Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 170 (2020) 1122–1127

Procedia Computer Science

www.elsevier.com/locate/procedia

International Workshop on Statistical Methods and Artificial Intelligence (IWSMAI)
April 6 - 9, 2020, Warsaw, Poland

Pseudo Random Number Generation: a Reinforcement Learning approach

Luca Pasqualini[a,*], Maurizio Parton[b]

[a]*Department of Information Engineering and Mathematical Sciences - University of Siena, Via Banchi di Sotto 55, 53100 Siena, Italy*
[b]*Department of Economical Studies - University of Chieti-Pescara, 65129 Pescara, Italy*

Abstract

Pseudo-Random Numbers Generators (PRNGs) are algorithms produced to generate long sequences of statistically uncorrelated numbers, i.e. Pseudo-Random Numbers (PRNs). These numbers are widely employed in mid-level cryptography and in software applications. Test suites are used to evaluate PRNGs quality by checking statistical properties of the generated sequences.

Machine learning techniques are often used to break these generators, i.e. approximating a certain generator or a certain sequence using a neural network. But what about using machine learning to generate PRNs generators?

This paper proposes a Reinforcement Learning (RL) approach to the task of generating PRNGs from scratch by learning a policy to solve an N-dimensional navigation problem. In this context, N is the length of the period of the sequence to generate and the policy is iteratively improved using the average score of an appropriate test suite run over that period.

Aim of this work is to demonstrate the feasibility of the proposed approach, to compare it with classical methods, and to lay the foundation of a research path which combines RL and PRNGs.

Keywords: Pseudo-Random Number; Machine Learning; Reinforcement Learning; Deep Learning; Neural Networks

1. Introduction

Pseudo-Random Numbers (PRNs) sequence generation is a task of renown importance in cryptography and more in general in computer science. A Pseudo-Random Number Generator (PRNG) is a (usually, deterministic) algorithm which tries to emulate the statistical properties of a sequence of True-Random Numbers (TRNs). PRNGs are used in applications related to gambling, statistical sampling, computer simulation and in other areas where producing an unpredictable result is desirable. While TRN sequences are overall more unpredictable and as such better keys for cryptography systems, they are usually expensive to generate. Indeed, a True Random Number Generator (TRNG)

* Corresponding author. Tel.: +39-339-848-1206.
E-mail address: pasqualini@diism.unisi.it

1877-0509
10.1016/j.procs.2020.03.057

relies on natural phenomena like atmospheric or thermal noise, radioactive decay or cosmic background radiation. Measurement of these is known to be expensive. A wide variety of methods have been employed up to now to define PRNGs. To measure their quality some kind of statistical test suite is run over the result, i.e. the generated sequence usually analyzed in its binary format. In this paper the National Institute for Standard and Technologies (NIST) statistical test suite for random and pseudo-random number generators [2] is used to validate the PRNG.

Machine Learning (ML) is a field of artificial intelligence studying algorithms and statistical models to be used by computer systems to perform tasks without explicit instructions. Nowadays, widely used statistical models are a class of function approximators called Neural Networks (NNs). In particular, Deep Neural Networks (DNNs), that is, NNs with several hidden layers, are successfully employed in many fields like image recognition, natural language processing, games, etc. For the above reasons, ML has been used in the field of PRNGs to approximate generators using target sequences of pseudo-random or true-random bits. This technique is very useful when the goal is predicting the output of an existing generator, e.g. to break the key of a cryptography system.

There have been limited attempts at generating PRNGs using NNs by exploiting their structure and internal dynamics. For example, the authors of [3] use Recurrent Neural Networks (RNNs) dynamics to generate PRNs. In [4], the authors use the dynamics of a feed forward NN with random orthogonal weight matrices to generate PRNs. Neuronal plasticity is used in [1] instead. In [5] a Generative Adversarial Network (GAN) approach to the task is presented, exploiting an input source of randomness (like an existing PRNG or a TRNG).

