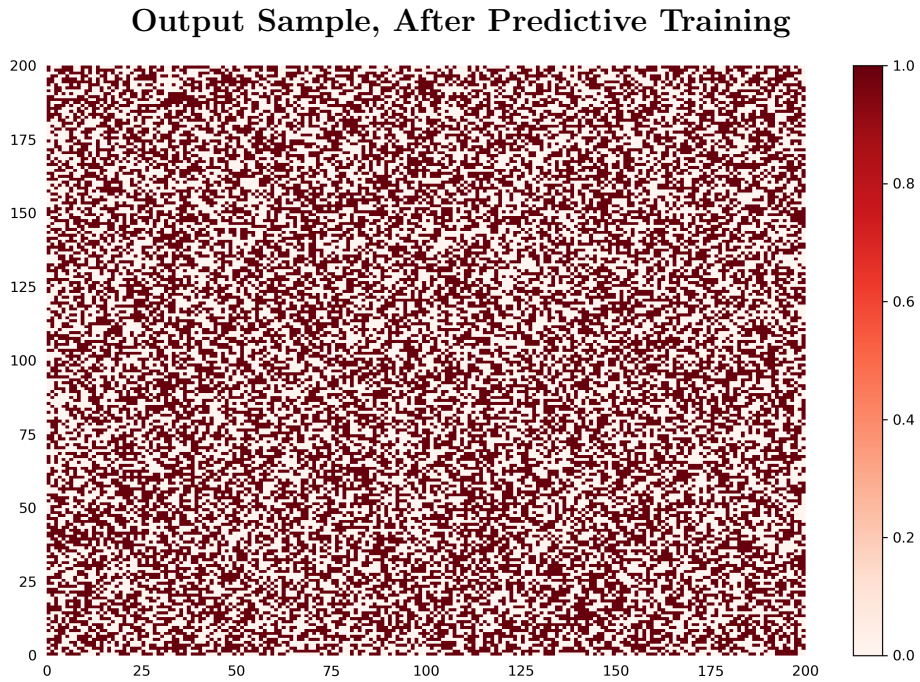
Figure 4.6: Visualization of the generator output as produced in the 9th predictive training instance, after 200000 training iterations. The first 40000 bits are presented as a 200 by 200 grid. As with the discriminative training, no patterns can reasonably be discerned by a human observer; clearly the network has learned some notion of randomness.

5.   Conclusion

Pseudo-random number generators, deterministic algorithms producing sequences of numbers which appear randomly sampled [15, p. 7], are used throughout the field of cryptography and are a fundamental element of many cryptographic systems such as encryption algorithms [38, p. 169] [31, p. 1]. As PRNGs are often a single point of failure for such systems, their design and analysis is an important field of investigation [31, p. 2] [21].

Machine learning, and in particular optimizable parametric representations of non-linear functions called deep neural networks, has been tremendously successful in recent years [40, p. 24-29]. Despite this, the literature review carried out prior to this investigation suggests that little effort has gone into the application of machine learning methods to the implementation of PRNGs. Some attempts have been made with little success, and the papers are relatively obscure [22] [23] [43].

The aim of this investigation was to determine whether a deep neural network can be trained to behave as a PRNG, and whether such a PRNG could be used in a cryptographic context. The scope of the investigation was limited to the statistical properties of a PRNG's output, and did not include cryptanalysis of the implementation. This work relies on the assumption that a neural network can represent a good pseudo-random function, and that discovering such a function by means of gradient descent is tractable.

To accomplish the project aims, this project explored a novel approach, presenting two generative adversarial network (GAN) models designed to train a neural network to generate pseudo-random numbers. This avenue of investigation was motivated by the observation that the adversarial nature of GANs closely matches the roles of a PRNG and an adversary in a cryptographic context: the PRNG seeks to produce numbers that are unpredictable to the adversary, who attempts to predict future outputs.

The design is based on three innovations. Firstly, the standard GAN approach is modified: one of the two approaches, termed the predictive approach, eliminates the need for a reference distribution, relying solely on the interaction between the generator network and a convolutional predictor network trying to predict the generator's future outputs. This is contrasted with the classic discriminative approach, where a discriminator classifies input sequences as either belonging to the reference distribution, or as produced by the generator. The predictive approach is conceptually simpler and more efficient.

