Predicting Pseudo Random Values Using Convolutional Neural Networks

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

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

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

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

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

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

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract. This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

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

I. Introduction

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

C. Experimental Setup

The mathematical description of our experiment can be outlined below.

$$\{0, G(S_n, k) \rightarrow \mathbb{L}_n \rightarrow Split(\mathbb{L}_n) \rightarrow \mathbb{X}_n, \mathbb{Y}_n, n\} \qquad (1)$$

Where the outer set builder notation iterates the following from 0 to n: Where G is the chosen PRNG, S_n is nth seed in the iteration, and k is the length of the desired output vector, $\mathbb{L}_n$, it is implied that $\mathbb{L}_n$ is the output vector of G and $\mathbb{X}_n$ is $\mathbb{L}_n$ with the last element removed. $\mathbb{Y}_n$ is used as the corresponding label for $\mathbb{X}_n$.

The predictor is a convolutional neural network implementing the function...

$$P(\mathbb{X}, \mathbb{Y}) : \mathbb{X} \rightarrow \mathbb{B} \simeq \mathbb{Y} \qquad (2)$$

Where $\mathbb{X}_n$ represents a set containing n sets of $k-1$ values generated from a PRNG where 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 $\mathbb{X}_n \rightarrow \mathbb{Y}_n$. P, the predictive neural network, 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 the back-propagation due to previous predictions.

For a simplified graphical representation of the latter description, please reference the predictive model in Figure 2, the simplified experimental model in Figure 3, and the granular view of the experimental model in Figure 4.

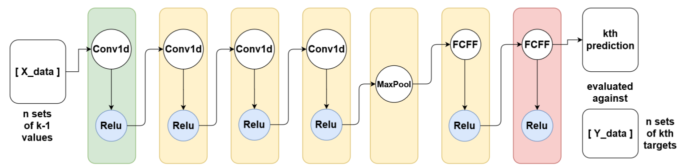


Fig. 2. Predictive Model

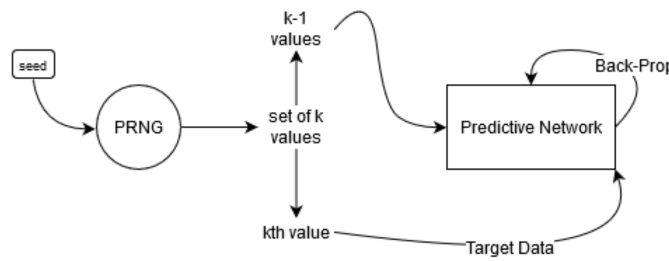


Fig. 3. Simplified Experimental Model

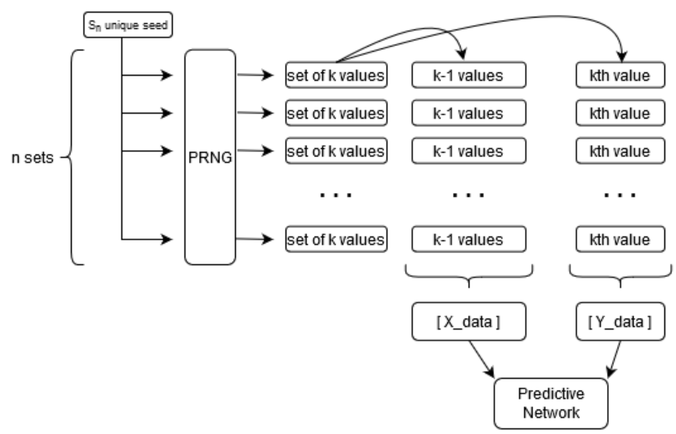
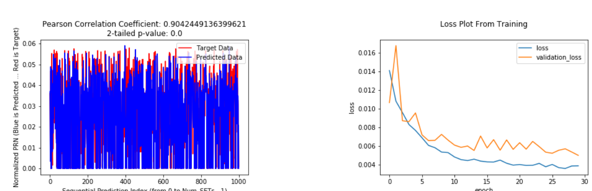


Fig. 4. Granular Experimental Model

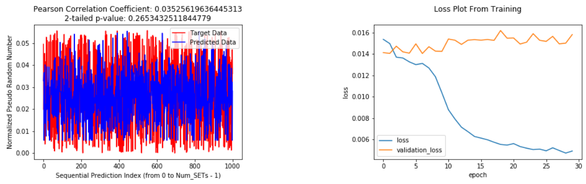
IV. Results



(a) Testing Regression Plot

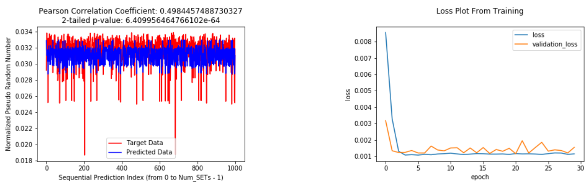
(b) Training Loss Plot

Fig. 5. Middle Square Results



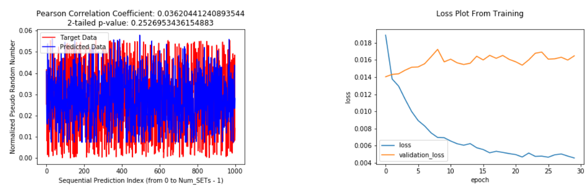
(a) Testing Regression Plot (b) Training Loss Plot

Fig. 6. Linear Congruential Results



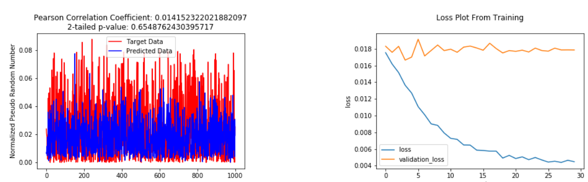
(a) Testing Regression Plot (b) Training Loss Plot

Fig. 7. Lagged Fibonacci Results



(a) Testing Regression Plot (b) Training Loss Plot

Fig. 8. Park Miller Results



(a) Testing Regression Plot (b) Training Loss Plot

Fig. 9. Mersenne Twister Results

V. Discussion

Start typing here her is Start typing here her is

.. 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