2022

Using Long Short-Term Memory Networks to Make and Train Neural Network Based Pseudo Random Number Generator

Aditya Harshvardhan
South Dakota State University, adityah835@gmail.com

USING LONG SHORT-TERM MEMORY NETWORKS TO MAKE AND TRAIN NEURAL NETWORK BASED PSEUDO RANDOM NUMBER GENERATOR

By

Aditya Harshvardhan

A thesis submitted in partial fulfillment of the requirements for the

Master of Science

Major in Computer Science

South Dakota State University

2022

THESIS ACCEPTANCE PAGE

Aditya Harshvardhan

This thesis is approved as a creditable and independent investigation by a candidate for the master's degree and is acceptable for meeting the thesis requirements for this degree. Acceptance of this does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department.

Kwanghee Won
Advisor Date

George Hamer
Department Head Date

Nicole Lounsbery, PhD
Director, Graduate School Date

This thesis is dedicated to my parents who supported and encouraged me throughout my degree program and my life.

ACKNOWLEDGEMENTS

I am grateful to my committee for their continued guidance and support. Dr. Sung Y. Shin, my committee chair and major advisor, Dr. Kwanghee Won, and Dr. Mostafa Tazarv. Please accept my appreciation for helping me learn and grow.

This thesis could not have been finished without you Dr. Won. Your eagerness to help and support me was very positive in going through the research in this “new” normal since the start of this pandemic. Also, sincere thanks to Dr. Tazarv for giving me an opportunity to work on a very interesting project to build User Interface for the Machine Learning Modules for the Civil Engineering Department.

Finally, I would like to thank my family, my father and my mother for keeping in touch and supporting me emotionally while being thousands of miles away and throughout my life.

TABLE OF CONTENTS

ABBREVIATIONS

AI	Artificial Intelligence
ASCII	American Standard Code for Information Interchange
CNN	Convolutional Neural Network
FC	Fully Connected
GAN	Generative Adversarial Network
Gen	Generator
LSTM	Long Short-Term Memory Recurrent Network
NIST	National Institute of Standards and Technology
NN	Neural Network
Pred	Predictor
PRNG	Pseudo Random Number Generator
ReLU	Rectified Linear Unit
Tanh	Hyperbolic Tangent

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

USING LONG SHORT-TERM MEMORY NETWORKS TO MAKE AND TRAIN NEURAL NETWORK BASED PSEUDO RANDOM NUMBER GENERATOR

Aditya Harshvardhan

2022

Neural Networks have been used in many decision-making models and been employed in computer vision, and natural language processing. Several works have also used Neural Networks for developing Pseudo-Random Number Generators [2, 4, 5, 7, 8]. However, despite great performance in the National Institute of Standards and Technology (NIST) statistical test suite for randomness, they fail to discuss how the complexity of a neural network affects such statistical results. This work introduces: 1) a series of new Long Short-Term Memory Network (LSTM) based and Fully Connected Neural Network (FCNN – baseline [2] + variations) Pseudo Random Number Generators (PRNG) and 2) an LSTM-based predictor. The thesis also performs adversarial training to determine two things: 1) How the use of sequence models such as LSTMs after adversarial training affects the performance on NIST tests. 2) To study how the complexity of the fully connected network-based generator in [2] and the LSTM-based generator affects NIST results. Experiments were done on four different sets of generators and predictors, i) Fully Connected Neural Network Generator (FC NN Gen) – Convolutional Neural Network Predictor (CNN Pred), ii) FC NN Gen - LSTM Pred, iii) LSTM-based Gen – CNN. Pred, iv) LSTM-based Gen – LSTM Pred, where FC NN Gen and CNN Pred were taken as the baseline from [2] while LSTM-based Gen and LSTM Pred were proposed. Based on the experiments, LSTM Predictor overall gave much consistent and even better results on the

NIST test suite than the CNN Predictor from [2]. It was observed that using LSTM generator showed a higher pass rate for NIST test on average when paired with LSTM Predictor but a very low fluctuating trend. On the other hand, an increasing trend was observed for the average NIST test passing rate when the same generator was trained with CNN Predictor in an adversarial environment. The baseline [2] and its variations however only displayed a fluctuating trend, but with better results with the adversarial training with the LSTM-based Predictor than the CNN Predictor.

1. INTRODUCTION

1.1 BACKGROUND

Random Numbers are used in many fields like information theory, probability theory, statistics, computer simulation, cryptography, pattern recognition, etc. [15]. Such numbers are generated by Random Number Generators (RNGs). These RNGs are of two types: 1) True Random Number Generator (or Random Number Generator [3]), 2) Pseudo Random Number Generator.[3]

A True Random Number Generator (TRNG) is a method of generating random numbers through physical process like thermal noise, phase jitter of oscillating signals and chaos [17]. This use of the physical sources results in greater randomness and unpredictability, meaning that these are almost impossible to predict since the source of randomness cannot be traced digitally through any cryptanalytic attacks. However, such TRNGs require dedicated hardware which are most often not available in a lot of computing systems [16].

On the other hand, a Pseudo Random Number Generator (PRNG) is an algorithm that outputs a sequence of values that exhibit randomness [9 pg. 112:3] by using a “seed” (initial machine state). Since these types of RNGs do not use any physical source, they are deterministic in nature, i.e., they have an inherent pattern to come up these potentially “random” numbers [9]. Despite that they are still used since they can be used to create random numbers in limited resource devices like IoT (Internet of Things) devices [9]. They are also used in making computer games [11].

In application security aspect of the PRNGs, operating systems like Windows [13] use PRNGs to create pseudo-randomness for ensuring security in any running processes

and applications. On the other hand, in communication, it is necessary for any low-end computing devices like the Internet of Things (IoT) devices [9]. Some use cases of pseudo randomness are transmitting low-level packets (pieces of information that travel over wired or wireless networks) and performing Artificial Intelligent (AI) algorithms [9]. To ensure safety and security during these operations, PRNGs are used to provide a starting random source for the communication encryption.

When it comes to game development, PRNGs are mostly used to create an environment or a game mechanic (level design, world rendering) through procedural generation [11]. Procedural generation is a technique where game developers make use of curated algorithms to make any aspect of game design (e.g., map generation, character generation, story, etc.) lwith ease, without having to spend hours working to handcraft such experiences for gamers [19].

Most of the existing PRNGs are hand-crafted [7, 13, 18] which have specific state-space in the algorithm or how the values are calculated (multiplication, modulo, etc.). The "seed" provides a starting point for the algorithm which then goes through a developer/researcher curated algorithm to produce 1 to n (where n is an integer) values as output.

Fig. 1: PRNG I/O Flow

While proposing PRNGs, one must also make sure that it passes statistical tests for randomness [3]. There are a lot of statistical test suite namely, NIST [3], TestU01 [20], and many more. Although TestU01 [20] is a very robust test for PRNG (introduced in 2007), however the newly updated (in 2010) NIST test [3] is the used in this thesis for testing randomness to allow a relevant comparison with previous works.

1.2 MOTIVATION AND OBJECTIVE

As seen in the previous section, the idea of so many things possible by Pseudo Random Number Generator is intriguing. Extensive work on this has been done using complex hand-crafted methods [18, 21]. However, recent works [2, 4, 5, 7, 8] have also shown the potential of developing new PRNGs without the need to completely handcraft it. These works make use of Neural Networks either partially or entirely in their PRNG algorithm. If these new works can be investigated upon carefully, one can also measure which Neural Network model parameters are responsible for being able to create such state-of-the art PRNGs.

The Long Short-Term Memory Networks (LSTM) have shown a good pattern recognizing ability [10]. These LSTMs were utilized in [7] where the authors used it as a part of a complex hand crafted PRNG algorithm. So, it will be interesting to see how LSTM-based networks would affect randomness (based on NIST test) both in value generation (PRNG) and prediction (Predictor) (used in adversarial training - pg 10 2.1.5 Generative Adversarial Network). However, in order to test the generators for cryptographic applications, it must not only pass statistical tests, but also perform multiple cryptanalytic tests to verify the model's robustness. This, however, is not in the scope of

this paper and the focus is on comparing the NIST statistical test results since these tests themselves can show the quality of randomness for most use cases. This thesis focuses on understanding how network complexity (number of layers in the network) affect the NIST results. This thesis takes the Fully Connected (FC) Neural Network Generator (for proposing baseline [2] variations) and Convolutional Neural Network (CNN) Predictor proposed in [2] as reference models.

In summary, main objectives of this research are to find out: 1) How the sequence models (e.g., LSTMs) affect the results from the NIST test suite for randomness. 2) How network complexity, especially the number of layers of the neural network generator affects the NIST results after going through adversarial training process (See Sections 2.1.5 and 3.1) [1, 2].

1.3 CONTRIBUTIONS

This research contributes in two ways. First, with the introduction of the new LSTM-based generator and predictor, the work introduces how involving the LSTM units in an adversarial training can result in a PRNG with great NIST results. In case of the LSTM-based Generator, the LSTM layer adds a degree of complexity to the network. On the other hand, the LSTM Predictor is introduced to exploit the LSTM's strength in solving cognitive tasks (here, sequence data) [10 pg. 33] to see if the generated number show any signs of repetition or pattern.

Second, the training is done with four combinations of the generator and predictor. Two of which are from [2] and two are proposed. Also, the experiments dissect the

generators in these experiments to determine how the increase or decrease in the number of layers in the network affects the NIST results.

1.4 THESIS ORGANIZATION

This thesis after chapter 1, consists of the following chapters. Chapter 2 is the ‘Literature Review’ which includes ‘Concept Review’ of important concepts and 'Related Work’ on Neural Network based PRNGs. Chapter 3 introduces the ‘Design and Implementation’ of the generators (PRNGs), both proposed and baseline [2] variations, the proposed and baseline [2] predictors, the activation functions, adversarial training setup to train the generators. Chapter 4 discusses the experiments done using the different generator-predictor combinations after which Chapter 5 discusses the results and makes a conclusion. Finally, in Chapter 6, I discuss the limitations and future work

2. LITERATURE REVIEW

2.1 CONCEPT REVIEW

2.1.1 Perceptron: Introduced in 1958 [24], the "perceptron" focused on creating a probabilistic model of how visual information is organized and stored in higher order organism's brain like humans. In other words, the author proposed an analogous model that can take learn from the visual data to classify the said visual stimulus just like the human vision and neural system. The diagram below (Fig. 2) shows the organization of a photo perceptron (one that takes in visual input). The inputs (light or no light) (S-points) from the cells of retina goes to the 'Projection Area' (A_I) of the Association Cells (A-units). These inputs may either 'excite' or 'inhibit' the A-unit. If the sum of 'excite' and 'inhibit' states becomes more than the threshold, then A-unit fires with all-or-nothing value (0 or 1). The connection between the 'Projection Area' (A_I) and 'Association Area' (A_{II}) is random (only forward connection). The responses from A-unit are then received by the Responses (R-unit) cells R_1, R_2, ..., R_n. The connection from the A-units to the R-units are also random (bi-directional connection).

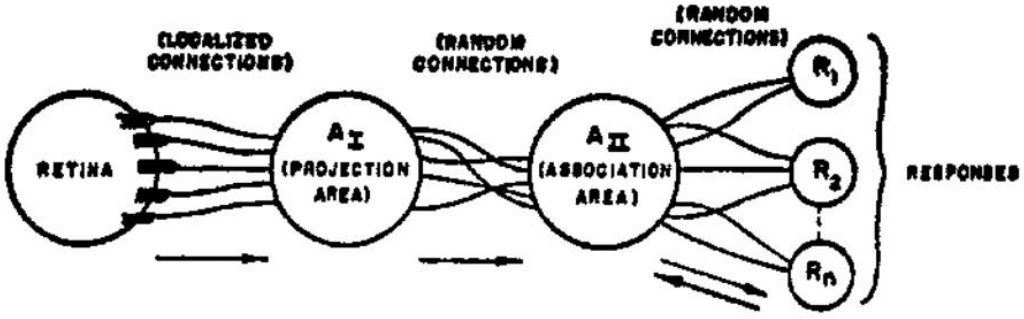

Fig 2: Organization of a photo-perceptron (Courtesy: [24])

The perceptron is in fact an enhanced Threshold Logic Unit (TLU) [25 pg. 70]. The idea is that inputs to the neuron are multiplied by the weight values which are then added. This weighted sum then goes through an activation function, f(s) to give an output.