Furthermore, the deep neural network representing the generator uses a custom activation function at its output layer which computes a modulo operation on the network's outputs. This has a dramatic impact on the performance of the generator, presumably "shuffling" many of the patterns that may otherwise be discernible in the output.

Lastly, previous research in the field has used recurrent neural networks to model the statefulness of PRNGs, leading to complex systems that may be difficult to train. This work sidesteps these difficulties by implementing the generator as a simple stateless fully connected feed-forward network, where the state of the PRNG is modeled by additional non-random inputs to the model.

These design innovations resulted in a system that is elegant, conceptually simple, and robust. It is elegant in the sense that there is a close analogy between the structure of generative adversarial networks and the real-world security scenario involving a PRNG and a malicious agent. The system's simplicity and robustness are a result of the novel simplifications to the GAN architecture as well as the simple architecture of their constituent neural networks (in particular the generator).

This report has demonstrated that a generative adversarial network with the illustrated modifications can train the generator in such a manner as to produce pseudo-random number sequences with good statistical properties. In every experiment, the generator's performance on the NIST test suite improved from passing no tests to passing the majority of tests. At best, the generator could pass around 99% of test instances, and around 98% of unique tests. This metric alone does not justify use of the model in a cryptographic context, where a thorough cryptanalysis of the generator would be required, in addition to failing none of NIST's statistical tests.

As far as the author is aware, this novel research has outperformed the majority of published attempts in the generation of pseudo-random numbers using neural networks. Crucially, it has done so with a system that is much simpler and more robust than previous attempts, which have often been complicated even by their own authors' admission. Thanks to its simplicity, the approach considered in this work is easy to understand and modify. A number of obvious improvements to the system could be made, possibly producing even better results and warranting further investigation.

6. Further Investigation

The strong PRNG performance of the proposed generative adversarial networks, combined with the simplicity and robustness of the approach, suggest that the topic merits further investigation. Additional tests beyond the scope of this investigation were carried out to determine whether increasing the generator's size or input dimensionality would improve performance. However, the analysis of these experiments could not be included in this report. A number of suggestions for further research include:

Architectural Cross-Validation. The architecture of the neural networks was chosen by intuition as well as by trial-and-error. Systematic cross-validation of different architectures could improve performance. Especially the network structure, the input and output dimensions, the loss functions, and the gradient descent variant almost certainly have room for improvement [29, Neural Networks Part 3: Learning and Ealuation].

Hyperparameter Optimization. The models' hyperparameters were also chosen by trial-and-error. Systematic hyperparameter search to fine-tune the model could improve performance. In particular, important hyperparameters such as learning rate and mini-batch size should be optimized [29, Neural Networks Part 3: Learning and Ealuation].

Recurrent Neural Networks. The use of recurrent networks may add further capabilities to the networks [27, chap. 10].

Cryptanalysis. Formal cryptanalysis of the implementation should be performed to determine whether a GAN-based PRNG can truly be cryptographically secure [39, Abstract].

Potential Applications. Potential applications of the system should also be considered. This could include existing public-key encryption algorithms that make use of random numbers, such as RSA. It could also include entirely new algorithms. For example, a hypothetical symmetric-key encryption/decryption algorithm could entail two communicating entities using synchronized neural network CSPRNGs. The two networks could independently and secretly generate the same encryption/decryption key for each message being transmitted. Initial synchronization could be performed by transmitting a truly random seed and a truly random sequence offset, using a key exchange protocol like Diffie-Hellman key exchange [14, p. 174-177].

7. Bibliography

[1] Docker. `https://www.docker.com/`. Accessed: 23/04/2018.

[2] git. `https://git-scm.com/`. Accessed: 23/04/2018.

[3] Google's tensorflow gaining popularity. why? `https://analyticsindiamag.com/googles-tensorflow-gaining-popularity/`. Accessed: 04/11/2018.

[4] Jetbrains: Pycharm. `https://www.jetbrains.com/pycharm/`. Accessed: 23/04/2018.

[5] Numpy - the fundamental package for scientific computing with python. `http://www.numpy.org`. Accessed: 25/02/2018.

