Predicting Pseudo Random Values Using Convolutional Neural Networks

1st Spencer Arnold
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

2nd Najib Ali
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

3rd Jacob Anderson
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

4th Matthew Hawks
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

5th Ryan Hines
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

6th Tae Kweon
Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee
email address

***Abstract*—Sample Abstract: OBJECTIVE: The role of antibiotic therapy in managing acute bacterial sinusitis (ABS) in children is controversial. The purpose of this study was to determine the effectiveness of high-dose amoxicillin/potassium clavulanate in the treatment of children diagnosed with ABS.**

METHODS: This was a randomized, double-blind, placebo-controlled study. Children 1 to 10 years of age with a clinical presentation compatible with ABS were eligible for participation. Patients were stratified according to age (¡6 or 6 years) and clinical severity and randomly assigned to receive either amoxicillin (90 mg/kg) with potassium clavulanate (6.4 mg/kg) or placebo. A symptom survey was performed on days 0, 1, 2, 3, 5, 7, 10, 20, and 30. Patients were examined on day 14. Childrens conditions were rated as cured, improved, or failed according to scoring rules.

RESULTS: Two thousand one hundred thirty-five children with respiratory complaints were screened for enrollment; 139 (6.5

CONCLUSIONS: ABS is a common complication of viral upper respiratory infections. Amoxicillin/potassium clavulanate results in significantly more cures and fewer failures than placebo, according to parental report of time to resolution. (9)

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Observation:

Society relies on Pseudo-random numbers for a multitude of reasons. Whether the usecases fall in Information Security or, more broadly, modeling and simulation, we think it is important to not only push the boundary for the modern pseudo random number directly, but to also analyze the history of pseudo randomness to aide in that effort.

Whether true randomness is a inhibition of the human perception or not, there is clear need to push the modern pseudo-random number closer to converging on perceived ”true” randomness.

Questions/Aim:

To train a neural net to predict PRNs from a chosen PRNGs output. We aim to train neural nets to predict values of assigned PRNGs and track attributes of each to make generalized conclusions about the development of PRNGs with respect to time and any PRN correlations we uncover.

Hypotheses:

1) We predict there will be a positive trend over time on the cryptographic strength of each subsequent PRNG, given the nature of the increasing importance of stronger PRNGs.

2) We also predict that as we get into cryptographically stronger generation methods, our prediction success rates (even with learning) will be less effective.

3) We expect to uncover correlations in Pseudo-random numbers based on each individual generator, and aim to extract more generalized correlations between generators themselves.

II. Background

III. Methods

A. Seeding Method

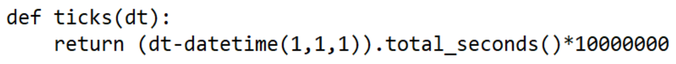
We went with a seed generation method that allowed a way to introduce some minor level of entropy to avoid letting the neural network aimlessly swim through the entropy of a strong seed instead of gaining stochastic insight on the data from the PRNG.

The seed generation method we chose derives from the concept of using the system time as an element for seed generation. The specific implementation we chose took inspiration from Microsoft’s .NET system.datetime.ticks property. [1] We chose to single out this method due to its documentation and unique simplicity. System time is widely used as a parameter for modern seed generation methods.

To put it more trivially: ”a pseudo-random number generator is a deterministic algorithm that, given an initial number (called a seed), generates a sequence of numbers that adequately satisfy statistical randomness tests. Since the algorithm is deterministic, the algorithm will always generate the exact same sequence of numbers if it's initialized with the same seed. That's why system time (something that changes all the time) is usually used as the seed for random number generators." [2]

According to the Microsoft documentation, "A single tick represents one hundred nanoseconds or one ten-millionth of a second. There are 10,000 ticks in a millisecond, or 10 million ticks in a second. The value of this property represents the number of 100-nanosecond intervals that have elapsed since 12:00:00 midnight, January 1, 0001 in the Gregorian calendar." [1]

We used a fairly similar Python port, as seen in Figure 1



```
def ticks(dt):
    return (dt-datetime(1,1,1)).total_seconds()*10000000
```

Fig. 1. Default Seeding Method

This python port of Microsoft's tick method can be attributed to "mhawke" on StackOverflow. [3] The author had some noteworthy comments about this implementation, namely some porting side effects:

1) UTC times are assumed.
2) The resolution of the DateTime object is given by DateTime.resolution, which is DateTime.timedelta(0, 0, 1) or microsecond resolution (1e-06 seconds). CSharp Ticks are purported to be 1e-07 seconds.

