Pseudo Random Number Generation through Reinforcement Learning and Recurrent Neural Networks

Luca Pasqualini [1,*] **and Maurizio Parton** [2]

1   Department of Information Engineering and Mathematical Sciences, University of Siena, 53100 Siena, Italy
2   Department of Economical Studies, University of Chieti-Pescara, 65129 Pescara, Italy; parton@unich.it
*   Correspondence: pasqualini@diism.unisi.it

check for updates

Abstract: A Pseudo-Random Number Generator (PRNG) is any algorithm generating a sequence of numbers approximating properties of random numbers. These numbers are widely employed in mid-level cryptography and in software applications. Test suites are used to evaluate the quality of PRNGs by checking statistical properties of the generated sequences. These sequences are commonly represented bit by bit. This paper proposes a Reinforcement Learning (RL) approach to the task of generating PRNGs from scratch by learning a policy to solve a partially observable Markov Decision Process (MDP), where the full state is the period of the generated sequence, and the observation at each time-step is the last sequence of bits appended to such states. We use Long-Short Term Memory (LSTM) architecture to model the temporal relationship between observations at different time-steps by tasking the LSTM memory with the extraction of significant features of the hidden portion of the MDP's states. We show that modeling a PRNG with a partially observable MDP and an LSTM architecture largely improves the results of the fully observable feedforward RL approach introduced in previous work.

1. Introduction

Generating random numbers is an important task in cryptography, and more generally in computer science. Random numbers are used in several applications, whenever producing an unpredictable result is desirable—for instance, in games, gambling, encryption algorithms, statistical sampling, computer simulation, and modeling.

An algorithm generating a sequence of numbers approximating properties of random numbers is called a Pseudo-Random Number Generator (PRNG). A sequence generated by a PRNG is "pseudo-random" in the sense that it is generated by a deterministic function: the function can be extremely complex, but the same input will give the same sequence. The input of a PRNG is called the seed, and to improve the randomness, the seed itself can be drawn from a probability distribution. Of course, assuming that drawing from the probability distribution is implemented with an algorithm, everything is still deterministic at a lower level, and the sequence will repeat after a fixed, unknown number of digits, called the *period* of the PRNG.

While true random number sequences are more fit to certain applications where true randomness is necessary—for instance, cryptography—they can be very expensive to generate. In most applications, a pseudo-random number sequence is good enough.

The quality of a PRNG is measured by the randomness of the generated sequences. The randomness of a specific sequence can be estimated by running some kind of statistical test suite.

In this paper, the National Institute of Standards and Technology (NIST) statistical test suite for random and pseudo-random number generators [1] is used to validate the PRNG.

Neural networks have been used for predicting the output of an existing generator, that is, to break the key of a cryptography system. There have also been limited attempts at generating PRNGs using neural networks by exploiting their structure and internal dynamics. For example, the authors of [2] use recurrent neural networks dynamics to generate pseudo-random numbers. In [3], the authors use the dynamics of a feedforward neural network with random orthogonal weight matrices to generate pseudo-random numbers. Neuronal plasticity is used in [4] instead. In [5], a generative adversarial network approach to the task is presented, exploiting an input source of randomness, like an existing PRNG or a true random number generator.

A PRNG usually generates the sequence incrementally, that is, it starts from the seed at time $t = 0$ to generate the first number of the sequence at time $t = 1$, then the second at $t = 2$, and so on. Thus, it is naturally modeled by a deterministic Markov Decision Process (MDP), where state space, action space, and rewards can be chosen in several ways. A Deep Reinforcement Learning (DRL) pipeline can then be used on this MDP to train a PRNG agent. This DRL approach has been used for the first time in [6] with promising results, see modeling details in Section 2.2. This is a probabilistic approach that generates pseudo-random numbers with a "variable period", because the learned policy will generally be stochastic. This is a feature of this RL approach.

However, the MDP formulation in [6] has an action set where the size grows linearly with the length of the sequence. This is a severe limiting factor, because when the action set is above a certain size it becomes very difficult, if not impossible, for an agent to explore the action space within a reasonable time.

In this paper, we overcome the above limitation with a different MDP formulation, using a partially observable state. By observing only the last part of the sequence, and using the hidden state of a Long-Short Term Memory (LSTM) neural network to extract important features of the full state, we significantly improve the results in [6], see Section 3. The code for this article can be found at GitHub repository [7].

2. Materials and Methods

The main idea is quite natural: since a PRNG builds the random sequence incrementally, we model it as an agent in a suitable MDP, and use the DRL to train the agent. Several "hyperparameters" of this DRL pipeline must be chosen: a good notion of states and actions, a reward such that its maximization gives sequences as close as possible to true random sequences, and a DRL algorithm to train the agent.