[6] A short history of machine learning – every manager should read. `https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#29b44c5c15e7`. Accessed: 17/03/2018.

[7] Tensorflow: an open-source software library for machine intelligence. `www.tensorflow.org`. Accessed: 12/08/2017.

[8] Tensorflow: Graphs and sessions. `https://www.tensorflow.org/programmers_guide/graphs`. Accessed: 11/04/2018.

[9] Tensorflow: Introduction. `https://www.tensorflow.org/programmers_guide/low_level_intro`. Accessed: 11/04/2018.

[10] Tensorflow: Tutorials. `https://www.tensorflow.org/tutorials/`. Accessed: 21/04/2018.

[11] Martín Abadi and David G Andersen. Learning to protect communications with adversarial neural cryptography. *arXiv preprint arXiv:1610.06918*, 2016.

[12] Saleh Albelwi and Ausif Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, 19(6):242, 2017.

[13] Inc. Amazon.com. Machine learning on aws. `https://aws.amazon.com/machine-learning/`. Accessed: 17/03/2018.

[14] Ross J Anderson. *Security engineering: a guide to building dependable distributed systems.* John Wiley & Sons, 2010.

[15] Elaine B Barker and John Michael Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[16] Deborah J Bennett. *Randomness*. Harvard University Press, 2009.

[17] Peter Bierhorst, Emanuel Knill, Scott Glancy, Yanbao Zhang, Alan Mink, Stephen Jordan, Andrea Rommal, Yi-Kai Liu, Bradley Christensen, Sae Woo Nam, et al. Experimentally generated randomness certified by the impossibility of superluminal signals. *arXiv preprint arXiv:1803.06219*, 2018.

[18] MedCalc Software bvba. Medcalc manual: Mod function. `https://www.medcalc.org/manual/mod_function.php`. Accessed: 23/04/2018.

[19] Microsoft Corporation. Azure machine learning - open and elastic ai development spanning the cloud and the edge. `https://azure.microsoft.com/en-gb/overview/machine-learning/`. Accessed: 17/03/2018.

[20] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[21] Lih-Yuan Deng and Dale Bowman. Developments in pseudo-random number generators. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(5), 2017.

[22] VV Desai, VB Deshmukh, and DH Rao. Pseudo random number generator using elman neural network. In *Recent Advances in Intelligent Computational Systems (RAICS), 2011 IEEE*, pages 251–254. IEEE, 2011.

[23] VV Desai, Ravindra T Patil, VB Deshmukh, and DH Rao. Pseudo random number generator using time delay neural network. *World*, 2(10):165–169, 2012.

[24] Knuth Donald et al. The art of computer programming, volume 2: Semi numerical algorithms, 1998.

[25] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Cryptography engineering. *Design Princi*, 2010.

[26] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[28] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[29] Andrej Karpathy. Lecture notes for cs231n convolutional neural networks for visual recognition, 2017.

[30] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[31] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, pages 168–188. Springer, 1998.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] Alexander Klimov, Anton Mityagin, and Adi Shamir. Analysis of neural cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 288–298. Springer, 2002.

[34] A Lavasani and T Eghlidos. Practical next bit test for evaluating pseudorandom sequences. *Scientia Iranica. Transaction D, Computer Science & Engineering, Electrical*, 16(1):19, 2009.

[35] Google LLC. Cloud automl alpha - train high quality custom machine learning models with minimum effort and machine learning expertise. `https://cloud.google.com/automl/`. Accessed: 17/03/2018.

[36] JGraph Ltd. draw.io. `https://www.draw.io/`. Accessed: 20/03/2018.

[37] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[38] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[39] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-Allen and Hamilton Inc Mclean Va, 2001.

[40] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[41] Sagar Sharma. Activation functions: Neural networks. `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`. Accessed: 19/03/2018.

[42] Joel Shor. Tfgan tutorial. `https://github.com/tensorflow/models/blob/master/research/gan/tutorial.ipynb`. Accessed: 21/04/2018.

[43] Kayvan Tirdad and Alireza Sadeghian. Hopfield neural networks as pseudo random number generators. In *Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American*, pages 1–6. IEEE, 2010.