For experimental needs, we made additional changes to the implementation:

1) Changing start time from January 1, 0001 to January 1, 1970, which effectively reduced the length of the seed for experimental purposes.
2) Slicing the last 6 digits of the ticks result to acquire more digit variation for frequent invocation.

The final modified method allows enough spread between frequently retrieved ticks, where we are assuming reasonable pseudo-unpredictability. This serves as a simplistic but constantly changing control mechanism for being able to seed PRNGs and test experimental outcomes. While not the most cryptographically strong, we needed a way to have some controlled aspect of seed generation to feed into generators of varying cryptographic complexity (to have some baseline of comparison).

The original idea was to feed each PRNG different seeds from the same seed generator; however, many PRNG algorithms impose strict seed requirements to pass tests of randomness. Out of the five PRNG methods we implemented, Lagged Fibonacci was the only one that had special seed requirements, so we created a separate seed generator based on the same fundamental ticks generation method, but modified to meet the restrictions. Other PRNGs not implemented in this research that impose seed restrictions include Wichmann-Hill (which accepts three different seeds) and Maximally Periodic Reciprocals (which requires a Sophie Prime), among others.

You might ask: won't different seed generators introduce flaws or bias in the experiment? Well, it depends on what you are testing. In our case, we are strictly testing the "complexity" of the generator itself, so supplying a seed that is not blatantly predictable but also not unpredictable was sufficient. Our goal was to allow the characteristics of the generator to be exposed, for we were cracking the "complexity" of the generation algorithm, not the complexity of an arbitrary seed.

B. PRNG Implementations

For the experiment, we chose five PRNG methods by year of invention. We attempted to choose PRNG methods that displayed popularity in the world of pseudo random number generation while making sure the year of invention was reasonably distributed among the group. Choosing five PRNGs allowed us to focus on implementations, while leaving future research opportunities open for working with other PRNGs that we did not cover.

Chosen PRNGs:

- Middle-square method (1946)
- Linear congruential generator (1958)
- Lagged Fibonacci (1965)
- Park-Miller (1988)
- Mersenne Twister (1998)

The call definition of any given PRNG function is as follows:

```
PRNGfunc(seed, n)
```

This is so we can experimentally automate the calling of each PRNG without getting too complex. The given PRNG function should return a list of n generated numbers using the generation method. All other parameters outside of the seed and n are default values.

For example, if the call was PRNG(seed, 10), it might return something like [3,5,10,1,31,17,2,4,6,7]

We control parsing the n-length list and handling seeds externally. This plays logically with separation of concerns for our usecase.

While python generators can be useful iterating over previously generated iterables, we did not want to clutter our experimental execution code, so we stuck to a classical internal handling of all iterations.

Below are short descriptions and non-inclusive general notes of each PRNG and any implementation notes made during the developement process.

Middle Square:

- "To generate a sequence of n-digit pseudorandom numbers, an n-digit starting value is created and squared, producing a 2n-digit number. If the result has fewer than 2n digits, leading zeroes are added to compensate. The middle n digits of the result would be the next number in the sequence, and returned as the result. This process is then repeated to generate more numbers." [4]

Notes:

- Generally the value of the seed has to be even, but can be padded with leading zeros.
- If the middle n digits are all zeroes, the generator then outputs zeroes indefinitely. If the first half of a number

in the sequence is zeroes, the subsequent numbers eventually converges to zero.

Linear Congruential:

- A linear congruential generator is a PRNG that represents an ”additive congruential method”, with foundations in improving upon ”unsatisfactory” tests with entropy in fibonacci sequences. [5]

Notes:

- The generator is not sensitive to the choice of c, as long as it is relatively prime to the modulus (e.g. if m is a power of 2, then c must be odd), so the value c=1 is commonly chosen.
- If c = 0, the generator is often called a multiplicative congruential generator (MCG), or Lehmer RNG (which is used for our implementation). If c 0, the method is called a mixed congruential generator.
- Parameters were chosen based on $2^3 2$ numbers in table 2 of the article ”Tables of linear congruential generators of different sizes and good lattice structure.” [6]

Lagged Fibonacci:

- The Lagged Fibonacci PRNG is a generalization of the Fibonacci Sequence [7], where the sequence is generated based off a seed and the sum of the last two values is the PRN (which is also serves as the next seed).