In this section, we first introduce the DRL notions that will be used in the paper, see Section 2.1; we then describe the fully observable MDP used in [6] and the partially observable MDP formulation used in this paper, see Section 2.2; in Section 2.3 we describe the reward function used for the MDP; in Section 2.4 we describe the recurrent neural network architecture used to model the environment state; and finally, Section 2.5 describes the software framework used for the experiments.
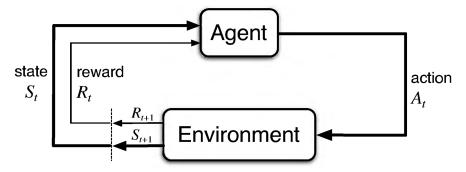
2.1. Reinforcement Learning (RL)

For a comprehensive, motivational, and thorough introduction to RL, we strongly suggest reading from 1.1 to 1.6 in [8]. RL is learning what to do in order to accumulate as much reinforcement as possible during the course of action. This very general description, known as *the RL problem*, can be framed as a sequential decision-making problem, as follows.

Assume an *agent* is interacting with an *environment*. When the agent is in a certain situation—a *state*—it has several options, called *actions*. After each action, the environment will take the agent to a next state, and will provide it with a numerical *reward*, where the pair "state, reward" may possibly be drawn from a joint probability distribution, called the *model* or the *dynamics* of the environment. The agent will choose actions according to a certain strategy, called *policy* in the RL setting. The RL problem can then be stated as finding a policy maximizing the expected value

of the total reward accumulated during the agent–environment interaction. To formalize the above description, see Figure 1 representing the agent–environment interaction.

Figure 1. The agent–environment interaction is made at discrete time-steps $t = 0, 1, 2, \ldots$. At each time-step t, the agent uses the state $S_t \in \mathcal{S}$ given by the environment to select an action $A_t \in \mathcal{A}$. The environment answers with a real number $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, called a reward, as well as a next state S_{t+1}. Going on, we obtain a trajectory $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ Ref. [8] (Figure 3.1).

At each time-step t, the agent receives a state $S_t \in \mathcal{S}$ from the environment, and then selects an action $A_t \in \mathcal{A}$. The environment answers with a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and a next state S_{t+1}. This interaction gives rise to a *trajectory* of random variables:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

In the case of interest to us, $\mathcal{S}, \mathcal{A}$, and $\mathcal{R}$ are finite sets. Thus, the environment answers the action $A_t = a$ executed in the state $S_t = s$ with a pair $R_{t+1} = r, S_{t+1} = s'$ drawn from a discrete probability distribution p on $\mathcal{S} \times \mathcal{R}$, the model (or dynamics) of the environment:

$$p(s', r|s, a) := p(s', r, s, a) := \Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a).$$

Note the visual clue of the fact that $p(\cdot, \cdot|s, a)$ is a probability distribution for every state-action pair s, a. Figure 1 implicitly assumes that the joint probability distribution of S_{t+1}, R_{t+1} depends on the past only via S_t and A_t. In fact, the environment is fed only with the last action, and no other data from the history. This means that, for a fixed policy, the corresponding stochastic process $\{S_t\}$ is Markov. This gives the name Markov Decision Process (MDP) to the data $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$. Moreover, it is a *time-homogeneous* Markov process, because p does not depend on t. In certain problems, the agent can see only a portion of the full state, called *observation*. In this case, we say that the MDP is *partially observable*. Observations are usually not Markov, because the non-observed portion of the state can contain relevant information for the future. In this paper, we model a PRNG as an agent in a partially observable MDP. When the agent experiences a trajectory starting at time t, it accumulates a *discounted return* G_t:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma \in [0, 1].$$

The return G_t is a random variable, whose probability distribution depends not only on the model p, but also on how the agent chooses actions in a certain state s. Choices of actions are encoded by the policy, that is, a discrete probability distribution π on $\mathcal{A}$:

$$\pi(a|s) := \pi(a, s) := \Pr(A_t = a|S_t = s).$$

A discount factor $\gamma < 1$ is used mainly when rewards far in the future are becoming increasingly less reliable or important, or in *continuing* tasks, that is, when the trajectories do not decompose naturally into *episodes*. Since the partially observable MDP formulation we use is episodic (see Section 2.2), in this paper we use $\gamma = 1$. The average return from a state s, that is, the average

total reward the agent can accumulate starting from *s*, represents how good the state *s* is for the agent *following the policy* π, and it is called a *state-value* function:

$$v_\pi(s) := E_\pi[G_t|S_t = s].$$

Likewise, one can define the *action-value* function (known also as the *quality* or *q-value*), encoding how good *choosing an action a from s and then following the policy* π is:

$$q_\pi(s,a) := E_\pi[G_t|S_t = s, A_t = a].$$

Since the return G_t is recursively given by $R_{t+1} + \gamma G_{t+1}$, the RL problem has an optimal substructure, expressed by recursive equations for v_* and q_*. If an accurate description of the dynamic *p* of the environment is available and if one can store all states into memory, then dynamic programming iterative techniques can be used, and an approximate solution v_* or q_* to the Bellman optimality equations can be found. From v_* or q_*, one can then easily recover an optimal policy—for instance, $\pi_*(s) := \text{argmax}_{a \in \mathcal{A}} q_*(s,a)$ is a deterministic optimal policy.