This paper proposes a novel approach to the task of generating PRNGs from scratch. The proposed approach works without data nor external inputs and without employing any structural dynamics. Indeed, it works just by using Deep Reinforcement Learning (DRL), that is, RL and a DNN as function approximator. Our approach generates a sequence of PRNs with variable period by directly optimizing its score computed by the NIST test suite. We show both the advantages of our approach and future research paths required to overcome its current limits.

2. Methodology

When a PRNG generates a sequence of bits, it manipulates a starting state, called seed, according to a certain algorithm that generates following states. It can then be thought as a sequence of decisions made from a starting state following a certain strategy. In this setting, a good PRNG corresponds to a good strategy in a decision-making process. The state could be represented by the sequence itself up to that point. The strategy could instead be the action of changing some parts of the sequence.

2.1. Reinforcement Learning

For a comprehensive, motivational and thorough introduction to RL, we strongly suggest reading sections from 1.1 to 1.6 in [10]. In the context of ML, RL is learning what to do in order to accumulate as much reward as possible in a *Markov Decision Process* (MDP) $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$, where the Markov property is only a restriction on the states $\mathcal{S}$. All RL algorithms used in this paper use function approximation via a DNN.

2.2. N-Dimensional Navigation

From the considerations described at the beginning of this section, the problem of generating PRNs seems suitable for RL. There is however one caveat: the naive approach that comes to mind, that is, using as state the last generated number, is inherently not Markovian: whatever reward we use to measure the randomness of the sequence, it must depends on the whole sequence that was generated before. Using as state the whole sequence, and increasing the length of the sequence by appending a new number is calling for the curse of dimensionality!

The goal of this decision task is to perturb the sequence without changing its length, and this can be obtained by adding or subtracting a certain fixed value at some position in the sequence. Actions will then be given by ”add or subtract, position” pairs, and will be described more in detail in next subsections concerning each specific formulation.Finally, we model the task as finite-horizon episodic, by fixing a termination time T. Thus, the state S_T is the output sequence of the PRNG, i.e. the sequence of N PRNs. A fixed length output however violates one requirement of PRNGs, that is, a PRNG has to be able to generate a (possibly) infinite amount of numbers. To solve this problem,

we can think of the generated sequence as the period of the PRNG. By concatenating multiple output of the same PRNG it is possible to obtain a (possibly) infinite amount of numbers. Please note that a feature of this RL approach is the fact that this period is also variable, as we will show in the next section. The starting state of this task should be given by a number, known as the seed. According to our definition of state, we set all N numbers of the sequence corresponding to the starting state equal to the seed value, e.g. 0.

This in general an N-dimensional navigation task. When actions are chosen by a policy maximizing a total reward associated to "randomness" in some sense, this navigation will end up in a PRNs sequence. A simple yet informative reward function could be the average of the scores given by the NIST tests computed over the binary representation of the sequence at certain time steps. The NIST test suite and its tests are described more in detail in section 2.4.

2.3. Binary Formulation

Binary Formulation (BF) is based directly on the binary representation of the sequences as analyzed by the NIST test suite. Assume each integer value is represented by m-bits. Given a sequence of integer numbers of length N, the state is B-dimensional and defined as

$$S = [b_1, b_2, \ldots, b_B]$$

where $b_1, b_2, \ldots, b_B$ are binary values and $B = m * N$. Its cardinality is $2^B = 2^{m \cdot N}$. The action set is defined as

$$\mathcal{A} = \bigcup_{n=1}^{B} \{1_n, 0_n\}$$

where 1_n is the action of setting the n bit to 1 and 0_n is the action of setting the n bit to 0. The null action, that is, do nothing, is not required since to keep current position in the space the agent can just set a certain bit to its current value. The action set is discrete and its size is $2 \cdot B$, i.e. $2 \cdot m \cdot N$. While binary representation can produce terminal states uncorrelated to seed states, probably because of the reduced sparseness of the good states, the size of the action set is large even for short decimal sequences.