Notes:

- It is refered to a ”lagged” generator, because ”j” and ”k” lag behind the generated pseudorandom value.
- Our implementation is called a ”two-tap” generator, in that you are using 2 values in the sequence to generate the pseudorandom number. However, note that a two-tap generator has some problems with randomness tests, such as the Birthday Spacings. Creating a ”three-tap” generator could addresses this problem.

Park Miller:

- Park Miller (also known as Lehmer) can be viewed as a particular case of the Lemher PRNG, which is a particular case of the linear congruential PRNG, where c=0 and particular parameters are specified.

Notes:

- ”In 1988, Park and Miller [8], suggested a Lehmer RNG with particular parameters m = 231 1 = 2,147,483,647 (a Mersenne prime M31) and a = 75 = 16,807 (a primitive root modulo M31), now known as MINSTD.” [9]

Mersenne Twister:

- ”The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field” [10]

Notes:

- This PRNG is similar to a common LFSR. Its MT19937 implementation is probably the most commonly used modern PRNG.
- It is also the default generator in the Python language starting from version 2.3.
- For our implementation, we used numpy’s version of the Mersenne Twister.

C. Experimental Setup

The following is a vastly informal mathematical representation of our experimental model, using a loose coupling of TLA+ notation and some set-builder theory. Further explanations and graphics will follow to aide in interpretation of the design of the experimental setup.

Let n be any arbitrary natural number such that $\{n \in \mathbb{N}\}$. Let E represent the experiment definition. Let $seed$ be a function such that when called will return a seed. Given k and S_n, let $prng$ be a function such that when called will return a set of pseudo random numbers, dervied from an initial seed S_n and a given algorithm, where the length of the set is k.

$$\begin{aligned}
\mathbb{N} = \{0, ..., n\} \\
E(\mathbb{N}, k, networkparams) \triangleq [n \in \mathbb{N} \mapsto \\
S_n = seed[] \\
prng[S_n, k] \mapsto \mathbb{L}_n where \\
\{i \in \mathbb{L}_n | 0 \leq i \leq k\} \\
\mathbb{X}_n \triangleq \{n \in \mathbb{N} | 0 \leq n - 1\} \\
\mathbb{Y}_n \triangleq \{n \in \mathbb{N} | n \neq \mathbb{X}_n\} \\
P(\mathbb{X}, \mathbb{Y}, networkparams) : \\
\mathbb{X} \wedge networkparams \rightarrow \mathbb{B} \simeq \mathbb{Y}]
\end{aligned}$$

Fig. 2. Notated Experiment Model

Given an enumerated set, $\mathbb{N}$, where $prng$ is the chosen PRNG, S_n is nth seed in the iteration, and k is the length of the desired output vector ($\mathbb{L}_n$) of $prng$, $\mathbb{X}_n$ represents an enumerated set containing n sets of $k-1$ values generated from a PRNG ($prng$) and each $\mathbb{X}_n$ set is dervied from a different seed. $\mathbb{Y}_n$ represents a set containing n sets of kth values with a direct mapping to each $\mathbb{X}_n$ such that $\mathbb{X}_n \mapsto \mathbb{Y}_n$. P a predictor function that represents a convolutional neural network, which takes in $\mathbb{X}_n$ and $\mathbb{Y}_n$, yields a new set $\mathbb{B}_n$ implied from $\mathbb{X}_n$, where the model trains $\mathbb{B}_n$ to be similar or equal to $\mathbb{Y}_n$ based on back-propagation due to previous predictions, thus using supervised learning to build a regression model.

For a simplified graphical representation of the latter description, please reference the predictive model in Figure 4, the simplified experimental model in Figure 3 , and the granular view of the experimental model in Figure 5.