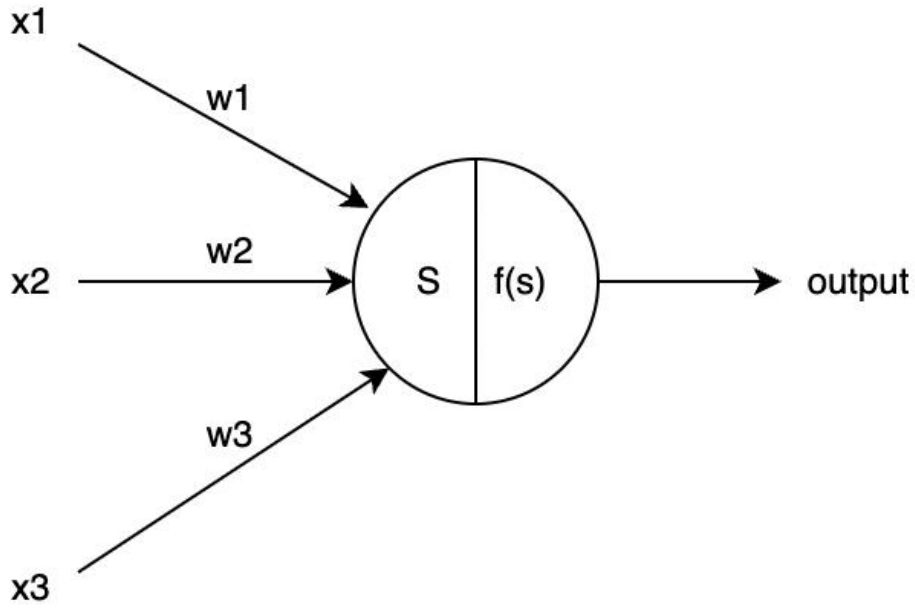

Fig 3: A Perceptron (Neuron) [25]

2.1.2 Fully Connected Neural Network (FC NN): The Fully Connected Neural Network is a class of Feed Forward Network using multiple neurons in multiple layers (Multi-Layer Perceptron) [25 pg. 159]. Below are the Single Layer Perceptron (Fig. 4) and Multi-Layer Perceptron or MLP (Fig. 5). In Fig 4, each neuron is fed 3 inputs which then get multiplied with the weight values (the w-values in Fig 4) and pass through the activation (part of the circle representing neuron + activation) to produce same number of outputs as the number of neurons. On the other hand, Fig 5, each neuron (perceptron) is connected to the other neuron in the next layer. An MLP having more than two layers has three types of layers: Input layer (takes the input values ‘x1’, ‘x2’, ‘x3’), output layer (gives out one (here

it is 'y') or multiple outputs based on number of neurons in the layer), and hidden layer (layers in between input and output layer).

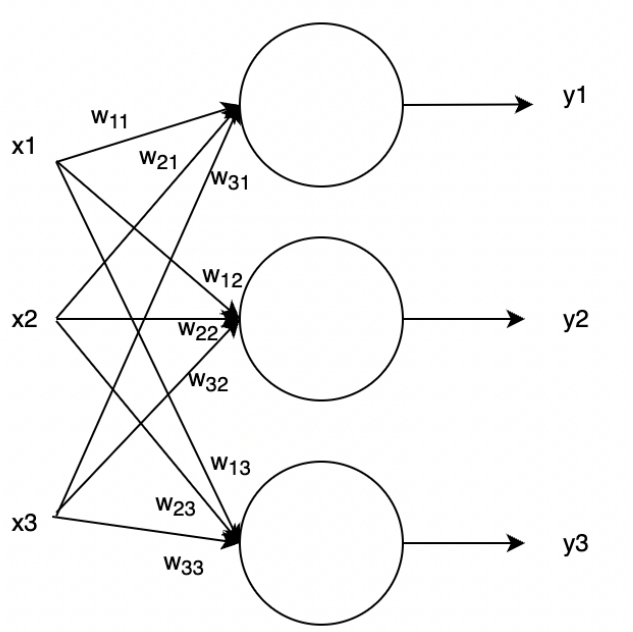

Fig 4 A Single Layer Perceptron [25]

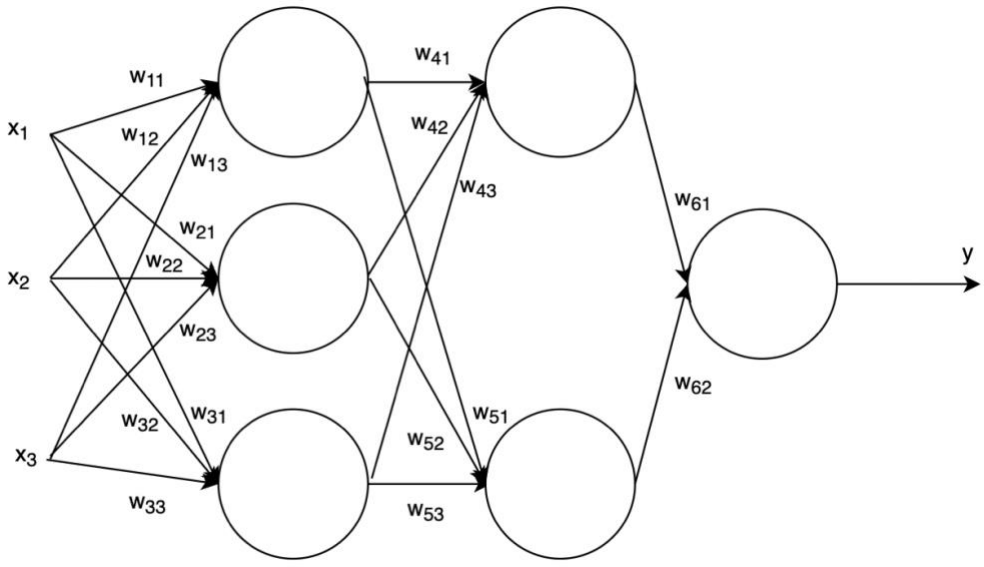

Fig 5 A Multi-Layer Perceptron [25]

2.1.3 Feed Forward and Back Propagation: In a Feed Forward Network like MLP, the direction of the neuron connections is only one way, from layer 1 to n. To train the neural network, the difference between real and predicted value should be propagated back as ‘loss’ or ‘gradient’ to update the weight values for a better prediction in the next run. The algorithm to calculate the error gradients is called ‘Back Propagation’ [25 pg. 98].

2.1.4 Convolutional Neural Network (CNN): While the Fully Connected Neural Network had great performance on numerical dataset, complicated datasets like images caused a huge increase in number of parameters needed, slowing convergence [26]. So, Convolutional Neural Networks (CNNs) were introduced to solve these problems [27]. Each neuron in a CNN is connected to a small number of neurons from the previous layer (local connections) [26, 27]. The CNN also allow same weights to be shared among the connected groups of neurons [27]. Also, the introduction of a pooling layer causes the decrease in the image size (down sampling) [27]. All these factors result in much less parameter and faster convergence when dealing with complex data like images. The figure below shows the first CNN architecture (LeNet-5) used for object detection [27].

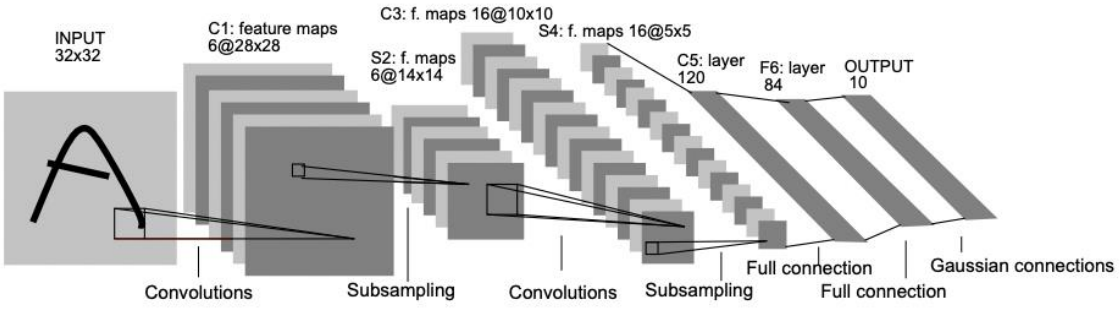

Fig 6: LeNet-5 Architecture (Courtesy [27])

In Fig 6, the CNN is used for digit recognition. The image (32 x 32) features are extracted through the local neuron units (filters) which only know about the region they

scan. The extracted features are numeric values between 0-255 (grey scaled image) in stacked 2-dimensional matrices called *feature maps* (6 maps each of size 28 x 28). After that only the high-level features are taken from the said feature maps through *sub sampling*, which reduce the spatial resolution or size of the matrices (6 maps each of size 14 x 14) containing the feature information. This process repeats again after which the feature maps are converted into a Fully Connected Convolution Layer which then goes to the Fully Connected Neural Network (MLP) hidden layer and finally a neural network output layer with 10 neurons. The last layer neurons contain different probability values for each of the 10 classes indicating what the input image is closer to.

2.1.5 Generative Adversarial Network (GAN): Generative Adversarial Networks are class of generative models which were first introduced to create realistic sample images by training the 'generator' against a 'discriminator' [1]. The generator G tries to create a sample from random noise. After that the output to the generator is fed to the discriminator D, which classifies the output from G being from the real distribution or the from the noise. The job for G is to ensure that the loss for G is minimum but D wants the loss of G to be maximum. Eventually the generator outsmarts the discriminator, and the discriminator cannot distinguish between the real data from the generated data.

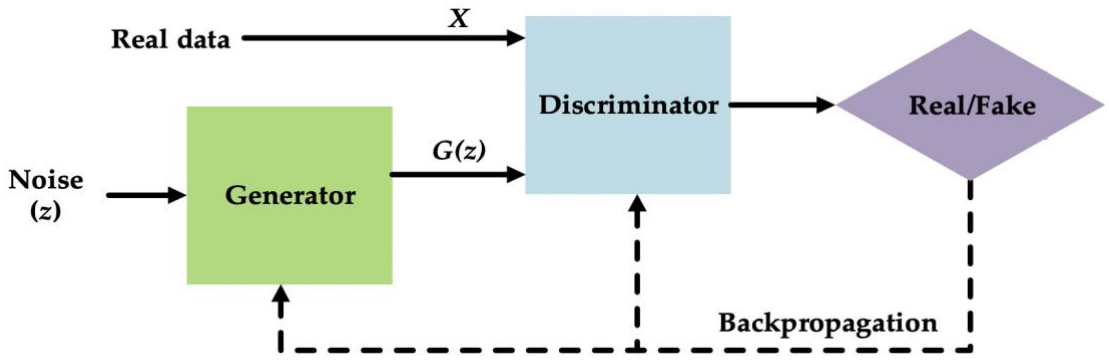

Fig 7: Original GAN [1] (Image Courtesy [31])

2.1.6 Long Short-Term Memory Network (LSTM): The Recurrent Neural Networks (RNNs) were introduced to add the concept of memory [28] to the regular Fully Connected Neural Network to store any context information when going through the sentence to predict the next word. These RNNs had a "vanishing gradient problem" (error too close to 0) or the blown-up (very high) error values when theses RNNs were long. To solve this long-term dependency issue, Long Short-Term Memory Networks (LSTM) was introduced [10, 28]. The figure below (Fig 7) shows a typical LSTM Cell. These individual cells are connected in different ways to create a LSTM.