However, in most problems we have only *partial knowledge of the dynamics*, if any. This can be overcome by *sampling trajectories* $S_t = s, A_t = a, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, \ldots$ to *estimate* the q-value $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, instead of computing a true expectation. Moreover, in most problems *there are way too many states to store them in memory*, or we just have to go through every state once. In this case, the estimate of $q_\pi(s,a)$ must be stored in a parametric function approximator $q_\pi(s,a;w)$, where *w* is a parameter vector living in a dimension much lower than $|\mathcal{S} \times \mathcal{A}|$. Due to their high representational power, deep neural networks are nowadays widely used as approximators in RL, where the combination of deep neural networks with RL is called Deep Reinforcement Learning (DRL).

Iterative dynamic programming techniques can then be approximated, giving a family of algorithms known as *Generalized Policy Iteration* algorithms. They work by sampling trajectories to obtain estimates of the true values $\mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, and use supervised learning to find the optimal parameters vector *w* for $q_\pi(s,a;w)$. This estimated q-value is used to find a policy π' better than π, and iterating over this evaluation-improvement loop usually gives an approximate solution to the RL problem.

Generalized Policy Iteration is *value-based*, because it uses a value function as a proxy for the optimal policy. A completely different approach to the RL problem is given by Policy Gradient (PG) algorithms. They directly estimate the policy $\pi(a|s;\theta)$, without using a value function. The parameters vector θ_t at time *t* is modified to maximize a suitable scalar performance function $J(\theta)$, with the gradient ascent update rule:

$$\theta_{t+1} := \theta_t + \alpha \widehat{\nabla J(\theta_t)}.$$

Here, the *learning rate* α is the step size of the gradient ascent algorithm, determining how much we are trying to improve the policy at each update, and $\widehat{\nabla J(\theta_t)}$ is any estimate of the performance gradient $\nabla J(\theta)$ of the policy. Different choices for the estimator corresponds to different PG algorithms. The vanilla choice for the estimator $\widehat{\nabla J(\theta_t)}$ is given by the Policy Gradient Theorem, leading to an algorithm called REINFORCE and to its baselined derivatives—see, for instance, [8] (Section 13.2 and onwards). Unfortunately, vanilla PG algorithms can be very sensitive to the learning rate, and a single update with a large α can spoil the performance of the policy learned so far. Moreover, the variance of the Monte Carlo estimate is high, and a huge amount of episodes are required for convergence. For this reason, several alternatives for $\widehat{\nabla J(\theta_t)}$ have been researched.

In this paper, we use the best-performing algorithm in [6], a PG algorithm called Proximal Policy Optimization (PPO), described in [9] and considered state-of-the-art in PG methods. PPO tries to take the biggest possible improvement step on a policy using the data it currently has, without stepping too far and making the performance collapse.

The PPO used in this paper is an instance of PPO-Clip, as described by OpenAI at [10], with only partially shared value and policy heads. More details on the neural network architecture are in Section 2.4. We use 0.2 as the clip ratio, as well as early stopping: if the mean KL-divergence of the new policy from the old grows beyond a threshold, training is stopped for the policy, while it continues for the value function. We used a threshold of $1.5 \cdot K$, with $K = 1 \times 10^{-2}$. To reduce the variance, we used Generalized Advantage Estimation, as in [11], to estimate the advantage, with $\gamma = 1$ and $\lambda = 0.95$. Saved rewards R_t were normalized with respect to when they were collected (called rewards-to-go in [10]).

2.2. Modeling a PRNG as a MDP

In [6], we modeled a PRNG as a fully observable MDP in the following way. We chose a bit length for the sequence, say B. The state space is given by all possible bit sequences of length B:

$$\mathcal{S} := \{(b_1, b_2, \ldots, b_B) : b_n \in \{0, 1\}\}.$$

Action 1_n is the action of setting the nth bit to 1, and 0_n is the action of setting the nth bit to 0. The action space is then:

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

This finite MDP formulation has $|\mathcal{S}| = 2^B$ and $|\mathcal{A}| = 2B$, and was called the Binary Formulation (BF) in [6]. The main problem with BF is the fact that the size $2B$ of the action set grows linearly with the increasing length B of the sequence. Above a certain size, it is no longer possible to learn a policy with a high average score. If the size is big enough, no policy can be learned at all because it is almost impossible for an agent to explore an action space so huge within a reasonable time. Consider that for PRNGs, a sequence of 1000 bits is quite short, while for a RL problem, 2000 actions are way too much.