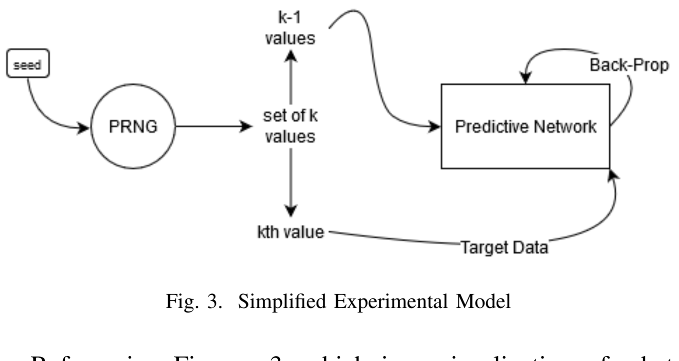


Fig. 3. Simplified Experimental Model

Referencing Figure 3, which is a visualization of what a 1-dimensional architecture of our experiment might entail, the predictive neural net as visualized in Figure 4 is fed outputs of a specific PRNG, which will attempt to predict an kth value, based on previous k-1 values in a particular set of input data, where each set is generated based off of a single unique seed. After being trained on, ideally thousands of sets, the predictive network will form a better stochastic ”understanding” of how the PRNG works underneath, thus being able to more accurately predict numbers generated from that PRNG in the future (i.e.,a supervised regression model). The backpropagation will flow through the predictive network to acheive the adversarial nature of a traditional GAN setup, but in reality, we won't be using a generative network, but rather a PRNG algorithm, so the generative part of our setup won't be defensive.

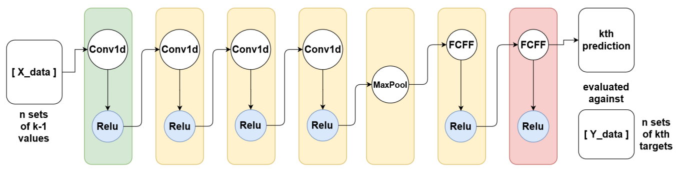


Fig. 4. Predictive Model

Referencing Figure 4, which is a visualization of the layer stack we used for the predictive network in each trial of the experiment, it ”consists of four stacked convolutional layers, each with 4 filters, kernel size 2, and stride 1, followed by a max pooling layer and two FCFF layers with 4 and 1 units, respectively. The stack of convolutional layers allow the network to discover complex patterns in the input.” [11] Explain here how/why we used that research's dicriminator model in our experiment exactly, except we implemented Relus instead of leaky-relus. Talk about the input xdata and ydata shape and reference the granular model Figure 5 for detail on how these data are aggregated.

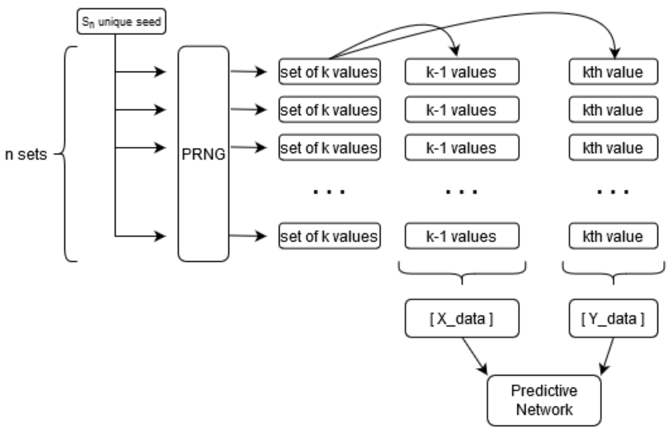


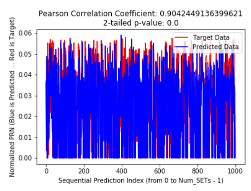
Fig. 5. Granular Experimental Model

Talk generally about the data aggregation process, just walk through it.

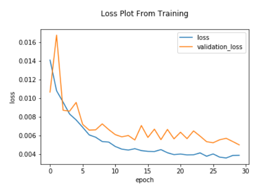
D. Experimental Execution

Highlight configuration parameters used to execute the experiment. Make a note the reasoning behind our small num of epochs and general reasoning behind all parameters.