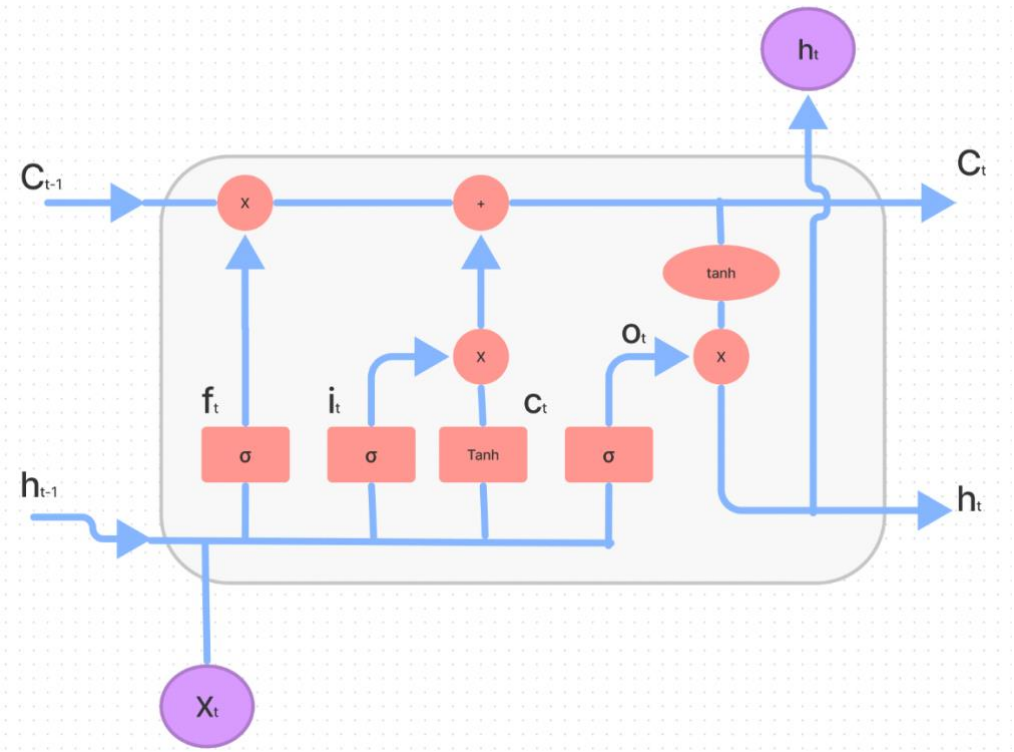

Fig 8: An LSTM Cell [28]

In the above diagram (Fig 7), the LSTM Cell takes output from previous LSTM Cell or hidden state (h_{t-1}), memory from previous cell, C_{t-1}, and current input X_t as inputs. The h_{t-1} and X_t together through three gates, forget (f_t), input (i_t) and output (o_t) gates. The forget gate decides which information should be forgotten. The input gate ensures the information to remember and output gate is used to decide what should be the output. All these gates also include sigmoid function that output value between 0 to 1 to denoted percentage of each of the operations. The Tanh is used together with sigmoid for updating cell state Ct as well as for the changing the cell output h_t.

2.1.7 Rectified Linear Unit (ReLU): This activation function converts an input x to an output ranging between [0, inf). The equation for ReLU is [29]:

$$y = \max(0, x)$$

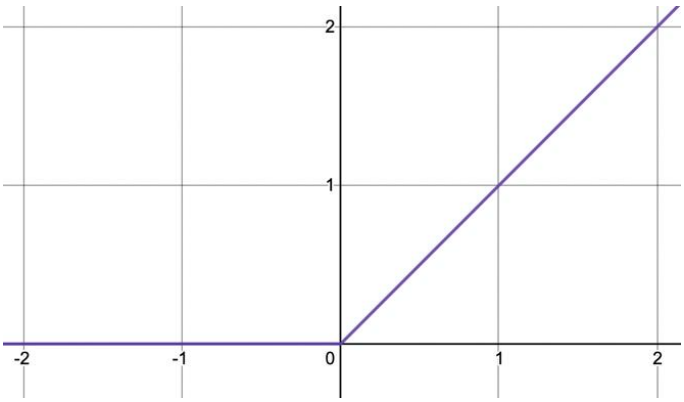

Fig 9: Rectified Linear Unit (ReLU) [29]

2.1.8 Leaky Rectified Linear Unit (Leaky ReLU): Using ReLU in some cases causes "Dying ReLU" issue, where any negative values are lost due to ReLU returning 0 for x < 0. This increases learning time for the model. A variation of this, Leaky ReLU does not completely zero out any negative values. Leaky ReLU allows negative values to pass through after being multiplied by a factor 'alpha' (α). The equation for Leaky ReLU is [29]:

$$y = \begin{cases} \alpha * x & if\ x \leq 0 \\ x & if\ x > 0 \end{cases}$$

where α=0.01 in most cases but can be adjusted

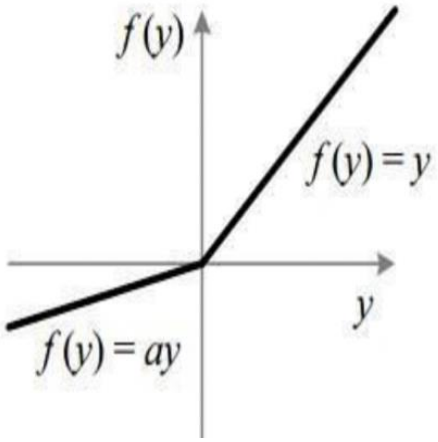

Fig 10: Leaky ReLU (Image Courtesy [29])

2.1.9 Hyperbolic Tangent (Tanh): This activation function converts an input x to an output ranging between (-1, 1). The equation for Tanh is [29]:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

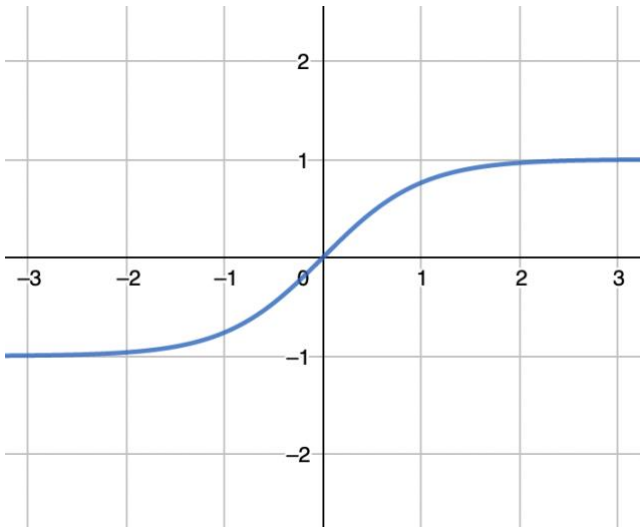

Fig 11: Hyperbolic Tangent (Tanh) [29]

2.1.10 Sigmoid (σ): This activation function converts an input x to an output ranging between (0, 1). The equation for sigmoid is [29]:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

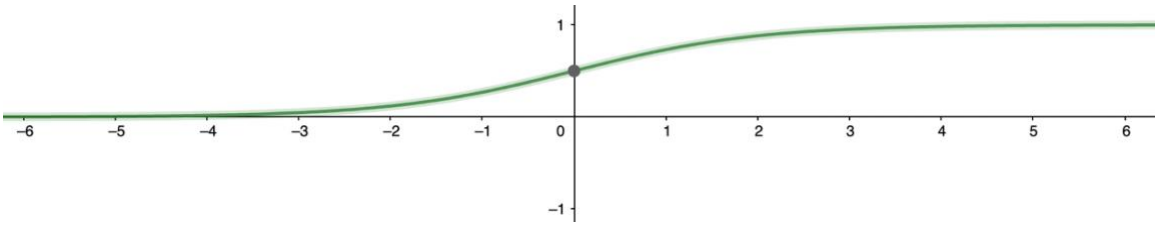

Fig 12: Sigmoid Function [29]

2.1.11 NIST Statistical Test Suite: This test suite was compiled by the National Institute of Standards and Technology (NIST) which consists of 15 tests to check the randomness of the sequences generated by the Pseudo Random Number Generator (PRNG) [3]. These tests were designed to determine many types of patterns or non-randomness

properties present in the generated sequence. The input sequence is taken by these tests in two forms: a Binary file or an ASCII file (.txt) having sequence numbers as a series of 0s and 1s. Below are the tests explained with their purpose [3]:

The Frequency (Monobit) Test: This is a preliminary test done to determine whether the generated sequence is fit for any subsequent tests since a failure would mean that there is no randomness in the produced sequence [3]. This test does this by checking the proportion of 0s and 1s in the sequence. Since a random sequence would have about same number of 1s and 0s, so the test checks if the number of 1s is close to ½ and number of zeroes to be approximately equal.

Frequency Test within a Block: This test checks how many 1s are present in the M-bit blocks. If there are equal number of 1s and 0s within the M-bit block from the entire sequence, the sequence passes this test. If M=1, then it is the same as Monobit test (Frequency) [3].

The Runs Test: This test is done to how often the 0s and 1s are oscillating [3]. If the oscillations are too fast or too slow, then the sequence may have a pattern (non-randomness) and hence fail the test.

Tests for the Longest-Run-of-Ones in a Block: This determine whether an irregularity in the longest run of 1s in the sequence in an M-bit block. [3]

The Binary Matrix Rank Test: This test checks if there is a linear dependence in the substrings of fixed length from the entire output sequence [3].

The Discrete Fourier Transform (Spectral) Test: The test determines if there are similar patterns near each other. If that is the case, then the assumption of the sequence being random is incorrect and test fails [3].

The Non-overlapping Template Matching Test: This test detects if the provided non-random patterns are found multiple times in the m-bit (m number of bits from the entire sequence) sequence. If the pattern is not found, the m-bit window slides one bit from its current position. The test fails if there are multiple occurrences of this aperiodic pattern [3]

The Overlapping Template Matching Test: This test detects if the provided given patterns occurring multiple times in the m-bit (m-bit from the entire sequence) sequence. If the pattern is found, the m-bit window slides one bit from its current position. The test fails if there are multiple occurrences of this pre-specified pattern. [3]

Maurer's "Universal Statistical" Test: This test determines whether the sequence can be compressed down to smaller size without losing any sequence information. If the sequence can be compressed considerably, then the sequence is not random. [3]

The Linear Complexity Test: This test determines the length of Linear Feedback Shift Register. If the length of the LFSR is too short, then it is a non-random sequence. [3]

The Serial Test: This test checks if the number of 2^m m-bit overlapping patterns are about the same in the entire sequence. If there is any non-uniformity or unequal chance of one m-bit pattern over the other, then it is not a random sequence [3]. With m = 1, it is the same as Frequency test.

The Approximate Entropy Test: The test is done to determine the frequency of the 2^m m-bit overlapping patterns. This is different that Serial test since it compares the overlapping block of consecutive lengths like m and m+1 bits and see if the patterns are equally likely just like a random sequence [3].

The Cumulative Sums (Cusums) Test: This test conducts a random walk test from the 0. 1 is considered as “forward” or +1 and 0 is considered “backward” or -1. This test

then determines if the cumulative sum of the partial sequences performing random walk is 0 or close to 0 [3].

The Random Excursions Test: This test checks if the within a cycle (unit length steps taken from origin and back to it), number of visits deviates from the expected random sequence. This can be divided into 8 different tests with states as -4, -3, -2, 1, +1, +2, +3, +4, each having a test and conclusion [3].

The Random Excursions Variant Test: The test determines if total number of visits on the specific state in a Cusum walk [3]. This can be broken into 18 tests along with conclusions for states: -9, -8, ..., -1, +1, ..., +8, +9.

