2023 IEEE International Conference on Machine Learning and Applied Network Technologies (ICMLANT) | 979-8-3503-0391-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICMLANT59547.2023.10372968

Time Series Prediction using Convolutional Neural Networks

1st Aishwarya Asesh
University of Pennsylvania
Salt Lake City, USA
a.asesh@gmail.com

2nd Meenal Dugar
University of Pennsylvania
Salt Lake City, USA
meenal.dugar@utah.edu

***Abstract*—Pseudo-Random number generators (PRNG) are fundamental in cryptography. The quality of generated pseudo-random numbers (PRN) often determines the "cryptographic strength" of a system. Historically, there have been numerous attempts to develop PRNGs that reduce the predictability and correlation of each successive value produced. This paper introduces a novel approach to predicting sequences of specific PRNGs by leveraging deep convolutional networks (CNN) in a regression-based supervised learning process. An experimental framework is provided, and by building upon existing prediction models, it is demonstrated that the cryptographic strength of subsequent PRNGs increases. Simultaneously, prediction success rates, even with advanced learning, remain relatively ineffective. This underscores the growing importance of robust PRNGs across various industries.**

***Index Terms*—pseudo random number (PRN), pseudo random number generator (PRNG), convolutional neural networks (CNN), neural cryptography.**

I. Introduction

Pseudo-random numbers play an integral role in various applications, from Information Security to modeling and simulation. Pushing the boundary for modern pseudo-random numbers is essential, as is understanding the history of pseudo randomness to enhance future developments. The concept of true randomness, whether a limitation of human perception or a tangible objective, underscores the imperative to refine modern pseudo-random numbers, moving them closer to the ideal of "true" randomness. This paper's primary objective is to train a predictive neural network to anticipate the values of different PRNGs. The research seeks to draw generalized conclusions about the evolution of PRNGs over time and uncover any inherent correlations in PRN sequences.

The objective was achieved by implementing five distinct PRNGs:

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

While these PRNGs are discussed in detail in the methods section, the primary rationale for selecting these five was their representation of pivotal milestones in the chronological development of stronger PRNGs. Despite focusing on these five, further research might explore other PRNGs.

The research was prefaced with the following hypotheses:

1) A positive trend will be observed over time concerning the cryptographic strength of each successive PRNG, reflecting the escalating significance of potent PRNGs.
2) As the research delves into cryptographically superior generation methods, it is anticipated that prediction success rates, even with enhanced learning, will diminish in efficacy.
3) Correlations within PRNs stemming from individual generators will be identified, with an attempt to discern broader correlations between the generators themselves.

II. Background

PRNGs predominantly are deterministic algorithms, which employ an input seed to generate PRNs. The predictability of these numbers can vary, contingent on statistical analyses. In this investigation, convolutional neural networks are utilized to anticipate the subsequent series of numbers spawned by a PRNG. Conventionally, convolutional neural nets (CNNs) address image classification challenges. Nonetheless, their foundational design, derived from the architecture of a multilayer perceptron, facilitates localized pattern recognition, proving invaluable for extracting insights from potentially noisy data, typical of signal processing scenarios. [1] As [2] articulates, "CNNs harness spatial locality by implementing a local connectivity pattern between neurons of adjacent layers. This architectural design ensures that the filters being learned resonate most with a spatially localized input pattern." Such a feature becomes advantageous in scenarios demanding sequential analysis, as in time series challenges. Contrary to the conventional perception of CNNs as solely tools for image classification, a deeper understanding showcases their versatility. By considering pixel data as enumerated sequences of figures, the potential applications of CNNs, such as formulating regression models for PRN sequences, become evident. Subsequent sections delve deeper into these discussions.

III. Methods

A. Seeding Method

A seed generation method was adopted that incorporated a modest degree of entropy. This choice aimed to prevent

the neural network from navigating through the extensive entropy of a potent seed, directing it instead towards obtaining stochastic insights from the PRNG data.

The selected seed generation method draws inspiration from the strategy of employing system time as a seed generation element. The specific methodology embraced was influenced by Microsoft's .NET system.datetime.ticks property. [3] This approach was singled out due to its comprehensive documentation and inherent simplicity. In a broader context, system time is frequently used as an input for contemporary seed generation techniques.