2.4. NIST Test Suite

The NIST statistical test suite for random and pseudo-random number generators is the most popular application to test the randomness of sequences of bits. In this paper, it is used to compute the average value of all eligible tests in the battery run on the generated PRNs sequences. If a test is failed its value is set to zero. Some tests are not eligible on certain sequences because of their length, and in this case they are not considered for the average. This value is then used as a reward function for the MDP we are defining. Experimentally we saw performance to be better when reward is assigned only at the end of the episode, even if that means incurring in the credit assignment problem. Please notice that, since test statistic values are probabilities according to statistical hypothesis definition, rewards belong to $[0, 1]$.

3. Experiments

Our experiments consists on multiple sets of training processes with different hyperparameters. We divide the experiments according to the formulation and reward function used, and by the RL algorithm employed. The combination of the first two gives the RL environment, while the latter is the RL agent. The environment employed in the shown experiments uses BF for states and actions, and assigns a reward $R_t = 0$ at every time step $t \neq T$. At termination

time T the reward is the average value of the NIST test suite over the state S_T:

$$R_t = \begin{cases} \text{avg}_{\text{NIST}}(S_t) & \text{if } t = T \\ 0 & \text{otherwise} \end{cases}$$

We experimented with three model-free RL algorithms: two on-policy policy gradient algorithms (Vanilla Policy Gradient and Proximal Policy Optimization) and one off-policy (deep Q-learning in its Dueling Deep Q-Network [12] flavor, proved to be better than the original DQN algorithm from [6], with Prioritized Experience Replay as in [7]). From our experiments only policy gradient algorithms succeeded in solving the task without great instability.

3.1. Framework

The framework used for the RL algorithms is USienaRL[1]. This framework allows for environment, agent and interface definition using a preset of customizable models. The NIST test battery is run with another framework, called NistRng[2]. Finally, the code for this article can be found at this GitHub repository: *https://github.com/InsaneMonster/pasqualini2019prngrl*.

3.2. Experimental setup

The policy optimization algorithms we used are Vanilla Policy Gradient (VPG) [11] and Proximal Policy Optimization (PPO) [9], both with Generalized Advantage Estimation (GAE) [8]. The VPG model used in our experiments is composed by six dense layers with 4096 neurons each initialized with Xavier initialization, the learning rate of the policy stream is set to $3e-4$ and the learning rate of the value stream is set to $1e-4$. Discount factor is $\gamma = 0.99$ and $\lambda = 0.95$. At each update, 80 value steps are performed. The model is updated 10 times each volley. The PPO model is likewise composed by six dense layers with 4096 neurons each initialized with Xavier initialization. The learning rate of the policy stream is set to $3e-4$ and the learning rate of the value stream is set to $1e-4$. Discount factor is $\gamma = 0.99$ and $\lambda = 0.97$. At each update, 80 policy and value steps are performed. The model is updated 10 times each volley. The clip ratio is set to 0.2 and the target KL divergence is 0.01.

3.3. Results

In this subsection we present experimental results in the form of plots. We set the environment to $B = 80$ and then $B = 200$, respectively environments with sequences of 80 and 200 bits. Since the reward is assigned only at the end, our performance evaluation criteria is the average total reward over the volley, i.e. the average reward per episode over the training volley. A training volley is set of training episodes, or trajectories.

We can see in figure 1 that for sequences of 80 bits both VPG and PPO converge with consistent results. We also note that our model average score is greater than the reference average score over 1000 sequences of equal fixed length $B = 80$ generated by the NumPy uniform PRNG. We believe this to be a very interesting result. Apparently, for sequences of length 200 VPG is able to converge, as seen in 2a, but not in all experiments. PPO, while having better training trends overall as shown in figure 2b, proves to be even too cautious in improving its policy and tends to converge to worse final results or to converge in more steps. In table 1 some generated sequences are shown in decimal representation. All the generated sequences are results of one trained PRNG with one seed state. Since these fixed length sequence can be considered the period of the generator, the trained PRNG has variable period..