2.2 RELATED WORK

Multiple works have shown that using Neural Networks to create practical PRNGs is possible [2, 5, 7, 8, 23]. The work from Marcello et al. [2] was the first one to attempt in creating a cryptographically secure PRNGs using Fully Connected Neural Network through an "end-to-end" adversarial training [1], like Generative Adversarial Networks [2]. The work makes use of two different types of approaches: Adversarial and Predictive. In the Adversarial approach, the Generator and Discriminator compete where the Discriminator would classify the sequence of the 8 generated values [x_1, …x_8] into 0 or 1 (0 – generated and 1 – real). The Mean Square Error [30] losses are calculated for both generator and discriminator. On the other hand, in the Predictive approach, the setup is the same. The main difference is that instead of feeding the Predictor the entire generator output, only 7 out of 8 values are given as input. The Predictor then goes through the sequence as it goes through each Convolution, Max Pool and finally through the Fully

Connected layers to finally output the predicted value next in the sequence. The loss for the generator would be higher if Predictor gives a value very close or equal to the actual value in the generator output (last value in sequence). Inverse is the case for Predictor loss, being more as predicted value is further away from the actual value. The NIST results based on the training setup (200,000 epochs, 2048 mini-batch) is 95% overall pass rate compared to other general purpose PRNGs.

Another work from [7], makes use of a Long Short-Term Memory (LSTM) to copy the behavior of irrational numbers (ex. pi). But it does not use LSTMs completely as a solution, rather a part of the complex algorithm to generate sequence in an iterative manner. The curated algorithm ensures that no repetition is present in the generated sequence by taking seed, buffer and input sequence as input. This input then goes to the Iterative Generator containing an LSTM trainable module which learns the nature of irrational numbers. Rest of the generator consists of "seed shift", "Buffer sequence update" and "Input Sequence". The output (random number) is added to the output sequence *prn* which then goes to the Secure Hash Algorithm 2 (SHA-2). This SHA-2 is what creates binary sequence as output which is then fed to the NIST test suite program to test the quality of the output sequence. The results from the NIST test suite indicate that the algorithm can be useful in parameter approximation in machine learning.

Other work [5] has a different take type of neural network training. The paper models the PRNG as agent present in a Markovian Decision Process (MDP) [22] in a Reinforcement Learning environment (USiennaRL) containing: i) A set of states, *S* and distribution of starting states $p(s_0)$, ii) A set of actions *A*, iii) Transition dynamics to map state-action at time t, iv) Instant Reward Function, *R*, v) A Discount factor (lower values

means more focus on instant reward). A bit length is chosen for sequence (state space *B*). So, the MDP is $|S| = 2^B$, $|A| = 2B$. The reward function *R*, the NIST test suite result is used as reward after each step for MDP. As for the model architecture [5], two LSTM layer are employed to avoid an exponential increase in the state-space while the RL-training. The model network then splits into a policy function and value function subnetworks, each with FC layers of 256, 512, 256 neurons in order (ReLU activation). The work has shown great NIST results for long sequences.

Taking all these works as reference, especially the works from [2] and [7], this thesis proposes a series of LSTM-based generators due to the network complexity (Number of layers) experiments. As for the LSTM predictor, the pattern finding nature of the LSTM [10] is an inspiration for proposing a predictor for the experiments conduct in this research both for the series of dissected Fully Connect Neural Network PRNG from [2] and the proposed LSTM-based generator.

3. DESIGN AND IMPLEMENTATION

3.1 ADVERSARIAL SETUP

The training process aims to minimize the possibility of predictor successfully predicting a value same as or close to the real value. The training idea is taken from [1, 2], where the generator initially generates easily predictable values. However, as the training goes for a given pair of seed and offset through more epochs/steps, the generator becomes cleverer and starts producing diverse values. Eventually, the predictor is not able to exactly guess the value and the generator "wins" this tug-of-war between itself and the predictor [1, 2].

The Fig 11 below shows the Adversarial training flow in one epoch (step). During each epoch, one (seed, offset) vector is taken from in the input evaluation dataset is fed to the generator (either one of the proposed LSTM-based or baseline [2] + variation Generators). The generator outputs 8 values ({n1, ..., n8} for each {seed, offset} from the input dataset) which then goes through split method (function) in a 7-1 split. The training aims at updating the weights for generator and predictor. With each generated 7-1 split output, the vector with 7 values ({n1, ..., n7} for each {seed, offset} from the input dataset) is fed to the predictor (LSTM or baseline [2] CNN Predictor) which then comes out as 1 "predicted" output. Then the real value n8 (last value from 7-1 split of the generator output) is compared with the Predictor output p0 through Predictor Loss (See Section 3.4) 3 times each epoch.

Then the second part is when generator weight is updated. Predictor once again is used to generate a few predicted values (one value p0 for each {seed, offset} from the input dataset). Then generator loss is calculated once every epoch. Finally, one epoch is completed, and the entire process is repeated till the last epoch.

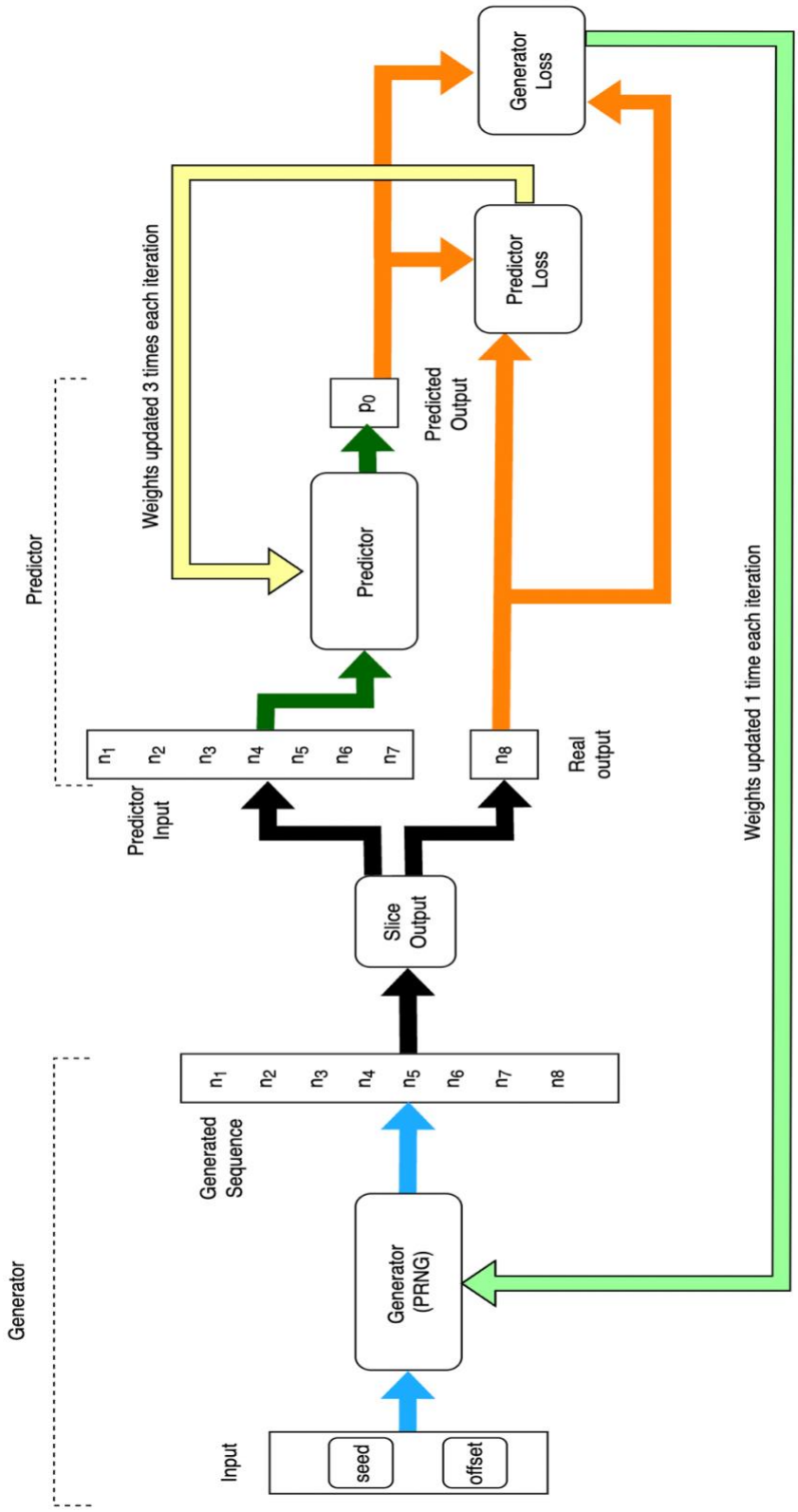

Fig 13: Adversarial Training Flowchart (in 1 step)

3.2 GENERATORS

The generators use the same input and output configuration as mentioned in [2]. The experiments on the generators are done with 16-bit unsigned integers. The inputs to the generators are the seed s and offset o. After going through different layers of these generators, the output is a Tensor of 8 unsigned 16-bit integers. Below are the two types of generators used in this paper:

3.2.1 LSTM-based generator: The LSTM-based generator is a modification of the fully connected generator seen in [2]. The first difference is the change in the default activation function of the forget (f_t), input (i_t) and output (o_t) gates from sigmoid (σ) to Leaky ReLU to avoid the LSTM cell to completely ignore any negative values.

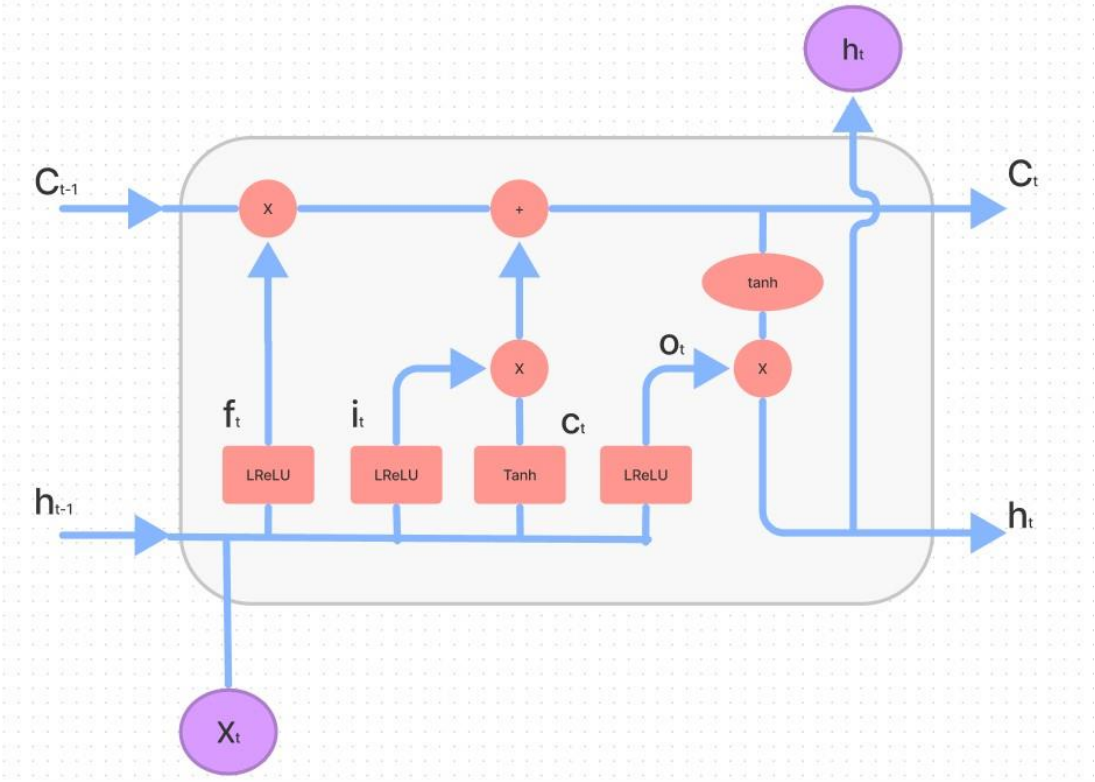

Fig 14: Modified LSTM Cell of the proposed LSTM-based Generator

The proposed generator starts with an LSTM layer, which accepts seed and offset values and outputs 32 values (through 32 identical LSTM units one of which is shown in Fig 12). Here, each output value is obtained from an individual LSTM Cell. After the LSTM layer, these 32 values through the Fully Connected layers. The fully connected (FC) layer in this class of the generator is also modified to have 32 neurons instead of 30 as in [2]. In all the variations of the LSTM-based generator (except the generator with LSTM + 1 FC Layer), the hidden layer has a Leaky ReLU activation to avoid the vanishing gradient problem [2, 10]. The last Fully Connected layer uses the Modulo activation which is the same as [2] to ensure that values are within the expected range of 0 – 65535. All the LSTM-based generators used in the experiments are highlighted in Fig 12a – Fig 12d.