In this paper, we overcome this limitation of BF by hiding a portion of the full pseudo-random sequence, letting the agent see and act only on the last N bits. This removes the correlation between the final length of the sequence and the number of actions, at the cost of introducing a temporal dependency among states, breaking Markovianity and making the resulting MDP partially observable. This new problem is solved by approximating the hidden portion of the state with the hidden state of a recurrent neural network with memory—see Section 2.4 for details. We call this new approach Recurrent Formulation (RF).

Let $N \in \mathbb{N}$ be the number of bits at the end of the full sequence that we want to expose, or, in other words, let the observations space be the set $\mathcal{O} := \{0, 1\}^N$. We want the agent to be able to freely change the last N bits—that is, the action space $\mathcal{A}$ coincides with $\mathcal{O}$. We also fix a predetermined temporal horizon T for episodes. This means that the size of the action space is $|\mathcal{A}| = 2^N$ for a generated sequence of $T \cdot N$ bits. Both $|\mathcal{A}|$ and the length of S_T depend on N, but they do not depend on each other. Increasing the length of the generated sequence can then be done by increasing the horizon T, leaving the action space size constant. For example, with $N = 3$, the action and observation sets are:

$$\mathcal{O} = \mathcal{A} = \{[000]; [001]; [010]; [100]; [011]; [101]; [110]; [111]\}.$$

Continuing the example, assume at $t = 0$ we start from a random initial state $S_0 = O_0 = [001]$, and that $A_0 = [111]$. Now, the full state is $S_1 = [001111]$, but only the last three bits $O_1 = [111]$ are observed by the agent. If the agent now chooses $A_1 = [101]$, the full state becomes $S_2 = [001111101]$, and so on. If we set the episodes' length to 100, at the end of the episode the full state is a 300 bit sequence.

Clearly, this formulation can work only if the policy approximator $\pi(\cdot|\cdot; \theta)$ can preserve some information from the time series given by the past observations. Recurrent neural networks can model memory, and for this reason, are one of the possible approaches to process time series, and temporal

relationships among data in general. This is very useful in RL environments where, from the point of the view of the agent, the Markov property does not hold, as it is typically the case in many partially observable environments. This approach was suggested in [12]—see [13,14] for examples of cases far different from ours.

State-of-the-art recurrent neural networks for this kind of problem are the Gated Recurrent Unit (GRU) ones, and the Long-Short Term Memory (LSTM) ones. Typically, GRU performs better than LSTM, but for this particular formulation we have experienced good performance with LSTM. Thus, we use LSTM layers to approximate the policy network—see Section 2.4 for details on the neural network architecture.

2.3. Reward and NIST Test Suite

The NIST statistical test suite for random and pseudo-random number generators is the most popular application used to test the randomness of sequences of bits. It has been published as a result of a comprehensive theoretical and experimental analysis, and may be considered as the state-of-the-art in randomness testing for cryptographic and not cryptographic applications. The test suite has become a standard stage in assessing the outcome of PRNGs shortly after its publication.

The NIST test suite is based on statistical hypothesis testing and contains a set of statistical tests specially designed to assess different pseudo-random number sequence properties. Each test computes a test statistic value, a function of the input sequence. This value is then used to calculate a p-value that summarizes the strength of the evidence for the sequence to be random. For more details, see [1].

If S_t is the sequence produced by the agent at time t, the NIST test suite can be used to compute the average p-value of all eligible tests run on S_t. Some tests return multiple statistic values: in that case, their average is taken. If a test has failed its value for the average, it is set to zero. Some tests are not eligible on certain sequences which are too short, and in this case they are not considered for the average. This average $\text{avg}_{\text{NIST}}(S_t)$ is used at the end T of each episode as a reward function for the MDP:

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

This is the same reward strategy used in [6]. Note that, since p-values are probabilities, rewards belong to $[0, 1]$, and that NIST test suite accuracy grows with the tested sequence length.

2.4. Neural Network Architecture

PPO is an actor-critic algorithm. This means that it requires a neural network for the policy (the actor) and another neural network for the value function (the critic). Moreover, since RF is a partially observable MDP formulation, we need a way to maintain as much information as possible from previous observations, without exponentially increasing the size of the state space. We solve this problem with LSTM layers [15].

The neural network used for RF starts with two LSTM layers, with bias $= 1$ for the forget gate. After the LSTM layers, the network splits into two different subnetworks, one for the policy and one for the value function. The policy subnetwork has three stacked dense layers, with 256, 128, and 64 neurons, respectively. All of them have ReLU activation. After this, a dense layer with 2^N neurons provides preferences for the actions, which are turned into probabilities by a softmax activation. This is the policy head. The value function subnetwork starts exactly as the policy one (but with different weights): three dense layers with 256, 128, and 64 neurons stacked, respectively, all with ReLU activation. At the end, there is a dense layer with 1 neuron and no activation for the state value. This is the value head. All layers have Xavier initialization. For additional details, refer to the GitHub repository [7].