[1] Available on PyPi and also on GitHub: https://github.com/InsaneMonster/USienaRL.
[2] Available on PyPi and also on GitHub: https://github.com/InsaneMonster/NistRng.

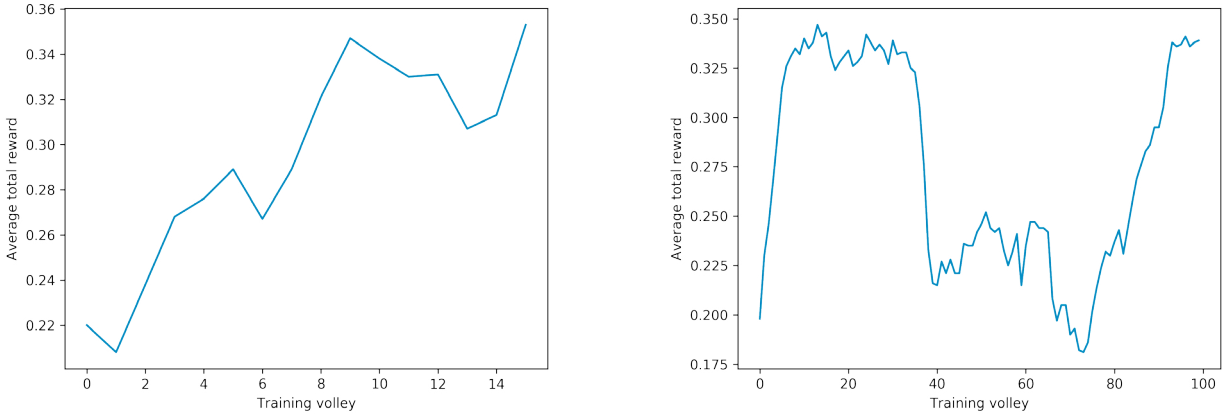

Fig. 1. Average total reward during training of VPG (a) and PPO (b) on BF, reward only at the end environment with $B = 80$. Volleys are composed by 1000 episodes each. The fixed length of each trajectory is $T = 100$ steps. Reference average value over 1000 sequences of same binary length $B = 80$ generated by NumPy uniform PRNG is of 0.33.

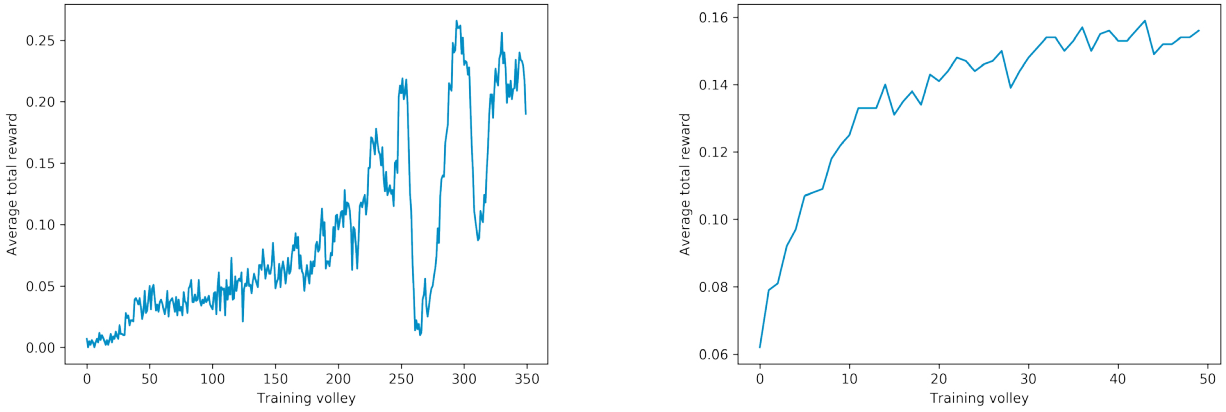

Fig. 2. Average total reward during training of VPG (a) and PPO (b) on BF, reward only at the end environment with $B = 200$. Volleys are composed by 1000 episodes each. The fixed length of each trajectory is $T = 100$ steps. Reference average value over 1000 sequences of same binary length $B = 200$ generated by NumPy uniform PRNG is of 0.35.