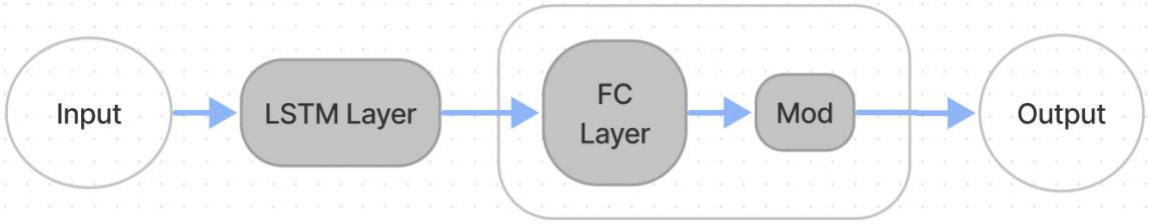

Fig 15a: LSTM-based Generator (LSTM Layer + 1 FC Layer)

Model unit	**Parameters**
Input	(seed, offset)
LSTM Layer	32 LSTM Cells (Fig 12)
FC Layer	32 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 1a: LSTM-based Generator (LSTM Layer + 1 FC Layer) Parameters

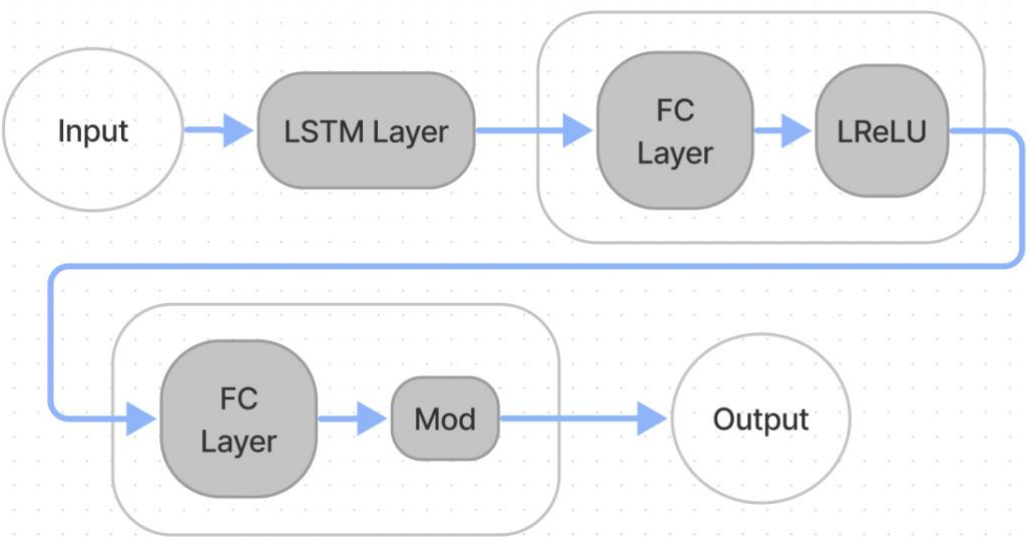

Fig 15b: LSTM-based Generator (LSTM Layer + 2 FC Layer)

Model unit	**Parameters**
Input	(seed, offset)
LSTM Layer	32 LSTM Cells (Fig 12)
FC Layer	32 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 1b: LSTM-based Generator (LSTM Layer + 2 FC Layer) Parameters

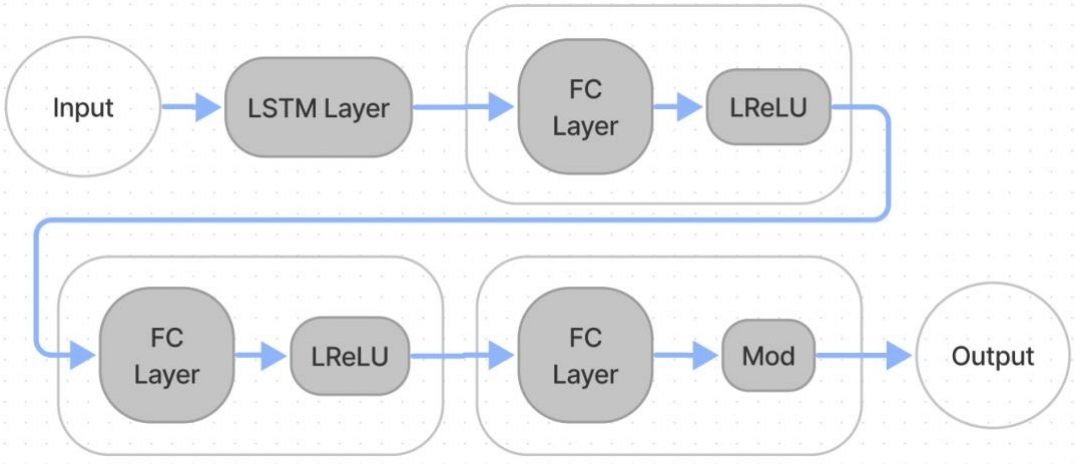

Fig 15c: LSTM-based Generator (LSTM Layer + 3 FC Layer)

Model unit	Parameters
Input	(seed, offset)
LSTM Layer	32 LSTM Cells (Fig 12)
FC Layer	32 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 1c: LSTM-based Generator (LSTM Layer + 3 FC Layer) Parameters

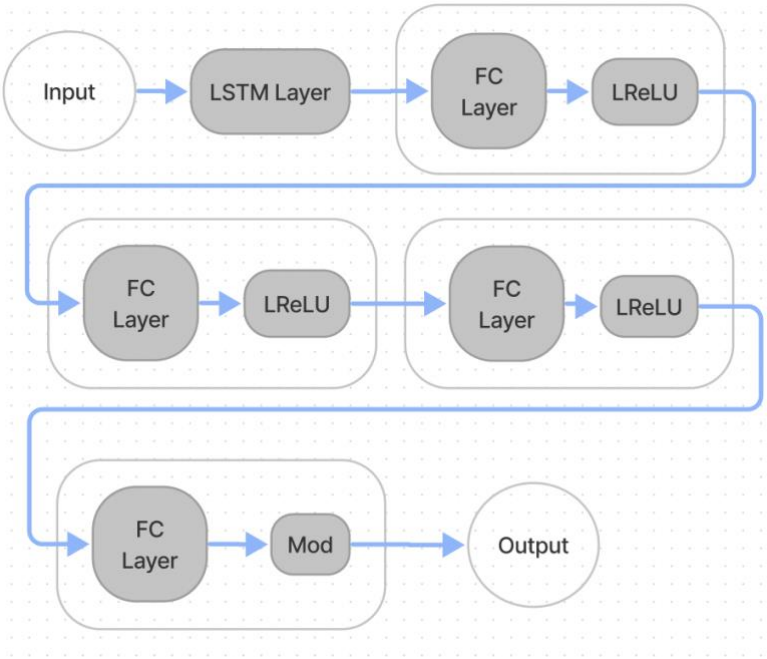

Fig 15d: LSTM-based Generator (LSTM Layer + 4 FC Layer)

Model unit	Parameters
Input	(seed, offset)
LSTM Layer	32 LSTM Cells (Fig 12)
FC Layer	32 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 1d: LSTM-based Generator (LSTM Layer + 4 FC Layer) Parameters

3.2.2 Fully connected Neural Network generator (baseline + variations): All the fully connected (FC) generators take seed and offset values as input which then goes to the hidden layers. Each FC layer consists of 30 neurons where every layer except the last layer has a Leaky ReLU activation. The last layer has Modulo activation to ensure that values are within the range of 0 – 65535. Below are all the FC generators (Fig 13a – 13g), out of which generator depicted in Fig 13d is the baseline taken from [2]. The rest of the generators are variations of the baseline, each having different number of the layers for conducting the experiments to see how the network complexity (number of layers) and its effect on the NIST test results.

* Note: Due to non-availability of the updated source code for the generator and predictor proposed in [2], best effort was made to reproduce the model (Fig 13d) from [2] in this paper

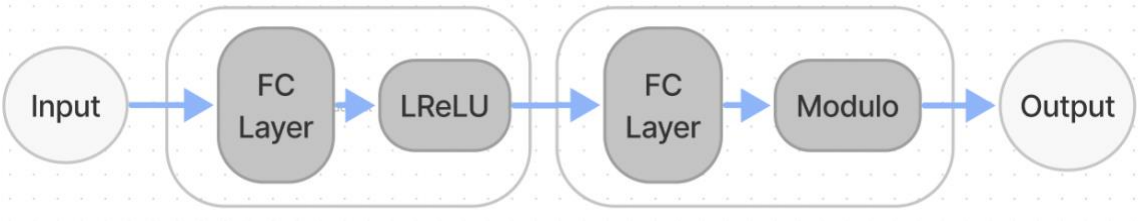

Fig 16a: Fully Connected Generator (2 layers)

Model unit	**Parameters**
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2a: Fully Connected Generator (2 Layers) Parameters

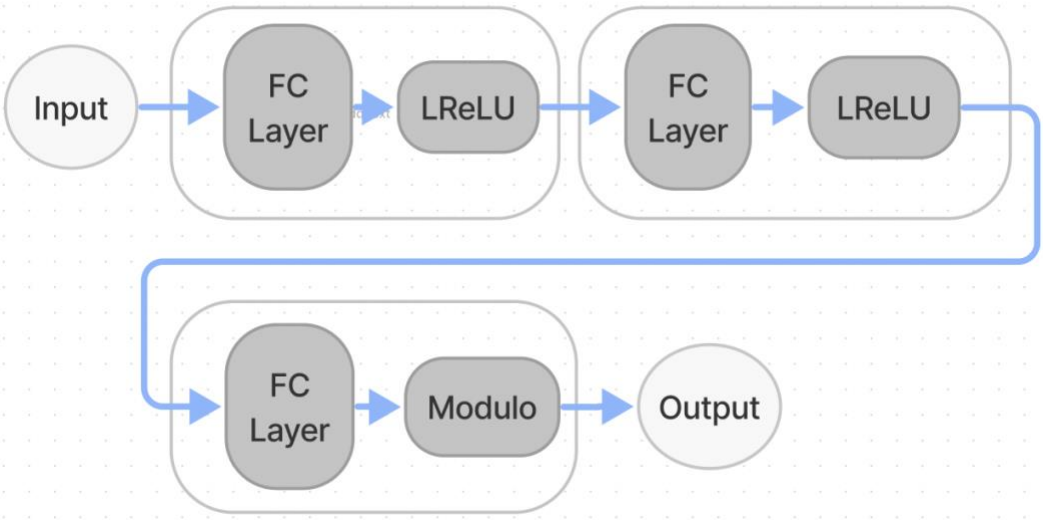

Fig 16b: Fully Connected Generator (3 layers)

Model unit	**Parameters**
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2b: Fully Connected Generator (3 Layers) Parameters