BF uses two different neural networks for the policy and the value function. The policy network has three dense layers with 256, 512, and 256 neurons stacked, respectively. All of them have ReLU

activation. After this, the policy head is the same as in RF: a dense layer with 2^N neurons and softmax activation. It is the same for the value function network: three dense layers with 256, 512, and 256 neurons stacked, respectively, with ReLU activation. After this, the value function head is the same as in RF: a dense layer with 1 neuron and no activation. All layers have Xavier initialization. For additional details, refer to the GitHub repository [7].

2.5. Framework

The framework used for the RL algorithms is USienaRL (Available on PyPi, and also on GitHub: https://github.com/InsaneMonster/USienaRL). This framework allows for environment, agent, and interface definition using a preset of configurable models. While agents and environments are direct implementations of what is described in the RL theory, interfaces are specific to this implementation. Under this framework, an interface is a system used to convert environment states to agent observations, and to encode agent actions into the environment. This allows to define agents operating on different spaces while keeping the same environment. By default, an interface is defined as pass-through, that is, a fully observable state where agents' actions have a direct effect on the environment. The two fashions of the BF are defined using different interfaces in the implementation. Specifically, the wanderer interface masks out all action, resulting in a bit being set to itself, while the baseline uses a simple pass-through interface.

The NIST test battery is run with another framework, called NistRng (Available on PyPi, and also on GitHub: https://github.com/InsaneMonster/NistRng). This framework allows us to easily run a customizable battery of a statistical set over a certain sequence. The framework also computes which tests are acceptable over certain sequences, such as due to their length. Acceptable tests for a certain sequence are called eligible tests. Each test returns a value and a flag stating whether the test was successfully exceeded or not by the sequence. If a test is not eligible with respect to a certain sequence, it cannot be run and it is skipped.

3. Results

Our experiments consist of multiple sets of training processes of various BF and RF agents. The goal of these experiments was to measure the performance of the new RF agents and compare it with the results achieved by BF agents. We consider three different agents, all trained by PPO-Clip described in Section 2.1, with an actor-critic neural network described in Section 2.4.
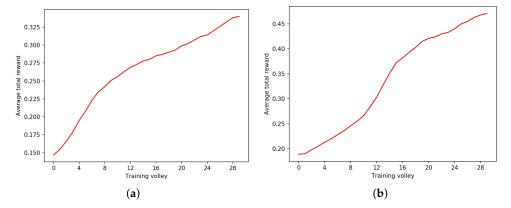
The agent based on the formulation RF introduced in this paper is called π_{RF}, and similarly, we denote by π_{BF} the agent based on BF, which we use as a baseline. We also introduce a third agent, a variation of π_{BF} denoted by $\hat{\pi}_{BF}$ and called "wanderer": this agent is forced to move as much as possible within the environment by masking out all actions that would keep the agent in the same state. In other words, at time-step t, the agent $\hat{\pi}_{BF}$ is forbidden from setting a certain bit to the same value it has at the previous time-step, $t-1$.

In the experiments, we optimize a PPO-Clip loss with GAE advantage estimation. The two separated policy and value losses are estimated by Adam with mini-batch samples of 32 experiences drawn randomly from a buffer. The buffer is filled by 500 episodes for π_{BF} and $\hat{\pi}_{BF}$, and by 1000 episodes for π_{RF}. Once the buffer is full, a training epoch is performed: thus, for instance, an agent π_{BF} building sequences of length $B = 200$ by episodes of length $T = 100$ will start training when the buffer is filled with $500 \times 100 = 50{,}000$ experiences, and the epoch will end after $[50{,}000/32] = 1562$ training steps. Learning rates are 3×10^{-4} for the policy and 1×10^{-3} for the value. At the end of the epoch, the buffer is emptied, and the RL pipeline goes on by experiencing new episodes.
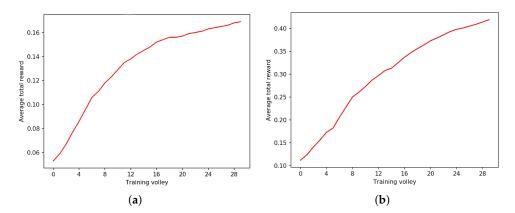
We present experimental results in the form of plots. Like in [6], the performance metric used is the average total reward across sets of epochs called *volley*. In this paper, a volley is made from two epochs—that is, the plots represent a moving average over a window of two epochs.

Figures 2a,b–4a,b describe three experiments with π_{BF} and $\hat{\pi}_{BF}$ over sequences of different lengths: $B = 80$ bits, $B = 200$ bits and $B = 400$ bits, respectively. The length of episodes is $T = 40$, $T = 100$ and $T = 200$, respectively.

Figure 2. Experiment on BF with $B = 80$. The learning curve is different and the average total reward is better with $\hat{\pi}_{BF}$. Volleys are composed of 1000 episodes each, and the fixed length of each trajectory is $T = 40$ steps. (**a**) π_{BF} with $B = 80$. (**b**) $\hat{\pi}_{BF}$ with $B = 80$.