[44] Shiping Wen, Zhigang Zeng, Tingwen Huang, and Yide Zhang. Exponential adaptive lag synchronization of memristive neural networks via fuzzy method and applications in pseudorandom number generators. *IEEE Transactions on Fuzzy Systems*, 22(6):1704–1713, 2014.

A. Statistical Test Result Sample

The outputs of the NIST test suite are extensive and cannot be fully reproduced in this appendix. One output report, produced for the 9th training instance of the predictive approach, is provided as an example.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <1_janice_dieharder.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10   P-VALUE   PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  1   0   0   0   0   1   2   2   1   3  0.350485     10/10      Frequency
  2   2   0   1   0   0   0   2   2   1  0.534146      9/10      BlockFrequency
  1   0   0   0   0   2   0   1   2   4  0.066882     10/10      CumulativeSums
  1   0   0   0   0   2   0   1   0   6  0.000199     10/10      CumulativeSums
  0   0   0   0   3   0   2   2   2   1  0.213309     10/10      Runs
  1   0   1   3   0   1   2   0   1   1  0.534146     10/10      LongestRun
  7   0   1   1   1   0   0   0   0   0  0.000003 *    4/10   *  Rank
 10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  FFT
  0   0   0   2   0   1   2   2   1   2  0.534146     10/10      NonOverlappingTemplate
  1   1   0   2   1   0   0   1   2   2  0.739918     10/10      NonOverlappingTemplate
  1   2   2   0   1   1   0   1   1   1  0.911413     10/10      NonOverlappingTemplate
  0   1   1   2   0   1   2   2   1   0  0.739918     10/10      NonOverlappingTemplate
  0   1   0   1   2   0   2   1   0   3  0.350485     10/10      NonOverlappingTemplate
  1   2   0   0   2   1   0   1   2   1  0.739918     10/10      NonOverlappingTemplate
  0   1   1   0   0   0   2   3   1   2  0.350485     10/10      NonOverlappingTemplate
  2   1   3   0   0   0   1   0   1   2  0.350485     10/10      NonOverlappingTemplate
  0   0   0   2   0   3   0   3   0   2  0.066882     10/10      NonOverlappingTemplate
  2   0   1   1   2   0   1   1   1   1  0.911413     10/10      NonOverlappingTemplate
  0   0   2   1   1   1   1   1   2   1  0.911413     10/10      NonOverlappingTemplate
  2   2   2   0   1   1   0   1   1   0  0.739918      9/10      NonOverlappingTemplate
  1   0   0   1   3   1   1   1   0   2  0.534146     10/10      NonOverlappingTemplate
  0   1   0   0   1   0   0   2   4   2  0.066882     10/10      NonOverlappingTemplate
  1   0   2   2   0   0   2   0   1   2  0.534146     10/10      NonOverlappingTemplate