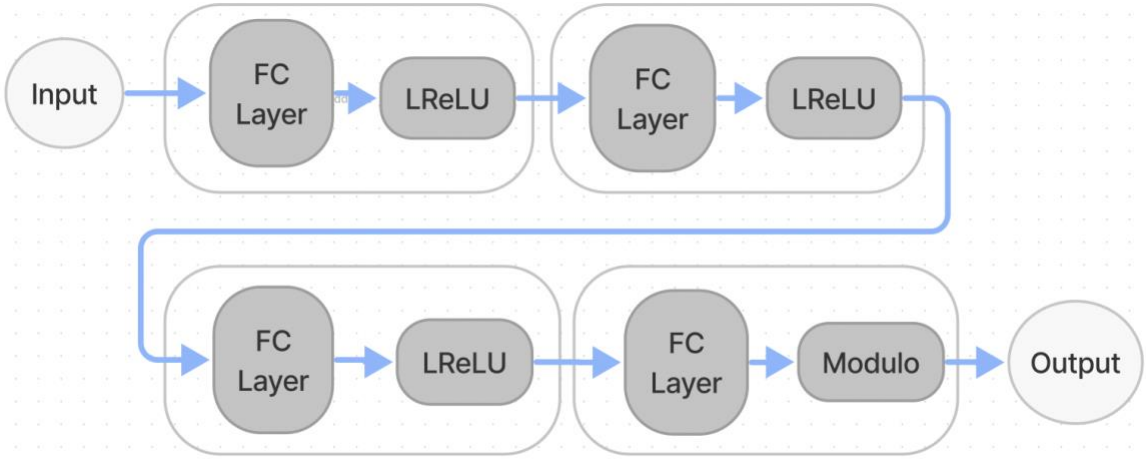

Fig 16c: Fully Connected Generator (4 layers)

Model unit	Parameters
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2c: Fully Connected Generator (4 Layers) Parameters

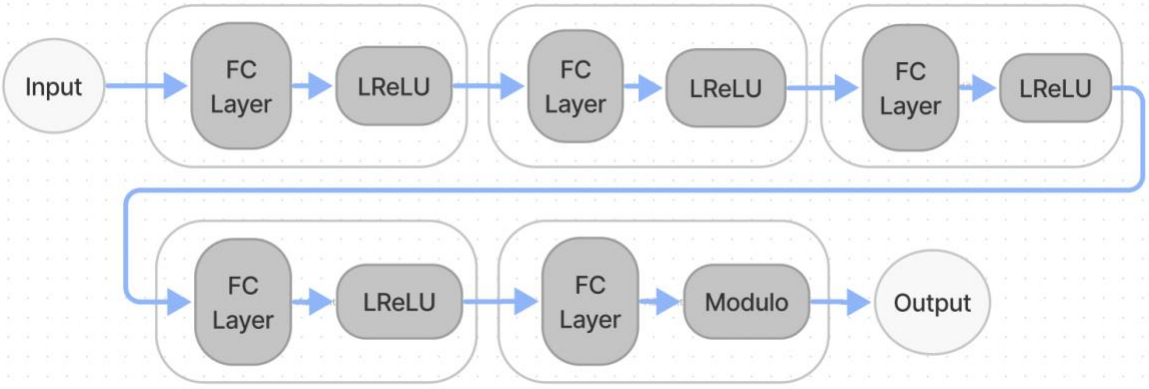

Fig 16d: Fully Connected Generator (5 layers) – baseline [2]

Model unit	Parameters
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2d: Fully Connected Generator (5 Layers) – baseline [2] Parameters

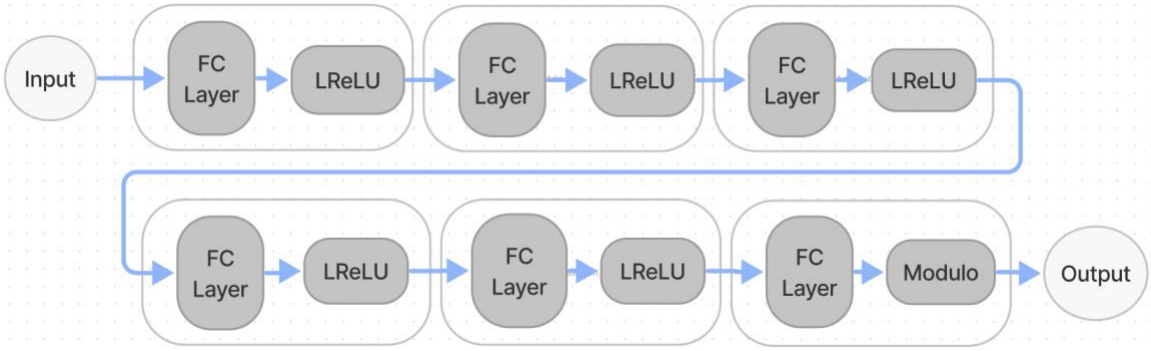

Fig 16e: Fully Connected Generator (6 layers)

Model unit	**Parameters**
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2e: Fully Connected Generator (6 Layers) Parameters

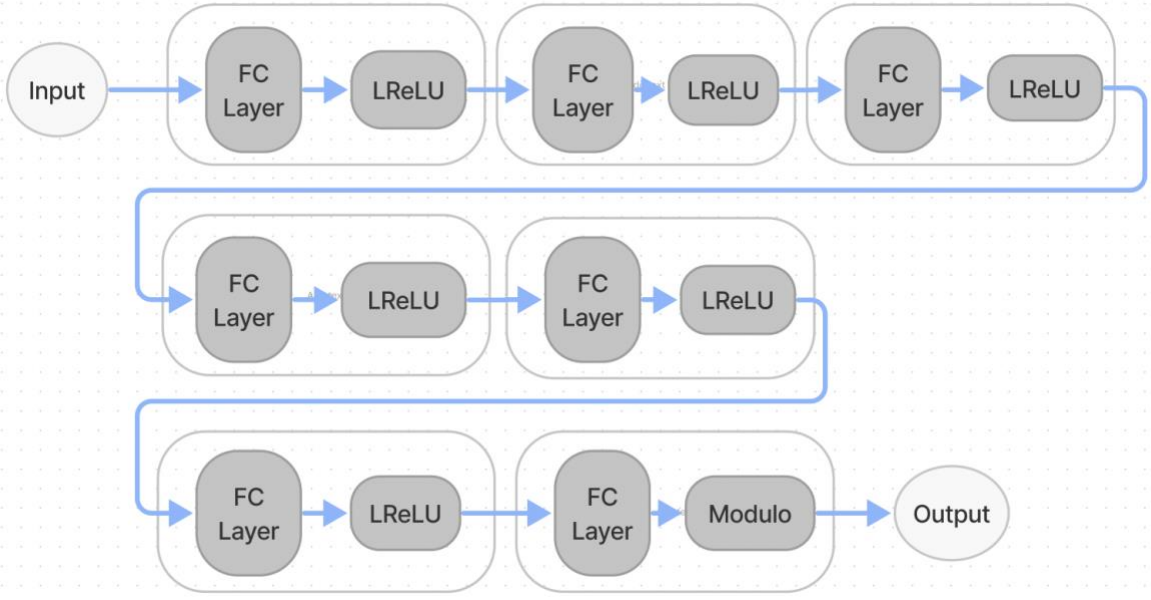

Fig 16f: Fully Connected Generator (7 layers)

Model unit	Parameters
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2f: Fully Connected Generator (7 Layers) Parameters

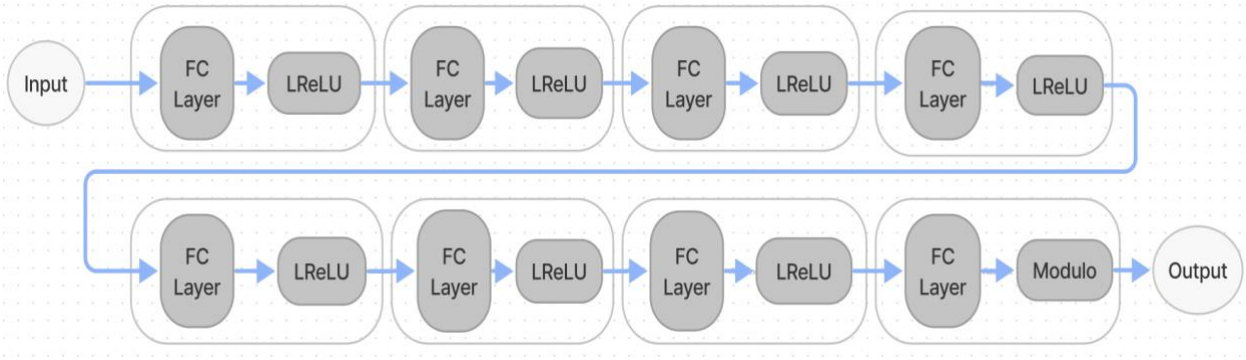

Fig 16g: Fully Connected Generator (8 layers)

Model unit	Parameters
Input	(seed, offset)
FC Layer	30 neurons
Output	Vector of 8 values for every (seed, offset) pair

Table 2g: Fully Connected Generator (8 Layers) Parameters

3.3 PREDICTORS

The input and output configuration for both baseline Convolutional Neural Network (CNN) predictor and proposed LSTM predictor is same as in [2]. The predictors take n-1 values (in this case n = 8) from generator output and attempts to output an unsigned 16-bit integer value as close to the actual (last) value from the generated sequence (generator output).

3.3.1 LSTM Predictor: The LSTM predictor consists of LSTM layers with different modification of the LSTM Cell than the LSTM-based generator. The LSTM Cell in the LSTM Predictor is modified to have LReLU activation instead of the Tanh activation which is usually employed in the LSTM based networks.

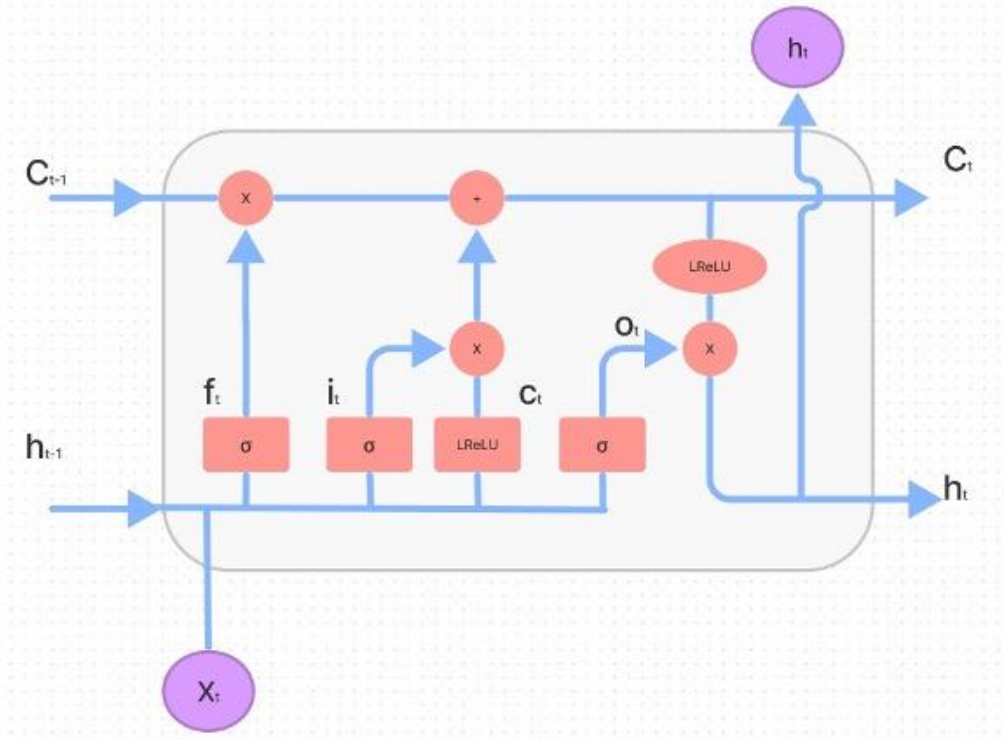

Fig 17: Modified LSTM Cell of the LSTM Predictor

The LSTM Predictor takes the n-1 values (here n = 8) from the generator output. These values then go through an LSTM layer of 8 units (8 identical LSTM Cells, one of which is shown in Fig 14) which produces 8 values, after which the values are reshaped and fed to the last LSTM layer with 1 unit. This layer outputs the predicted value during adversarial training.