Figure 3. Experiment on BF with $B = 200$. Despite the similar learning curve, there is a huge difference in the achieved average total reward per episode between $\hat{\pi}_{BF}$ and π_{BF} at the end of the training process. Volleys are composed of 1000 episodes each and the fixed length of each trajectory is $T = 100$ steps. (**a**) π_{BF} with $B = 200$. (**b**) $\hat{\pi}_{BF}$ with $B = 200$.
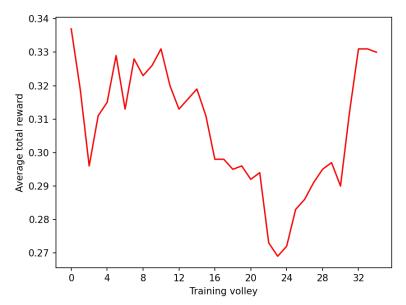
For very short sequences of 80 bits, $\hat{\pi}_{BF}$ has better performance at the end, yet very similar performance at the beginning of training. Making the sequence longer, $B = 200$ bits, allows the wanderer agent to perform much better than the baseline—starting from a slightly different performance at the beginning, $\hat{\pi}_{BF}$ has a much higher average total reward at the end of the training process. This trend is confirmed with sequences of $B = 400$ bits.

Figures 5–7 describe three experiments with π_{RF}. In this case, the length of the trajectory T directly influences the length of the sequence generated by π_{RF}. We keep $T = 100$ constant across π_{RF} experiments. This value is chosen experimentally as giving the best performance while keeping the inference time acceptable. To obtain sequences that are comparable with the ones generated by BF agents, we chose to append $N = 2$, $N = 5$ and $N = 10$ bits, respectively, resulting in final sequences of length $2 \times 100 = 200$ bits, $5 \times 100 = 500$ bits, and $10 \times 100 = 1000$ bits, respectively. The seed S_0 of the PRNG was drawn from a standard multivariate normal distribution of dimensions $N = 2$, $N = 5$, and $N = 10$, respectively.
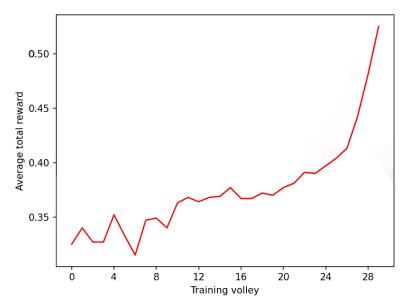
(**a**)

(**b**)

Figure 4. Experiment on BF with $B = 400$. The difference in the achieved average total reward per episode between $\hat{\pi}_{BF}$ and π_{BF} is similar to the case with $B = 200$, while the learning curve is different. Volleys are composed of 1000 episodes each, and the fixed length of each trajectory is $T = 200$ steps. (**a**) π_{BF} with $B = 400$. (**b**) $\hat{\pi}_{BF}$ with $B = 400$.
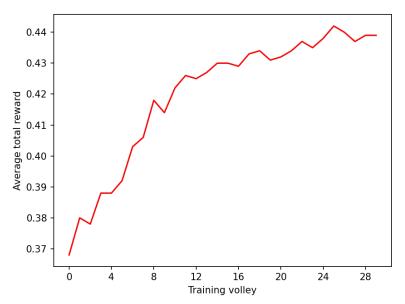

Figure 5. Average total rewards during training of π_{RF} with $N = 2$ and $T = 100$. Volleys are composed of 2000 episodes each.

For $N = 2$, training was not successful. We do not have a clear explanation for this fact. A possible reason is that we were trying to represent the non-observable portion of the state with a very low-dimensional input. However, this is not completely true, because in theory, the inputs from every time-step were preserved in the hidden state of the LSTM. This issue could be related to the difficulties that recurrent neural networks have shown with gradient descent optimizers—see [16].

For $N = 5$, that is, sequences of 500 bits, π_{RF} vastly outperforms π_{BF}, and albeit by a narrower margin, $\hat{\pi}_{BF}$ with sequences of 200 bits. Thus, we had better performance with 150% additional bits in the sequence. For $N = 10$, that is, sequences of 1000 bits, π_{RF} vastly outperforms π_{BF} with $B = 400$. The wanderer agent in this case performs similarly, but π_{RF} still produces sequences that are 150% longer than the ones produced by $\hat{\pi}_{BF}$.
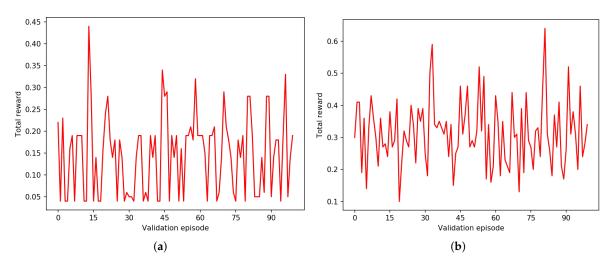
Figure 6. Average total rewards during training of π_{RF} with $N = 5$ and $T = 100$ steps. Volleys are composed of 2000 episodes each.
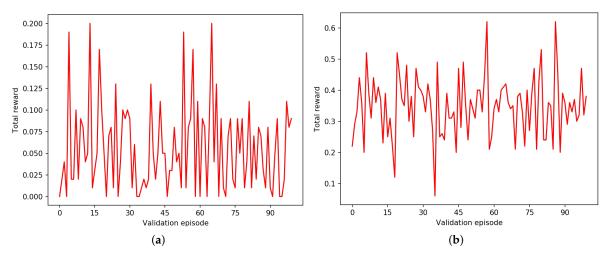