Simplifying, as described by [4]: "a pseudo-random number generator is a deterministic algorithm that, when given an initial seed, produces a series of numbers that pass statistical randomness tests. Since the process is deterministic, the exact sequence of numbers will invariably be replicated if initialized with the same seed. This predictability underscores why system time, a constantly changing parameter, often serves as the seed for random number generators."

Per Microsoft's documentation, "A tick represents one hundred nanoseconds, or one ten-millionth of a second. With 10,000 ticks in a millisecond, and 10 million ticks in a second, this property's value signifies the number of 100-nanosecond intervals elapsed since 12:00:00 midnight, January 1, 0001 in the Gregorian calendar." [5]

Certain aspects of this adaptation, in particular pointing out two side effects [6]:

1) The assumption of UTC times.
2) The resolution of the DateTime object, represented by DateTime.resolution, equals DateTime.timedelta(0, 0, 1) or a microsecond resolution (1e-06 seconds). In contrast, CSharp Ticks claim a resolution of 1e-07 seconds.

To cater to experimental requirements, the following modifications were incorporated:

1) The commencement time was adjusted from January 1, 0001 to January 1, 1970, leading to a truncation of the seed's length for the experiments.
2) The terminal six digits of the tick outcome were extracted to foster greater digit variation during frequent calls.

The adopted modified method offers sufficient dispersion among frequently accessed ticks, postulating a plausible pseudo-unpredictability. This approach provides a basic yet continuously evolving control system to seed PRNGs and evaluate experimental outcomes. Despite not being the epitome of cryptographic strength, there was a need for a controlled seed generation element, offering a foundation to feed into generators with diverse cryptographic intricacies, ensuring a foundational comparison benchmark.

Initially, the plan was to supply distinct seeds from a consistent seed generator to every PRNG. However, several PRNG algorithms dictate rigorous seed prerequisites to ensure the randomness tests are passed. Among the five PRNG techniques implemented, only the Lagged Fibonacci imposed specific seed demands. Consequently, a dedicated seed generator was devised, grounded on the foundational ticks generation method, but adapted to meet the stipulated conditions. Other PRNGs not incorporated in this study, which impose seed constraints, include the Wichmann-Hill (which necessitates three disparate seeds) and the Maximally Periodic Reciprocals (demanding a Sophie Prime), to name a few.

One might wonder: could varying seed generators inject defects or skewness into the experiment? This largely hinges on the research focus. In this context, the primary objective was to test the inherent "complexity" of the generator. Thus, supplying a seed that straddles the boundary between predictability and unpredictability sufficed. The aim was to unveil the generator's distinct attributes, emphasizing the "complexity" intrinsic to the generation algorithm rather than the intricacy of a random seed.

B. PRNG Implementations

As delineated in the introduction, the study incorporated five PRNG techniques based on their invention years. The selection of these five methods was strategic, ensuring thorough implementation analysis while preserving potential avenues for future exploration. This sub-section elucidates the pivotal roles these PRNGs undertook in the research, alongside detailed annotations regarding each realized PRNG adaptation.

The general call definition for any PRNG function is presented as:

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

Such a structure facilitates experimental automation of each PRNG invocation without introducing undue complexity. The stipulated PRNG function is expected to yield a list of n numbers, generated through the designated method. Excluding the seed and n, all other parameters are defaulted.

Illustratively, if the invocation was `PRNG(seed, 10)`, a potential outcome could be:

$$[3, 5, 10, 1, 31, 17, 2, 4, 6, 7]$$

In the context of time series prediction, the management of parsing the n-length list and seed handling are orchestrated externally. This decision aligns perfectly with the principle of separating concerns pertinent to our research, ensuring our prediction models have clean and well-organized data to train on.

Though Python generators can be advantageous for iterating over previously generated iterables, it was vital to maintain clarity in our experimental code. Hence, a classical approach was adopted for internal handling of all iterations, with a particular focus on the sequential nature of the data - a vital aspect for time series prediction.

Below, we provide brief descriptions of each PRNG, enriched with general notes and observations accrued during the developmental phase, offering insights into their potential in time series analysis.

Middle Square:

- If the result has fewer than 2n digits, leading zeroes are added to compensate. The middle n digits of the result become the next number in the sequence, and are returned

as the result. This process is then repeated to generate more numbers, forming a sequence crucial for time series forecasting. [7]

Notes:

- Typically, the seed's value must be even, though leading zeros can adjust it if necessary.
- In the context of time series, a significant observation is that if the middle n digits are all zeroes, the generator will persistently output zeroes. Similarly, if the initial half of a sequence is zeroes, the subsequent sequence will inevitably trend towards zero, impacting the predictability.