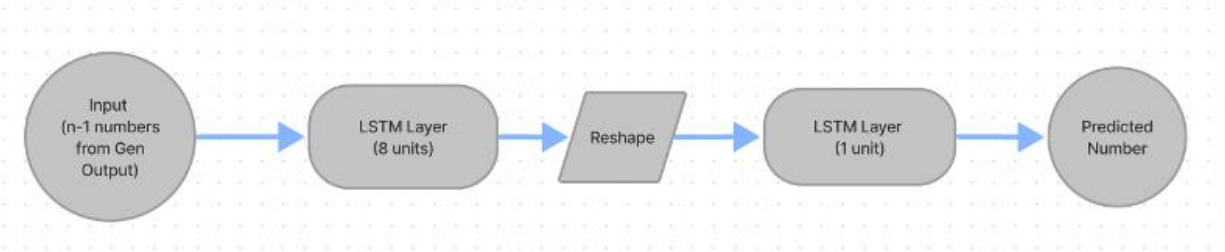

Fig 18: Proposed LSTM Predictor

3.3.2 Convolutional Neural Network Predictor: The Convolutional Neural Network (CNN) Predictor was introduced in [2]. It takes the output (n – 1 value, n = 8) from the previously discussed Pseudo Random Number Generators (both LSTM-based and FC Generators) as input. After going through several Convolutional and FC layers, it outputs a predicted output. The network consists of four 1-dimensional Convolutional layers, followed by a 1-dimensional Max Pool and finally two FC layers. Each convolution layer has 4 filters, kernel size = 2 and stride = 1. The FC layers are of two types one with 4 units and one with 1 unit. In all the layers except Max pooling layer, Leaky ReLU is used as activation.

3.4 LOSS FUNCTIONS AND OPTIMIZER

The loss function used during the training of the previously mentioned generators is absolute difference of the generator and predictor output [2]. As for the optimizer, the Adam optimizer is used for keeping track of gradients [2, 14] during the adversarial training of the generators.

4. EXPERIMENTS

The measure of how good the GAN/Adversarial training process help the generators is done by running NIST statistical test of randomness to test their output sequence. The generators are tested through the evaluation dataset (large dataset of 100,000,000 binary bits) once before training and then thrice after going through the training process.

4.1 PROCEDURE

The training procedure takes a lot of influence from [2] to keep things comparable during training of the generators, both proposed and baseline [2] (and its variations). The evaluation dataset is initialized, where input includes (seed, offset) vectors with seed being a fixed value in the entire dataset and offset value increasing by one as we go through these input vectors. The total number of elements/pairs in the dataset is determined by the following formula [2]:

$$Number\ of\ elements = total\_bits/(output\_size * num\_output\_bits)$$

Since there is a NIST test performed before training process, the generators are fed the evaluation dataset. To this the generators produce a corresponding output (output size = 8) sequence. These sequences are always in the range of uint16 (unsigned 16-bit integer) during the testing and can be changed based on the max value provided during the

initialization of the generators. The values generated are stored as ASCII value which is the same format used in [2] and a format supported by the NIST test suite software [3].

Once the training is done, the trained generator is fed the evaluation dataset and then NIST is run again. Now the generator is expected to improve the quality of the output sequence by increase in the pass percentage of the NIST tests when compared to before adversarial training.

4.2 TRAINING PARAMETERS

The PRNGs are trained like the Generative Adversarial Networks (GAN). So anytime 'GANs' show up in the rest of the paper, it is the PRNGs generators combined with the predictors to create this training model (see Section 3.1). The GANs are trained for 10,000 epochs with a batch size of 1, where the predictor gradient is updated 3 times and generator gradient is updated 1 time. The learning rate for the training process for all generators is 0.01. Output from the generator is a sequence of unsigned 16-bit integers ranging from 0 – 65535. This is obtained from passing the single precision (32-bit) floating-point values through the modulo activation in the last FC layer in all the generators like the generator in [2]. Evaluation dataset contains total of 100,000,000 bits (as an ASCII text file of 0s and 1s) per experiment for each type of the generator variations, totaling at about 781,250 input samples. The longer sequences and larger batches batch sizes were not considered for the experiments due to Google Colab's limited computing resources availability in a session.

4.3 NIST TESTING METHOD

The NIST statistical test suite is used to measure output sequence quality from the generators. Default settings for block lengths for the following tests:

- Block Frequency (Block length) = 128
- Nonoverlapping Template Test (Block Length) = 9
- Overlapping Template Test (Block Length) = 9
- Approximate Entropy Test (Block Length) = 10
- Serial Test (Block Length) = 16
- Linear Complexity Test (Block Length) = 500