  0   2   1   1   1   0   1   2   1   1  0.911413     10/10      NonOverlappingTemplate
  0   0   1   2   0   3   1   1   1   1  0.534146     10/10      NonOverlappingTemplate
  3   0   2   2   0   0   1   2   0   0  0.213309     10/10      NonOverlappingTemplate
  1   2   1   1   1   1   1   1   1   0  0.991468     10/10      NonOverlappingTemplate
  0   2   1   1   0   2   2   1   0   1  0.739918     10/10      NonOverlappingTemplate
  0   1   3   1   1   0   0   2   2   0  0.350485     10/10      NonOverlappingTemplate
  1   1   0   1   1   2   1   1   0   2  0.911413     10/10      NonOverlappingTemplate
  2   2   0   1   2   0   2   1   0   0  0.534146      9/10      NonOverlappingTemplate
  1   2   2   0   3   0   1   1   0   0  0.350485     10/10      NonOverlappingTemplate
  1   0   0   0   0   0   1   1   3   4  0.035174     10/10      NonOverlappingTemplate
  1   2   0   1   0   1   1   0   1   3  0.534146     10/10      NonOverlappingTemplate
  0   2   3   2   0   0   0   0   1   2  0.213309     10/10      NonOverlappingTemplate
```

3	2	1	0	2	1	0	0	1	0	0.350485	10/10	NonOverlappingTemplate
1	0	1	2	1	1	0	2	2	0	0.739918	10/10	NonOverlappingTemplate
0	0	2	2	0	2	0	1	2	1	0.534146	10/10	NonOverlappingTemplate
0	0	1	0	1	2	3	2	1	0	0.350485	10/10	NonOverlappingTemplate
1	0	1	1	1	0	1	2	1	2	0.911413	10/10	NonOverlappingTemplate
0	5	0	1	0	0	1	1	0	2	0.008879	10/10	NonOverlappingTemplate
1	1	0	1	3	0	0	0	0	4	0.035174	10/10	NonOverlappingTemplate
2	2	1	0	0	2	0	1	1	1	0.739918	10/10	NonOverlappingTemplate
1	2	1	0	1	2	1	0	2	0	0.739918	10/10	NonOverlappingTemplate
1	4	1	0	0	1	3	0	0	0	0.035174	9/10	NonOverlappingTemplate
0	0	0	0	3	1	1	2	2	1	0.350485	10/10	NonOverlappingTemplate
0	2	1	0	4	2	1	0	0	0	0.066882	10/10	NonOverlappingTemplate
1	3	1	1	0	1	0	2	0	1	0.534146	10/10	NonOverlappingTemplate
3	0	0	3	1	0	2	1	0	0	0.122325	10/10	NonOverlappingTemplate
3	3	1	0	1	0	0	1	0	1	0.213309	10/10	NonOverlappingTemplate
2	0	1	2	3	0	0	0	0	2	0.213309	9/10	NonOverlappingTemplate
2	1	0	0	2	1	1	1	1	1	0.911413	10/10	NonOverlappingTemplate
1	0	0	0	3	2	1	2	1	0	0.350485	10/10	NonOverlappingTemplate
0	0	0	1	3	1	0	3	0	2	0.122325	10/10	NonOverlappingTemplate
0	1	2	2	0	2	1	0	1	1	0.739918	10/10	NonOverlappingTemplate
0	0	1	3	2	2	0	1	0	1	0.350485	10/10	NonOverlappingTemplate
2	2	2	1	0	1	1	0	0	1	0.739918	9/10	NonOverlappingTemplate
0	1	1	0	0	2	0	2	2	2	0.534146	10/10	NonOverlappingTemplate
1	1	1	0	1	1	2	1	1	1	0.991468	9/10	NonOverlappingTemplate
1	0	1	0	3	1	0	2	1	1	0.534146	10/10	NonOverlappingTemplate
2	1	2	0	2	0	2	0	0	1	0.534146	9/10	NonOverlappingTemplate
0	1	0	1	1	0	2	0	3	2	0.350485	10/10	NonOverlappingTemplate
0	0	2	2	3	1	0	0	1	1	0.350485	10/10	NonOverlappingTemplate