Linear Congruential:

- The linear congruential generator, pivotal in time series modeling, is a PRNG typifying the "additive congruential method". It stems from endeavors to ameliorate the "unsatisfactory" entropy tests observed in Fibonacci sequences. For time series prediction, understanding the underlying sequences and periodicities this generator can produce is essential. [8]

Notes for **Linear Congruential**:

- In time series forecasting, a generator's sensitivity to parameter choice can play a pivotal role. The Linear Congruential generator is rather robust in this aspect. For instance, it's not particularly sensitive to the choice of c, provided it is relatively prime to the modulus. If, for example, m is a power of 2, c must be odd; hence, c=1 is a prevalent choice.
- If c = 0, the generator morphs into a multiplicative congruential generator (MCG), or what's commonly referred to as the Lehmer RNG (our choice of implementation). For time series applications, understanding these variations is imperative for choosing the right seed sequence. If c $\neq$ 0, the methodology adopts the moniker of a mixed congruential generator.
- Our choice of parameters was influenced by 2^32 numbers detailed in table 2 of the article "Tables of linear congruential generators of different sizes and good lattice structure." For time series models, this offers a balance between randomness and predictability. [9]

Lagged Fibonacci:

- The Lagged Fibonacci PRNG, which finds foundational resonance with the Fibonacci Sequence [10], evolves its sequences by the principle where the seed and the summation of the last two values culminate in the PRN. This PRN subsequently doubles up as the succeeding seed—a method particularly enticing for time series forecasting owing to its sequential nature.

Notes:

- When we refer to this as a "lagged" generator, it hints at how "j" and "k" lag behind the generated pseudorandom value. This inherent lag provides an element of predictability, crucial in time series analysis.
- Our rendition is a "two-tap" generator, signifying the use of 2 values in the sequence to architect the pseudorandom number. However, it's pivotal to recognize that a two-tap generator encounters challenges with some randomness tests, like the Birthday Spacings—a factor to consider in time series applications. Transitioning to a "three-tap" generator can potentially counteract this shortcoming.

Park Miller:

- Within the expanse of time series analysis, understanding the lineage of PRNGs can be pivotal. The Park Miller PRNG, colloquially known as Lehmer, can be envisioned as a specific instantiation of the Lehmer PRNG. The latter itself is a subset of the linear congruential PRNG, conditioned by c=0 and the stipulation of distinct parameters.

Notes:

- Providing context, in 1988, Park and Miller [11], endorsed a Lehmer RNG characterized by unique parameters: $m = 2^{31} - 1 = 2,147,483,647$ (a Mersenne prime M31) and $a = 7^5 = 16,807$ (a primitive root modulo M31). This methodology, today, is identified as MINSTD. For time series forecasting, the choice of such parameters ensures a blend of randomness and pattern, creating a conducive environment for modeling. [**?**]

Mersenne Twister:

- "The Mersenne Twister algorithm, widely recognized for its efficacy in time series predictions, operates on a matrix linear recurrence over a finite binary field" [12].

Notes:

- The Mersenne Twister exhibits similarities to a conventional LFSR. Notably, its MT19937 rendition is among the most prevalent in modern PRNG implementations, especially in time series forecasting.
- This algorithm has been so influential that it became the default generator in the Python programming language from version 2.3 onwards.
- To ensure optimal performance in time series forecasting using CNNs, the numpy version of the Mersenne Twister was utilized.

C. Experimental Setup

The segment presents a mathematical abstraction of the experimental model, amalgamating elements from TLA+ notation and set-builder theory. Successive sections will provide accompanying visuals and narratives to elucidate the underlying structure and functioning of the experimental framework in the realm of time series prediction.

Let n denote an arbitrary natural number, constrained by $\{n \in \mathbb{N}\}$. Symbolize the experiment definition as E. Let $seed$ act as a function that, upon invocation, retrieves a seed. Given parameters k and S_n, the function $prng$, when invoked, returns a collection of pseudorandom numbers. These numbers originate from an initial seed S_n, processed through a specified algorithm, resulting in a set of length k.