Moreover, the test suite also takes in 10 bitstream inputs, indicating that each sequence is 1 bit stream of 1,000,000 bits and hence there are 10 sequences tested within a total of 100,000,000 total input bits to the NIST test. As seen from the below screenshot (Fig. 7) of the final analysis of all the individual test results of one experiment done on the trained generators. The result includes, the p-value of each test, number of bitstreams (output sequence) out of the total passing the tests and name of test type. The Asterisk (*) in the test result means that the p-values is below critic value = 0.01 (default) and hence a test failure. The overall test fails if the number of individual test success is less than the minimum required number of passing tests (varies and is calculated by the test suite) or the p-value of each individual test run is less than the critical value.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <data/bin_lstm_gen_lstm_pred_output.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  0   0   0   0   2   0   4   2   1   1  0.066882     10/10      Frequency
  2   0   0   3   1   1   0   0   1   2  0.350485      9/10      BlockFrequency
  0   0   0   1   0   1   3   3   0   2  0.122325     10/10      CumulativeSums
  0   0   0   0   3   1   1   1   1   3  0.213309     10/10      CumulativeSums
  0   0   0   2   1   0   1   2   2   2  0.534146     10/10      Runs
  2   1   2   1   0   1   1   1   1   0  0.911413     10/10      LongestRun
  4   0   1   1   1   1   2   0   0   0  0.122325      7/10   *  Rank
 10   0   0   0   0   0   0   0   0   0  0.000000 *    2/10   *  FFT
  1   1   1   1   1   0   0   0   1   4  0.213309     10/10      NonOverlappingTemplate
  0   0   2   2   0   0   0   3   1   2  0.213309     10/10      NonOverlappingTemplate
  0   2   0   3   0   0   2   1   2   0  0.213309     10/10      NonOverlappingTemplate
  0   0   0   0   1   0   3   0   2   4  0.017912     10/10      NonOverlappingTemplate
  1   1   1   2   2   0   0   1   1   1  0.911413     10/10      NonOverlappingTemplate
  2   1   0   1   1   2   2   0   0   1  0.739918     10/10      NonOverlappingTemplate
  0   2   0   2   0   1   0   2   2   1  0.534146     10/10      NonOverlappingTemplate
  0   0   0   3   2   1   0   0   2   2  0.213309     10/10      NonOverlappingTemplate
  0   1   3   0   3   0   0   1   0   2  0.122325     10/10      NonOverlappingTemplate
  0   1   1   0   4   1   0   0   1   2  0.122325     10/10      NonOverlappingTemplate
  0   3   1   0   1   1   0   2   1   1  0.534146     10/10      NonOverlappingTemplate
  0   0   0   2   1   1   3   0   1   2  0.350485     10/10      NonOverlappingTemplate
  0   0   2   0   0   0   2   3   2   1  0.213309     10/10      NonOverlappingTemplate
  0   0   1   0   1   0   2   3   1   2  0.350485     10/10      NonOverlappingTemplate
  1   0   1   0   0   0   2   2   0   4  0.066882     10/10      NonOverlappingTemplate
  1   1   0   0   3   0   0   2   2   1  0.350485     10/10      NonOverlappingTemplate
  1   1   1   1   1   2   1   0   1   1  0.991468     10/10      NonOverlappingTemplate
  1   0   1   1   0   2   1   2   0   2  0.739918     10/10      NonOverlappingTemplate
  0   0   1   1   1   2   1   1   1   2  0.911413     10/10      NonOverlappingTemplate
  0   0   1   1   2   3   0   1   1   1  0.534146     10/10      NonOverlappingTemplate
  1   0   0   0   3   1   1   1   2   1  0.534146     10/10      NonOverlappingTemplate
  0   0   1   1   2   0   0   2   3   1  0.350485     10/10      NonOverlappingTemplate
  0   0   1   1   3   0   1   2   2   0  0.350485     10/10      NonOverlappingTemplate
  0   1   1   2   1   0   1   3   0   1  0.534146     10/10      NonOverlappingTemplate
  1   2   0   0   1   0   1   1   4   0  0.122325     10/10      NonOverlappingTemplate
  0   0   1   2   1   1   1   0   2   2  0.739918     10/10      NonOverlappingTemplate
```

Fig 19: Example NIST final analysis output for a trained generator

5. RESULTS AND CONCLUSION

5.1 RESULTS

The below tables depict the average pass percentage of the output sequence generated by both the proposed LSTM-based and FC generators (baseline from [2] + variations) with varying number of Fully Connected Neural Network layers. Table 1 shows the NIST performance of untrained LSTM generators, where none of the tests passed. Tables 2 and 3 show the NIST performance of the LSTM-based generators with both proposed LSTM Predictor and CNN Predictor from [2]. On the other hand, Table 4 depicts the NIST results for untrained FC generator (baseline from [2] + variations – all tests fail) the Tables 5 and 6 show the performance of FC generators (baseline from [2] + variations) when trained with both types of Predictors.

LSTM Generator:

LSTM + 1 FC layer	LSTM + 2 FC layers	LSTM + 3 FC layers	LSTM + 4 FC layers
0.0%	0.0%	0.0%	0.0%

Table 3. NIST test result (overall tests passing) for untrained LSTM generators

Predictor	LSTM + 1 FC layer	LSTM + 2 FC layers	LSTM + 3 FC layers	LSTM + 4 FC layers
LSTM Pred	57.5%	**96.76%**	**98.7%**	97.13%

Table 4. Average NIST result (out of 3 experiments) LSTM-based Gen + LSTM Pred

Predictor	LSTM + 1 FC layer	LSTM + 2 FC layers	LSTM + 3 FC layers	LSTM + 4 FC layers
CNN Pred	**58.06%**	78.3%	88.53%	**97.8%**

Table 5. Average NIST result (out of 3 experiments) LSTM-based Gen + CNN Pred

FC Generator:

FC Gen 2 layers	FC Gen 3 layers	FC Gen 4 layers	FC Gen 5 layers	FC Gen 6 layers	FC Gen 7 layers	FC Gen 8 layers
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 6. NIST test result (overall tests passing) for untrained FC generators

Predictor	FC Gen 2 layers	FC Gen 3 layers	FC Gen 4 layers	FC Gen 5 layers	FC Gen 6 layers	FC Gen 7 layers	FC Gen 8 layers
LSTM Pred	**64.5%**	**96.08%**	**97.8%**	94.2%	**97.4%**	94.6%	**97.4%**

Table 7. Average NIST result (out of 3 experiments) FC Gen + LSTM Pred

Predictor	FC Gen 2 layers	FC Gen 3 layers	FC Gen 4 layers	FC Gen 5 layers	FC Gen 6 layers	FC Gen 7 layers	FC Gen 8 layers
CNN Pred	0.52%	65.2%	97.6%	**97.6%**	96.1%	**95.7%**	97.2%

Table 8. Average NIST result (out of 3 experiments) FC Gen + CNN Pred

5.2 EVALUATION OF THE RESULTS

The generators fail all the NIST test cases before training, indicating that just increasing or changing the number of layers and initial weights in the FC layers of the generator is very much predictable, hance the test failures. Once the training procedure was done with generators, the pass rate in most cases was much higher, a lot of times touching the 94%-98% pass rates with so less number of epochs when compared to the massive 200,000 epochs with a mini-batch size of 2048 in [2]. These experiments show a better or an on

par performace when compared to the baseline. This however is pretty close to the work [23] where the pass rate was 98% in best case.

The generator and predictor loss plots are very chaotic and do not converge elegantly as they did during GAN training in the baseline [2]. The Fig. 18 below shows an example loss plot of Experiment 1 with LSTM-based generator- LSTM Predictor adversaries, possibly due the objective of both generator and predictor to stay away from one another as much as possible. Both generator loss (orange plot) and predictor loss (blue plot) are overlapping except around 100 to 200 epochs (steps).

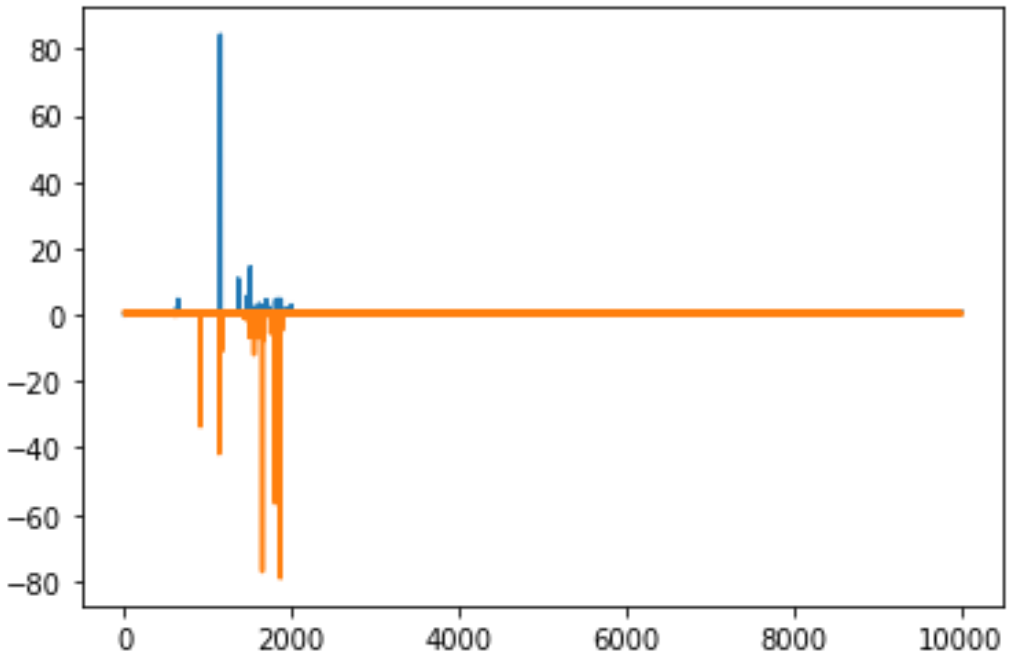

Fig 20: Loss plot during Experiment 1 for LSTM Gen 4 FC Layers - LSTM Pred

5.3 CONCLUSION

This thesis proposed a series of generators: LSTM-based and Fully Connected Neural Network (FCNN Gen) baseline [2] variations (including baseline [2] for experiments). These generators were then experimented with the proposed LSTM Predictor (LSTM Pred) and CNN Predictor (baseline predictor from [2]) to determine how the use of LSTM in adversarial training (Generative Adversarial Network training) affects NIST test results. The experiments were also done to find how different number of layers (neural network complexity) affect the NIST pass rate.

With the experimental setup, NIST performance is greater and more consistent when the generators were paired with LSTM Predictors compared to utilizing the Convolutional Neural Network Predictor from [2]. Out of the four pairs of generator-predictor in adversarial training, the FC Generator – LSTM Predictor gave the best NIST perform which was also very consistent. This was followed by the LSTM-based Generator – LSTM Predictor, then by FC Generator - CNN Predictor and finally the LSTM-based Generator – CNN Predictor giving the worst NIST performance. Hence the LSTMs can be very useful in creating Neural Network Based Generators (PRNGs) as well as the Predictors to help better train the generators.

Next, we look at relation of increasing network complexity (number of NN layers) with respect to the NIST randomness test performance. The increase in number of layers in the generators did not have an expected increasing trend, rather the NIST results almost plateaued after 4 FC layers for FC Generator (variation of the baseline [2]) and after LSTM layer + 3 FC layers configuration for LSTM-based Generator (proposed). So,

it can be summarized that just by adding a greater number of layers to the FC NN PRNG or LSTM-based PRNG does not guarantee a huge increase in the NIST performance, provided the greater increase in training time.

6. LIMITATIONS AND FUTURE WORK

This experiment was done to determine how the NIST test results are affected by different number of neural network layers in a series of proposed (LSTM-based) generators and the baseline [2] (FC NN) generator variations. The experiment also showed how the LSTM Predictor resulted in improved and a more consistent result. All this, however, is not without shortcomings. Something that was not explored is how different complexity Predictor (different number of layers) and different loss functions could affect the adversarial training of the generators, in the end potentially affecting NIST test suite in a whole different way.

Another aspect that was missing which could be explored is how increasing the batch size from a mere 1 (due to limited resources on Google Colab) to something larger like 2048 in the adversarial as done in [2] could affect the NIST test results, with the same configuration of the generators. Also, to make the PRNGs developed in this paper to be more secure, one can also look as to how to secure the neural network architecture from any manipulation from any external sources.

LITERATURE CITED

[1] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." Advances in neural information processing systems 27 (2014).

[2] Bernardi, Marcello De, M. H. R. Khouzani, and Pasquale Malacaria. "Pseudo-random number generation using generative adversarial networks." Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, Cham, 2018.

[3] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. “2010. SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical Report.” National Institute of Standards & Technology, Gaithersburg, MD, USA.

[4] Kim, Hyunji, et al. "Generative Adversarial Networks-Based Pseudo-Random Number Generator for Embedded Processors." International Conference on Information Security and Cryptology. Springer, Cham, 2020.

[5] Pasqualini, Luca, and Maurizio Parton. "Pseudo random number generation through reinforcement learning and recurrent neural networks." Algorithms 13.11 (2020): 307.

[6] Benjamin Lindemann, Timo Müller, Hannes Vietz, Nasser Jazdi, Michael Weyrich, “A survey on long short-term memory networks for time series prediction”, *Procedia CIRP*,

Volume 99, 2021,Pages 650-655, ISSN 2212-8271, https://doi.org/10.1016/j.procir.2021.03.088.

[7] Jeong, Young-Seob, et al. "Pseudo random number generation using LSTMs and irrational numbers." 2018 IEEE international conference on big data and smart computing (BigComp). IEEE, 2018.

[8] Desai, V., et al. "Pseudo random number generator using time delay neural network." World 2.10 (2012): 165-169.

[9] Peter Kietzmann, Thomas C. Schmidt, and Matthias Wählisch. “A Guideline on Pseudorandom Number Generation (PRNG) in the IoT”. ACM Comput. Surv. 54, 6, Article 112 (July 2022), 38 pages. DOI: https://doi.org/10.1145/3453159

[10] Pham, T.D. Time–frequency time–space LSTM for robust classification of physiological signals. Sci Rep 11, 6936 (2021). https://doi.org/10.1038/s41598-021-86432-7

[11] Fort, Travis, "Controlling Randomness: Using Procedural Generation to Influence Player Uncertainty in Video Games" (2015). HIM 1990-2015. 1707. https://stars.library.ucf.edu/honorstheses1990-2015/1707

[12] Kelsey J., Schneier B., Wagner D., Hall C. (1998) “Cryptanalytic Attacks on Pseudorandom Number Generators.” In: Vaudenay S. (eds) Fast Software Encryption. FSE 1998. Lecture Notes in Computer Science, vol 1372. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-69710-1_12

[13] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. 2007. Cryptanalysis of the windows random number generator. In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). Association for Computing Machinery, New York, NY, USA, 476–485. DOI: https://doi.org/10.1145/1315245.1315304

[14] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[15] Bhattacharjee, Kamalika, Krishnendu Maity, and Sukanta Das. "A search for good pseudo-random number generators: Survey and empirical studies." arXiv preprint arXiv:1811.04035 (2018).

[16] Olgun, Ataberk, et al. "QUAC-TRNG: High-throughput true random number generation using quadruple row activation in commodity DRAM chips." 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021.

[17] Fei Yu, Lixiang Li, Qiang Tang, Shuo Cai, Yun Song, Quan Xu, "A Survey on True Random Number Generators Based on Chaos", Discrete Dynamics in Nature and Society, vol. 2019, Article ID 2545123, 10 pages, 2019. https://doi.org/10.1155/2019/2545123

[18] Gomathisankaran, Mahadevan & Lee, Ruby Bei-Loh. Tantra: A fast PRNG algorithm and its implementation, paper, June 2009; (https://digital.library.unt.edu/ark:/67531/metadc94290/: accessed February 27, 2022), University of North Texas Libraries, UNT Digital Library, https://digital.library.unt.edu ; crediting UNT College of Engineering.

[19] Pereira de Araujo R., Souto V.T. (2017) Game Worlds and Creativity: The Challenges of Procedural Content Generation. In: Marcus A., Wang W. (eds) Design, User Experience, and Usability: Designing Pleasurable Experiences. DUXU 2017. Lecture Notes in Computer Science, vol 10289. Springer, Cham. https://doi.org/10.1007/978-3-319-58637-3_35

[20] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33, 4, Article 22 (August 2007), 40 pages. DOI: https://doi.org/10.1145/1268776.1268777

[21] Hongyan Zang, Yue Yuan, Xinyuan Wei, "Research on Pseudorandom Number Generator Based on Several New Types of Piecewise Chaotic Maps", Mathematical Problems in Engineering, vol. 2021, Article ID 1375346, 12 pages, 2021. https://doi.org/10.1155/2021/1375346

[22] Arulkumaran, Kai & Deisenroth, Marc & Brundage, Miles & Bharath, Anil. (2017). A Brief Survey of Deep Reinforcement Learning. IEEE Signal Processing Magazine. 34. 10.1109/MSP.2017.2743240.

[23] Tirdad, K., Sadeghian, A.: Hopfield neural networks as pseudo random number generators. In: Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American. pp. 1–6. IEEE (2010)

[24] Rosenblatt, Frank. “The perceptron: a probabilistic model for information storage and organization in the brain.” Psychological review 65 6 (1958): 386-408.

[25] Gurney, Kevin (1997). An Introduction to Neural Networks (1st ed.). UCL Press.

[26] Li, Zewen, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. "A survey of convolutional neural networks: analysis, applications, and prospects." IEEE Transactions on Neural Networks and Learning Systems (2021).

[27] LeCun, Yann, Patrick Haffner, Léon Bottou, and Yoshua Bengio. "Object recognition with gradient-based learning." In Shape, contour and grouping in computer vision, pp. 319-345. Springer, Berlin, Heidelberg, 1999.

[28] Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

[29] Szandała, Tomasz. "Review and comparison of commonly used activation functions for deep neural networks." In Bio-inspired neurocomputing, pp. 203-224. Springer, Singapore, 2021.

[30] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures," in IEEE Signal Processing Magazine, vol. 26, no. 1, pp. 98-117, Jan. 2009, doi: 10.1109/MSP.2008.930649.

[31] Feng, Jie & Feng, Xueliang & Chen, Jiantong & Cao, Xianghai & Zhang, Xiangrong & Jiao, Licheng & Yu, Tao. (2020). Generative Adversarial Networks Based on Collaborative Learning and Attention Mechanism for Hyperspectral Image Classification. Remote Sensing. 12. 1149. 10.3390/rs12071149.