0	0	2	1	0	2	1	2	1	1	0.739918	10/10	NonOverlappingTemplate
2	1	3	0	0	1	0	1	2	0	0.350485	10/10	NonOverlappingTemplate
2	1	3	1	1	2	0	0	0	0	0.350485	10/10	NonOverlappingTemplate
1	1	2	1	0	0	0	2	3	0	0.350485	10/10	NonOverlappingTemplate
0	1	0	2	2	1	1	2	1	0	0.739918	10/10	NonOverlappingTemplate
0	1	0	1	0	1	2	1	4	0	0.122325	10/10	NonOverlappingTemplate
1	1	2	3	0	1	1	0	1	0	0.534146	10/10	NonOverlappingTemplate
0	1	1	0	1	1	0	3	0	3	0.213309	10/10	NonOverlappingTemplate
0	0	1	1	2	0	0	3	1	2	0.350485	10/10	NonOverlappingTemplate
0	1	0	2	2	2	1	0	1	1	0.739918	10/10	NonOverlappingTemplate
1	1	0	0	0	1	3	2	1	1	0.534146	10/10	NonOverlappingTemplate
0	1	1	0	1	2	3	0	1	1	0.534146	10/10	NonOverlappingTemplate
0	1	1	1	1	0	1	3	1	1	0.739918	10/10	NonOverlappingTemplate
0	1	2	4	1	0	0	1	1	0	0.122325	10/10	NonOverlappingTemplate
1	0	1	1	1	2	1	2	1	0	0.911413	10/10	NonOverlappingTemplate
0	1	1	0	2	1	0	2	2	1	0.739918	10/10	NonOverlappingTemplate
1	0	1	4	1	1	1	0	1	0	0.213309	10/10	NonOverlappingTemplate
3	1	2	2	0	0	0	0	1	1	0.350485	10/10	NonOverlappingTemplate
0	1	1	1	1	1	0	2	0	3	0.534146	10/10	NonOverlappingTemplate
0	0	0	2	0	1	2	2	1	2	0.534146	10/10	NonOverlappingTemplate
0	3	1	1	1	1	1	1	0	1	0.739918	10/10	NonOverlappingTemplate
0	1	1	0	1	2	2	0	2	1	0.739918	10/10	NonOverlappingTemplate
1	0	4	0	0	0	0	2	1	2	0.066882	9/10	NonOverlappingTemplate
2	0	1	0	0	3	2	1	0	1	0.350485	9/10	NonOverlappingTemplate
1	2	1	2	0	3	0	1	0	0	0.350485	10/10	NonOverlappingTemplate
1	1	0	1	1	3	2	0	1	0	0.534146	10/10	NonOverlappingTemplate
1	1	0	0	1	2	2	1	2	0	0.739918	10/10	NonOverlappingTemplate
0	1	1	1	0	0	3	0	2	2	0.350485	10/10	NonOverlappingTemplate
1	0	1	1	2	2	0	2	0	1	0.739918	10/10	NonOverlappingTemplate
1	1	1	1	1	0	1	2	1	1	0.991468	10/10	NonOverlappingTemplate
0	0	0	0	0	2	1	3	1	3	0.122325	10/10	NonOverlappingTemplate
0	1	0	1	2	1	2	1	2	0	0.739918	10/10	NonOverlappingTemplate

0	0	0	2	1	0	2	1	1	3	0.350485	10/10	NonOverlappingTemplate
0	0	0	2	1	4	1	0	0	2	0.066882	10/10	NonOverlappingTemplate
4	1	0	1	1	0	0	0	2	1	0.122325	9/10	NonOverlappingTemplate
0	0	3	0	0	3	0	0	3	1	0.035174	10/10	NonOverlappingTemplate
1	0	0	1	2	1	0	3	1	1	0.534146	10/10	NonOverlappingTemplate
1	0	0	2	0	3	0	1	2	1	0.350485	10/10	NonOverlappingTemplate
1	0	3	0	0	1	2	2	0	1	0.350485	10/10	NonOverlappingTemplate
1	1	0	2	0	3	1	2	0	0	0.350485	10/10	NonOverlappingTemplate
2	2	0	1	0	0	0	2	1	2	0.534146	10/10	NonOverlappingTemplate
0	0	1	2	2	3	1	0	1	0	0.350485	10/10	NonOverlappingTemplate
1	0	1	1	1	1	1	2	2	0	0.911413	10/10	NonOverlappingTemplate
1	1	1	0	2	0	1	1	0	3	0.534146	10/10	NonOverlappingTemplate
0	1	2	2	0	0	1	1	2	1	0.739918	10/10	NonOverlappingTemplate
0	0	0	2	0	1	1	4	0	2	0.066882	10/10	NonOverlappingTemplate