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

Fig. 1. Illustrative Representation of the Experimental Model

Given an ordered set, $\mathbb{N}$, where $prng$ represents the selected PRNG, S_n refers to the nth seed, and k determines the desired vector length ($\mathbb{L}_n$) of $prng$, the set $\mathbb{X}_n$ encompasses n collections of $k-1$ values, each stemming from the PRNG ($prng$) but with a distinct seed. In turn, $\mathbb{Y}_n$ consists of n sets of kth values, each corresponding to an $\mathbb{X}_n$ set as per the relationship $\mathbb{X}_n \mapsto \mathbb{Y}_n$. The function P operates as a predictor, exemplifying a convolutional neural network. Accepting $\mathbb{X}_n$ and $\mathbb{Y}_n$, it generates a new set $\mathbb{B}_n$ derived from $\mathbb{X}_n$. This CNN trains $\mathbb{B}_n$ to approach or match $\mathbb{Y}_n$, leveraging back-propagation from prior predictions, thus harnessing supervised learning to craft a regression model suitable for predicting time series using CNNs.

For a streamlined visual elucidation of this description, one may refer to the predictive model depicted in Figure 3, the elementary experimental model in Figure 2, and the detailed perspective of the experimental model in Figure 4.

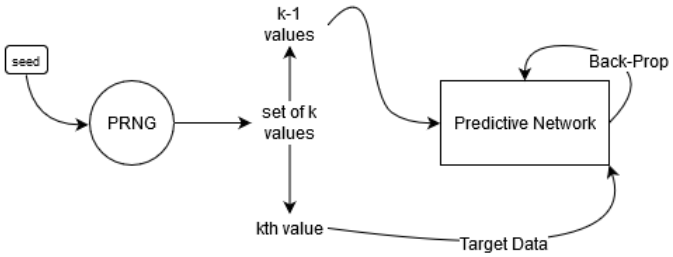


Fig. 2. Simplified Experimental Model

As illustrated in Figure 2, the experiment can be conceptualized as a 1-dimensional architecture. Here, the predictive neural network (shown in Figure 3) receives the outputs of a particular PRNG. The goal is to predict a kth value based on the preceding $k-1$ values in a specific set of input data. Each such set is generated from a unique seed. Through training on numerous such sets, the neural network develops a refined stochastic understanding of the underlying PRNG. This enables more accurate future predictions of numbers generated by the same PRNG, essentially functioning as a supervised regression model suitable for time series prediction.

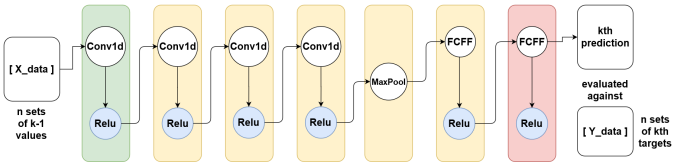


Fig. 3. Predictive Model

Figure 3 visualizes the layer architecture employed for the predictive network in each experimental trial. It features four stacked convolutional layers, each with four filters, a kernel size of 2, and stride 1. This is followed by a max-pooling layer and two fully connected feed-forward (FCFF) layers with 4 and 1 units, respectively. This architecture, adapted from existing research [13], enables the network to uncover complex patterns in the input data, thereby enhancing its capability to perform time series prediction. The use of ReLU activation functions instead of leaky-ReLUs is one of the distinctions in this model.

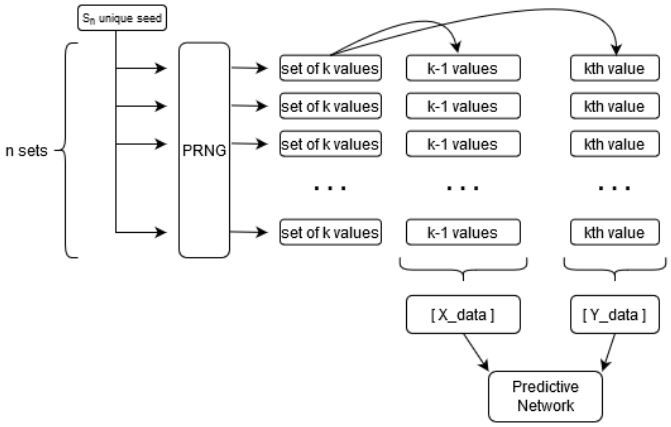


Fig. 4. Granular Experimental Model

Figure 4 offers a detailed depiction of the experiment's structure, showcasing its multi-dimensional aspects in data generation and aggregation. The same general process as shown in Figure 3 is followed. Here, the data set outputted from the PRNG, of length k, is divided into vectors containing $k-1$ and kth values, respectively. This process is executed twice—once to generate the training data and again for the testing data. Unlike many neural network applications, where training and testing data sets are finite and pre-validated, the use of PRNs here allows for the generation of arbitrarily large amounts of training and testing data, assuming the PRNG employed is mathematically sound and algorithmically robust. This flexibility is especially advantageous in building robust models for time series prediction.