Figure 7. Average total rewards during training of π_{RF} with $N = 10$ and $T = 100$ steps. Volleys are composed of 2000 episodes each.

We now want to test the hypothesis that, in order to generate PRNGs by DRL, modeling the MDP as in RF is much better than modeling it as in BF. To this aim, we compare the average total rewards per episode of random agents operating in RF and in the baseline BF. We call these agents ρ_{RF} and ρ_{BF}, respectively. This comparison is performed to exclude the possibility that PPO is performing better with RF, but maybe different algorithms would perform better with BF. Using a random agent means removing the algorithm from the equation.
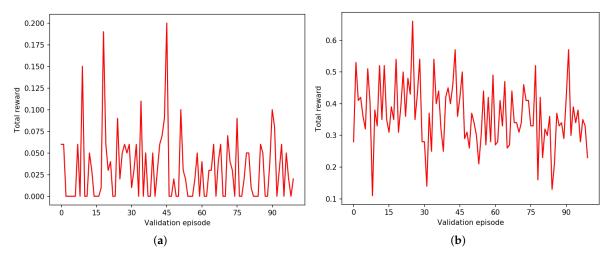
Figures 8a,b–10a,b describe three comparisons between the random agents producing short, medium, and long sequences, respectively. As before, short means 80 bits in BF and 200 bits in RF, medium means 200 bits in BF and 500 bits in RF, and long means 400 bits in BF and 1000 bits in RF. In each comparison, the average total rewards of ρ_{RF} is higher than ρ_{BF}. From these results, we can assess that RF is a better formulation than BF for the PRNG task.

Figure 8. Average total rewards of a random agent on BF and RF for short sequences. (**a**) ρ_{BF} with $B = 80$, episodes length $T = 40$. (**b**) ρ_{RF} with $N = 2$, episodes length $T = 100$.
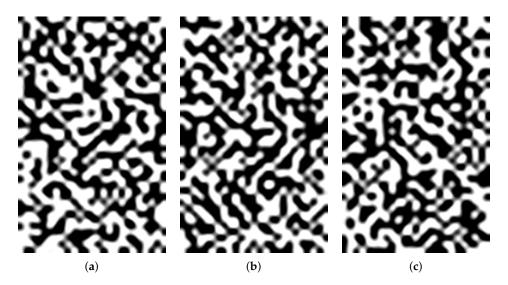

Figure 9. Average total rewards of a random agent on BF and RF for medium sequences. (**a**) ρ_{BF} with $B = 200$, episodes length $T = 100$. (**b**) ρ_{RF} with $N = 5$, episodes length $T = 100$.

Figure 10. Average total rewards of a random agent on BF and RF for long sequences. (**a**) ρ_{BF} with $B = 400$, episodes length $T = 200$. (**b**) ρ_{RF} with $N = 10$, episodes length $T = 100$.

Finally, Figure 11a–c graphically represents three sequences of 1000 bits generated by the same agent π_{RF} after training, with NIST scores of 0.4, 0.43, and 0.53, respectively. The 1000 bits were stacked on 40 rows and 25 columns, then ones were converted to 10×10 white squares, and zeros to 10×10 black squares. The resulting image was smoothed.

| (**a**) | (**b**) | (**c**) |

Figure 11. A graphical representation of three sequences of 1000 bits generated by the same trained π_{RF} with (**a**) NIST score 0.4, (**b**) NIST score 0.43, and (**c**) NIST score 0.53. Images were obtained by stacking the 1000 bits in 40 rows and 25 columns, then ones were converted to 10×10 white squares and zeros to 10×10 black squares. The resulting image was smoothed.

4. Discussion

In this paper, we introduced a novel, partially observable MDP modeling the task of generating PRNGs from scratch using DRL, denoted by RF for recurrent formulation. RF improves our previous MDP modeling in [6] because it makes the action space size independent of the length of the generated sequence. Experiments show that RF agents trained with PPO simultaneously obtains a higher average NIST score and longer sequences, thus improving [6] in two different ways. We used a PPO instance with a hidden state of an LSTM to encode significant features of the non-observed portion of the sequence—as far as we know, this is an original idea for PRNG. Experiments with a random agent show that RF is a better form of MDP modeling when compared with the binary formulation BF of [6]—that is, when RF is compared with BF, it obtains a higher average NIST score with longer sequences. All this means that RF scales better to PRNG with longer periods.