IV. Results

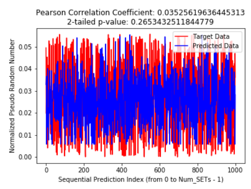


(a) Testing Regression Plot

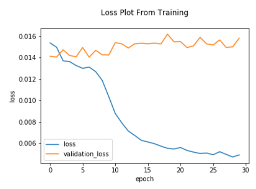


(b) Training Loss Plot

Fig. 6. Middle Square Results

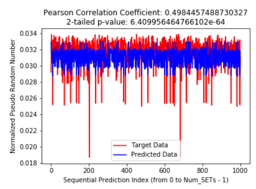


(a) Testing Regression Plot

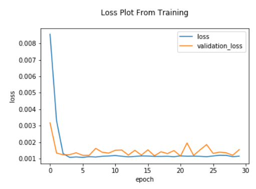


(b) Training Loss Plot

Fig. 7. Linear Congruential Results

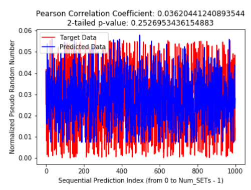


(a) Testing Regression Plot

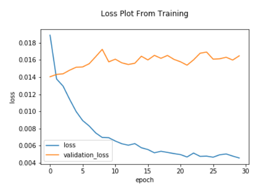


(b) Training Loss Plot

Fig. 8. Lagged Fibonacci Results

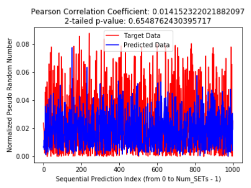


(a) Testing Regression Plot

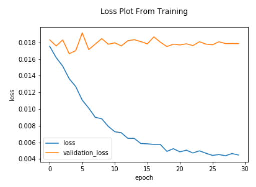


(b) Training Loss Plot

Fig. 9. Park Miller Results



(a) Testing Regression Plot



(b) Training Loss Plot

Fig. 10. Mersenne Twister Results

V. Discussion

Below are general notes for the discussion section....

Mistakes Made / how the research could be made more robust.

uncertainties

Practical Application: A GAN method similar to this could be used in the future during a real-time prediction attack scenario. If attackers are somehow able to break a strong PRNG that is used to generate numbers for cryptographic methods, a method like this could be used as a defense mechanism to prevent (or slow down) further PRN generation cycles from being compromised.

This provides a better framework for implementing new PRNGs, seed methods, and experimental designs... and fixing ours to be more mathematically accurate or more robust.

With the results we obtained by doing mere 20 epochs, a lot heavier training and testing could be implemented with powerful computation power.

The base code, such as seeding and PRNG code can also be significantly improved upon to be more mathematically accurate and produce better training and testing of models...

.. code:: ipython3

References

[1] Microsoft. Datetime.ticks property. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.datetime.ticks?view=netframework-4.8

[2] M. Afshari. Stackoverflow comment. [Online]. Available: https://stackoverflow.com/questions/1785744/how-do-i-seed-a-random-class-to-avoid-getting-duplicate-random-values#comment2290983_1785752

[3] mhawke. Stackoverflow answer. [Online]. Available: https://stackoverflow.com/a/29368771

[4] Wikipedia contributors, "Middle-square method — Wikipedia, the free encyclopedia," 2020, [Online; accessed 24-April-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Middle-square_method&oldid=951418817

[5] A. Rotenberg, "A new pseudo-random number generator," *J. ACM*, vol. 7, no. 1, p. 7577, Jan. 1960. [Online]. Available: https://doi.org/10.1145/321008.321019

[6] P. L'Ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Math. Comput.*, vol. 68, pp. 249–260, 1999.

[7] É. Lucas, *Le calcul des nombres entiers. Le calcul des nombres rationnels. La divisibilité arithmétique*, ser. Théorie des nombres. Gauthier-Villars, 1891. [Online]. Available: https://books.google.com/books?id=_hsPAAAAIAAJ

[8] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find," *Commun. ACM*, vol. 31, no. 10, p. 11921201, Oct. 1988. [Online]. Available: https://doi.org/10.1145/63039.63042

[9] Wikipedia contributors, "Lehmer random number generator — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Lehmer_random_number_generator&oldid=950526808, 2020, [Online; accessed 24-April-2020].

[10] M. Matsumoto and Y. Kurita, "Twisted gfsr generators," *ACM Trans. Model. Comput. Simul.*, vol. 2, no. 3, p. 179194, Jul. 1992. [Online]. Available: https://doi.org/10.1145/146382.146383

[11] M. De Bernardi, M. Khouzani, and P. Malacaria, "Pseudo-random number generation using generative adversarial networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 191–200.