D. Experimental Execution

The experimental process and predictive model, as detailed in the previous section, were implemented for the execution

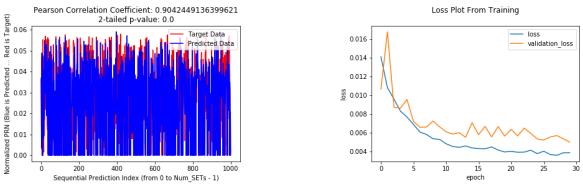
of the experiment. Each PRNG was individually loaded and the experiment was executed with the following parameters for training:

- Number of sets: 1000
- Length of each set (where each set gets a new seed): 2000
- batch size: 15
- Number of epochs: 30
- Validation split: 0.3
- Optimization method: nadam
- Learning rate: 0.001
- Loss method: mean absolute error

Following the training process, five separate trained prediction models corresponding to each PRNG were produced. These models were then tested using the testing data, with correlation coefficients produced to evaluate how effectively each model represented new test data from its respective PRNG.

IV. Results

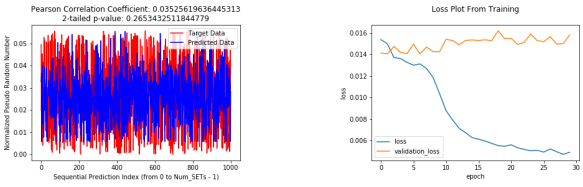
Presented below are the regression and loss plots obtained from testing each predictive model.



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

Fig. 5. Middle Square Results

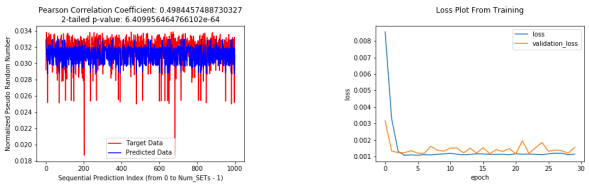
Referring to Figure 5, the predictive model performed commendably, accounting for approximately 90% of the variability in the data from the middle square PRNG.



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

Fig. 6. Linear Congruential Results

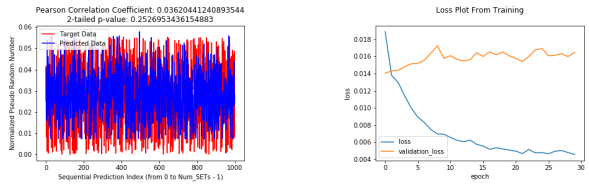
Referring to Figure 6, the predictive model had a suboptimal performance, accounting for only about 3.5% of the variability in the data from the linear congruential PRNG. Notably, with a 2-tailed p-value of 0.26, there's weak evidence supporting the efficacy of the model, hence no definitive conclusions can be drawn. Increasing the quantity of training data might enhance this model, potentially leading to a higher correlation coefficient and a more significant p-value. Additional training time could further bolster its performance.



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

Fig. 7. Lagged Fibonacci Results

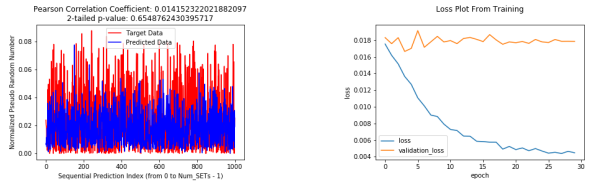
Referring to Figure 7, the predictive model exhibited a performance representing about 49% of the variability in the data from the lagged fibonacci PRNG. A 2-tailed p-value of 6.4, significantly worse than the linear congruential, indicates weak evidence supporting the model's efficacy. Increasing the quantity of training data and allocating more time for training may lead to improved results, potentially reflected by a higher correlation coefficient and a more significant p-value.