Table 1. Sequences in decimal representation of one trained PRNG with *seed* = 0 and $B = 80$, alongside their average score.

Sequence										Average Score
-10	-112	68	-39	-123	35	-45	66	-28	62	0.24
22	-113	34	-111	44	42	63	114	-63	-41	0.57
-48	-111	20	-102	10	-18	55	11	80	62	0.16

4. Conclusions

In this paper we propose a way to automatically generate PRNGs, a task of interest and a currently open field of research. Our approach uses RL to build a PRNG from scratch. To the best of our knowledge, this is a novel approach and results are promising. Our approach also presents the following interesting features:

- It requires no input data, so that the generated PRNG is always a novel algorithm.
- Each time a training process is run the resulting PRNGs are likely to be different from each other. This is an inherent property of RL as a whole.
- For a single starting state, multiple solutions can be obtained after one training process since RL policies can be stochastic. In short, we obtain a non-deterministic PRNG given a single seed. This is a novel property which is not present in current state-of-the-art PRNGs.
- Given that NNs are black-box approximators, the policy of the RL agent is black-box. Since the PRNG is the algorithm given by that policy, it is also black-box. This grants the nice property of having no human insights in the inner functioning of the PRNG.

Finally, results prove that this approach is feasible, and the task can be learned by RL techniques. We hope this paper will inspire future research which combines PRNGs and RL.

4.1. Future work

The current main limitation of our approach is the dimensionality N of the state, i.e. the period of the PRNG. Our experiments show that, at least on average hardware, is very complex to successfully train an agent on longer sequences, thus obtaining a PRNG with longer period. This will be the main focus of our research to come. Beside that, we aim to do the following:

- Improve the quality of the output sequence, i.e. each period, especially for longer sequences. This means to increase the average value obtained by the NIST test battery over the fixed size sequences.
- Increase the amount of supported seeds. We aim at reducing the variance introduced by additional seeds during learning. We believe that processing somehow the vector representation (for example with convolutional filters) could be a promising path to follow.
- Devise a better strategy to concatenate the output sequences. Ideally, this could also be learned, for example with hierarchical RL.
- Move towards a formulation in which the size of the action space does not grow (too much) with the length N of the sequence, without losing (too much) the ability to reach points in the lattice with a high value.

References

[1] Abdi, H., 1994. A neural network primer. Journal of Biological Systems 2, 247–281.
[2] Bassham III, L.E., Rukhin, A.L., Soto, J., Nechvatal, J.R., Smid, M.E., Barker, E.B., Leigh, S.D., Levenson, M., Vangel, M., Banks, D.L., et al., 2010. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications .
[3] Desai, V., Patil, R., Rao, D., 2012. Using layer recurrent neural network to generate pseudo random number sequences. International Journal of Computer Science Issues 9, 324–334.
[4] Hughes, J.M., 2007. Pseudo-random Number Generation Using Binary Recurrent Neural Networks. Ph.D. thesis.
[5] Marcello De Bernardi, M., Malacaria, P., . Pseudo-random number generation using generative adversarial networks, in: Workshop Proceedings.
[6] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A., 2013. Playing atari with deep reinforcement learning. ArXiv abs/1312.5602.
[7] Schaul, T., Quan, J., Antonoglou, I., Silver, D., 2015. Prioritized experience replay. arXiv preprint arXiv:1511.05952 .
[8] Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P., 2015. High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438 .
[9] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 .
[10] Sutton, R., Barto, A., 2018. Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning series, MIT Press. URL: https://books.google.it/books?id=6DKPtQEACAAJ.
[11] Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y., 2000. Policy gradient methods for reinforcement learning with function approximation, in: Advances in neural information processing systems, pp. 1057–1063.
[12] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., De Freitas, N., 2015. Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581 .