0	1	1	2	2	1	0	1	1	1	0.911413	10/10	NonOverlappingTemplate
1	0	1	0	1	1	0	2	1	3	0.534146	10/10	NonOverlappingTemplate
3	1	1	1	0	2	0	1	0	1	0.534146	8/10	NonOverlappingTemplate
1	1	0	0	1	2	4	0	1	0	0.122325	10/10	NonOverlappingTemplate
1	0	0	2	0	1	1	1	3	1	0.534146	10/10	NonOverlappingTemplate
0	1	1	3	1	0	1	1	2	0	0.534146	10/10	NonOverlappingTemplate
2	1	0	3	0	2	0	0	1	1	0.350485	10/10	NonOverlappingTemplate
1	0	1	1	1	0	2	1	2	1	0.911413	10/10	NonOverlappingTemplate
1	0	0	1	0	0	2	0	3	3	0.122325	10/10	NonOverlappingTemplate
0	0	0	1	0	2	1	0	3	3	0.122325	10/10	NonOverlappingTemplate
1	1	0	0	2	1	1	0	2	2	0.739918	10/10	NonOverlappingTemplate
2	0	0	0	0	1	3	1	0	3	0.122325	10/10	NonOverlappingTemplate
1	1	1	0	3	2	1	0	0	1	0.534146	9/10	NonOverlappingTemplate
0	2	1	2	1	1	0	1	0	2	0.739918	10/10	NonOverlappingTemplate
4	0	1	2	0	0	0	2	1	0	0.066882	9/10	NonOverlappingTemplate
1	3	0	0	2	3	1	0	0	0	0.122325	10/10	NonOverlappingTemplate
0	2	1	1	1	0	2	0	2	1	0.739918	10/10	NonOverlappingTemplate
0	3	1	0	2	0	0	2	2	0	0.213309	10/10	NonOverlappingTemplate
1	2	1	0	0	1	2	1	1	1	0.911413	10/10	NonOverlappingTemplate
0	0	4	1	2	2	0	1	0	0	0.066882	10/10	NonOverlappingTemplate
2	1	0	2	0	1	0	1	1	2	0.739918	10/10	NonOverlappingTemplate
1	2	1	0	0	1	0	2	0	3	0.350485	10/10	NonOverlappingTemplate
1	0	0	1	1	1	0	0	2	4	0.122325	10/10	NonOverlappingTemplate
0	1	1	2	1	1	1	1	1	1	0.991468	10/10	NonOverlappingTemplate
1	1	6	1	0	0	0	0	1	0	0.000439	10/10	NonOverlappingTemplate
0	0	1	1	0	0	2	0	2	4	0.066882	10/10	NonOverlappingTemplate
2	1	0	0	2	0	0	3	1	1	0.350485	10/10	NonOverlappingTemplate
1	0	0	1	4	1	2	1	0	0	0.122325	10/10	NonOverlappingTemplate
0	1	0	2	1	0	3	1	1	1	0.534146	10/10	NonOverlappingTemplate
1	2	0	0	0	1	0	3	0	3	0.122325	9/10	NonOverlappingTemplate
2	2	0	2	0	1	1	1	0	1	0.739918	10/10	NonOverlappingTemplate
1	1	0	0	1	0	2	1	2	2	0.739918	10/10	NonOverlappingTemplate
0	0	0	3	2	1	2	1	0	1	0.350485	10/10	NonOverlappingTemplate
2	2	1	0	3	1	0	0	0	1	0.350485	10/10	NonOverlappingTemplate
0	1	1	0	1	2	1	2	2	0	0.739918	10/10	NonOverlappingTemplate
4	2	0	0	0	2	1	0	0	1	0.066882	10/10	NonOverlappingTemplate
0	0	1	0	0	3	0	2	2	2	0.213309	10/10	NonOverlappingTemplate
3	3	1	0	0	0	0	1	2	0	0.122325	10/10	NonOverlappingTemplate
2	0	0	0	2	1	2	1	2	0	0.534146	10/10	NonOverlappingTemplate
0	1	0	1	1	1	2	2	0	2	0.739918	10/10	NonOverlappingTemplate
1	0	3	1	0	1	1	1	2	0	0.534146	9/10	NonOverlappingTemplate
0	4	0	1	2	0	1	1	1	0	0.122325	10/10	NonOverlappingTemplate
0	1	2	0	0	1	2	1	0	3	0.350485	10/10	NonOverlappingTemplate
0	1	0	0	2	0	2	1	1	3	0.350485	10/10	NonOverlappingTemplate
0	1	0	0	1	3	1	1	2	1	0.534146	10/10	NonOverlappingTemplate
1	0	0	2	0	2	2	1	1	1	0.739918	10/10	NonOverlappingTemplate