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

Fig. 8. Park Miller Results

Referring to Figure 8, the predictive model demonstrated a performance covering about 3.6% of the variability in the data from the Park Miller PRNG. With a 2-tailed p-value of 0.25, there's weak evidence supporting the model's efficacy. It is suggested that expanding the training dataset and investing more time in training may enhance the model's performance, leading to a higher correlation coefficient and a more significant p-value.



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

Fig. 9. Mersenne Twister Results

V. Discussion

The primary objective of this research was to train a predictive neural network on various PRNG outputs and derive general observations regarding the evolution of PRNGs over time.

During the course of this study, several challenges were encountered, and potential inaccuracies were identified. Given the intricate nature of PRN generation, there are numerous facets that could have been refined to minimize experimental discrepancies.

Issues were observed in seed generation for the PRNG implementations. A more comprehensive seeding approach might help negate the tendency of the seed to stabilize at 0. While this might be attributed to a flaw in the code, it predominantly manifested as an experimental hindrance. Notably, the middle-square PRNG implementation occasionally converged to zero. This tendency, particularly associated with the Middle-Square algorithm, can sometimes result in convergence given specific numbers.

While the PRNG methods employed were deemed accurate, it's worth noting that they might not be exemplary representations. Several generators have stringent criteria concerning parameters and operational logic. Though the implementations in this study are believed to be accurate, validating them using statistical randomness tests, such as those offered by the National Institute of Standards and Technology (NIST), would be advantageous. The seeding methodology utilized, being relatively simplistic, might also influence the observed results.

This research provides a foundation, offering an architectural template that future researchers might find beneficial for exploring enhanced seeding or generation techniques. Moreover, the experiment was limited to 30 epochs, suggesting that more computational resources might further refine the outcomes.

From a practical perspective, this predictive model holds potential in the realm of real-time prediction attacks. Should adversaries manage to compromise a robust PRNG pivotal for cryptographic purposes, techniques akin to the one explored in this study could play a pivotal role in mitigating or decelerating subsequent PRN generation cycles that could be vulnerable.

Future studies are urged to delve deeper into refining seeding approaches, PRNG methodologies, training criteria, and model selection to yield results that are more mathematically rigorous and robust.

ACKNOWLEDGMENT

All that I am, or hope to be, I owe to my angel mother.

REFERENCES

[1] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.

[2] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher *et al.*, "A survey of uncertainty in deep neural networks," *Artificial Intelligence Review*, pp. 1–77, 2023.

[3] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, "Lagrangian neural networks," *arXiv preprint arXiv:2003.04630*, 2020.

[4] L. Wu, P. Cui, J. Pei, L. Zhao, and X. Guo, "Graph neural networks: foundation, frontiers and applications," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 4840–4841.

[5] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 2019, pp. 443–452.

[6] D. Bau, J.-Y. Zhu, H. Strobelt, A. Lapedriza, B. Zhou, and A. Torralba, "Understanding the role of individual units in a deep neural network," *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30 071–30 078, 2020.

[7] M. Liu, H. Gao, and S. Ji, "Towards deeper graph neural networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 338–348.

[8] A. Abbas, D. Sutter, C. Zoufal, A. Lucchi, A. Figalli, and S. Woerner, "The power of quantum neural networks," *Nature Computational Science*, vol. 1, no. 6, pp. 403–409, 2021.

[9] Z. Hu, Y. Dong, K. Wang, K.-W. Chang, and Y. Sun, "Gpt-gnn: Generative pre-training of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1857–1867.

[10] S. Tang, D. Chen, L. Bai, K. Liu, Y. Ge, and W. Ouyang, "Mutual crf-gnn for few-shot learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 2329–2339.

[11] Y. Liu, X. Ao, Z. Qin, J. Chi, J. Feng, H. Yang, and Q. He, "Pick and choose: a gnn-based imbalanced learning approach for fraud detection," in *Proceedings of the web conference 2021*, 2021, pp. 3168–3177.

[12] Z.-A. Shen, T. Luo, Y.-K. Zhou, H. Yu, and P.-F. Du, "Npi-gnn: predicting ncrna–protein interactions with deep graph neural networks," *Briefings in bioinformatics*, vol. 22, no. 5, p. bbab051, 2021.

[13] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "Gnnlab: a factored system for sample-based gnn training over gpus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 417–434.