```
0  1  1  1  1  1  0  2  0  3  0.534146  10/10     NonOverlappingTemplate
6  0  2  0  0  1  0  1  0  0  0.000199   7/10  *  OverlappingTemplate
2  1  2  0  0  1  0  1  1  2  0.739918  10/10     Universal
0  0  2  0  1  0  1  0  1  5  0.008879  10/10     ApproximateEntropy
1  1  1  1  1  1  0  1  2  0  ----       9/9      RandomExcursions
2  1  0  3  0  1  0  0  0  2  ----       9/9      RandomExcursions
1  0  0  3  1  0  1  2  1  0  ----       9/9      RandomExcursions
3  1  0  0  0  1  0  3  0  1  ----       9/9      RandomExcursions
1  1  0  3  1  1  1  0  0  1  ----       9/9      RandomExcursions
0  0  2  0  2  1  3  0  0  1  ----       9/9      RandomExcursions
0  0  0  1  0  2  1  1  2  2  ----       9/9      RandomExcursions
1  1  0  4  0  0  0  2  1  0  ----       9/9      RandomExcursions
0  1  0  1  1  1  3  0  0  2  ----       9/9      RandomExcursionsVariant
0  0  0  3  0  2  1  2  0  1  ----       9/9      RandomExcursionsVariant
0  0  2  0  3  0  1  1  1  1  ----       9/9      RandomExcursionsVariant
0  0  1  2  2  1  1  0  1  1  ----       9/9      RandomExcursionsVariant
0  0  1  1  3  1  2  1  0  0  ----       9/9      RandomExcursionsVariant
0  0  1  3  0  2  0  1  1  1  ----       9/9      RandomExcursionsVariant
0  0  2  0  0  2  2  1  2  0  ----       9/9      RandomExcursionsVariant
0  1  1  1  0  0  1  1  2  2  ----       9/9      RandomExcursionsVariant
1  1  0  0  2  1  0  0  4  0  ----       9/9      RandomExcursionsVariant
0  1  1  0  0  1  3  2  0  1  ----       9/9      RandomExcursionsVariant
0  1  1  2  0  2  0  1  2  0  ----       9/9      RandomExcursionsVariant
0  1  1  1  1  1  2  1  0  1  ----       9/9      RandomExcursionsVariant
0  0  3  1  0  1  1  1  0  2  ----       9/9      RandomExcursionsVariant
0  2  0  0  1  1  1  1  3  0  ----       9/9      RandomExcursionsVariant
1  0  1  0  1  1  0  2  2  1  ----       9/9      RandomExcursionsVariant
1  0  1  0  2  0  3  1  0  1  ----       9/9      RandomExcursionsVariant
0  0  3  0  0  0  3  1  1  1  ----       9/9      RandomExcursionsVariant
0  0  1  0  1  1  2  3  0  1  ----       9/9      RandomExcursionsVariant
0  1  1  0  0  1  0  1  1  5  0.017912  10/10     Serial
1  0  0  0  0  0  0  1  3  5  0.002043  10/10     Serial
1  1  2  1  0  1  0  2  0  2  0.739918  10/10     LinearComplexity
```

- -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 8 for a
sample size = 10 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately = 8 for a sample size = 9 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- -

B. Summary of Supporting Material

This appendix section lists all the material that is available externally to this report, and provides instructions for how to access it.

The software, as well as the results of all performed statistical testing, are available in the supporting documentation submitted. Instructions for executing the software are in the readme submitted as part of the supporting documentation. Results are stored in the `results` directory. The raw generator outputs are not included in the supporting documentation, since the 5GB of data far exceeds the upload size limit. Instead, the raw output can be accessed at the following link on the author's OneDrive file storage:

https://1drv.ms/f/s!Ah1ayeYeFEuDiJVILSrGN5x_iiKSsw

Should the link not work, the author can provide a new link upon request. The link does not allow editing. The link points to a directory tree structured as follows:

```
> results
    > discriminative
        > 01
            > after
            > before
            > plots
            > sequences
        > 02
            ...
        ...
    > predictive
        > 01
            > after
            > before
            > plots
            > sequences
        > 02
            ...
        ...
```

The numbered folders indicate the experiment. For each experiment, the subdirectories contain the results of statistical testing using NIST from before and after training, the logged training loss values, and the actual output sequences (the sequences with names prefixed by a 0 are generated before training, while the ones prefixed by 1 are generated after training). The same directory structure in included in the supporting documentation submitted directly, with exception of the `sequences` subdirectories as these are too large.