However, while RF is a significant improvement over BF, the action space size grows as 2^N, where N is the bit-length of the appended sequence. Since DRL does not scale well to discrete and large action sets (see, for instance, [17]), this is a limitation for RF. In our experiments, we have found that $N > 10$ is not feasible for RF. Moreover, the vanishing gradient is an obstacle to increasing the episodes' length T with recurrent neural networks, as shown in [16]. In our experiments, we have been unable to train RF with $T = 200$, so we consider $T = 100$ an upper bound for RF with PPO and the LSTM architecture described in Section 2.4. Since the PRNG period is $T \cdot N$, we can say that RF, with all the hyperparameters described in this paper, does not scale well over 1000 bits. Another problem of this approach is the sparse reward, which in general makes it difficult for a RL agent to be trained. The above remarks takes to the following list of possible future improvements/research topics:

- Devise a different, less sparse, reward function.
- Devise an algorithm using RF modified with a continuous action space. In this way, we can append sequences of any length without increasing the size of $\mathcal{A}$. Some preliminary experiments show

that this approach could work, but at the time of writing, it still presents some performance issues, and is a work in progress.

- RF could be used to intelligently stack generated periods. For example, one could train a policy to stack sequences generated by one or multiple PRNGs, even with different sequence lengths. This would generate PRNGs with very long periods without a sensible drop in quality. This approach could use a mixture of state-of-the-art PRNGs, DRL generators like the ones seen in this paper, and so on. A challenge of this approach is how to measure the change in the NIST score when appending a random sequence to another.
- The upper bound given by the episodes' length T could be overcome, or at least mitigated, by other architectures capable of maintaining memory for a time longer than recurrent neural networks, like attention models.

Comparison with Relevant Research

As far as we know, our previous paper [6] with binary formulation BF is the first DRL application to make an automatic generation of PRNGs. A comparison of RF with BF is given in Sections 2.2 and 3, where it is shown that RF agents simultaneously obtain a higher average NIST score and longer sequences. Moreover, experiments with a random agent show that RF is a better MDP modeling of this problem, with respect to BF.

Abbreviations

The following abbreviations are used in this manuscript:

PRNG	Pseudo-Random Number Generator
MDP	Markov Decision Process
NIST	National Institute of Standards and Technology
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
LSTM	Long-Short Term Memory
GRU	Gated Recurrent Unit
PG	Policy Gradient
PPO	Proximal Policy Optimization
BF	Binary Formulation
RF	Recurrent Formulation
ReLU	Rectified Linear Unit
GAE	Generalized Advantage Estimation

References

1. Bassham, L.E., III; 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. *Sp 800-22 rev. 1a. a Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*; National Institute of Standards & Technology: Gaithersburg, MD, USA, 2010.
2. Desai, V.; Patil, R.; Rao, D. Using layer recurrent neural network to generate pseudo random number sequences. *Int. J. Comput. Sci. Issues* **2012**, *9*, 324–334.
3. Hughes, J.M. Pseudo-random Number Generation Using Binary Recurrent Neural Networks. Ph.D. Thesis, Kalamazoo College, Kalamazoo, MI, USA, 2007.

4. Abdi, H. A neural network primer. *J. Biol. Syst.* **1994**, *2*, 247–281. [CrossRef]

5. De Bernardi, M.; Khouzani, M.; Malacaria, P. Pseudo-Random Number Generation Using Generative Adversarial Networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 191–200.

6. Pasqualini, L.; Parton, M. Pseudo Random Number Generation: A Reinforcement Learning approach. *Procedia Comput. Sci.* **2020**, *170*, 1122–1127. [CrossRef]

7. Pasqualini, L. Pseudo Random Number Generation through Reinforcement Learning and Recurrent Neural Networks. GitHub Repository. 2020. Available online: https://github.com/InsaneMonster/pasqualini2020prngrl (accessed on 23 November 2020).

8. Sutton, R.; Barto, A. *Reinforcement Learning: An Introduction*; Adaptive Computation and Machine Learning Series; MIT Press: Cambridge, MA, USA, 2018.

9. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv* **2017**, arXiv:1707.06347.

10. OpenAI. Proximal Policy Optimization. OpenAI Web Site. 2018. Available online: https://spinningup.openai.com/en/latest/algorithms/ppo.html (accessed on 23 November 2020).

11. Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv* **2015**, arXiv:1506.02438.

12. Duell, S.; Udluft, S.; Sterzing, V. Solving partially observable reinforcement learning problems with recurrent neural networks. In *Neural Networks: Tricks of the Trade*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 709–733.

13. Wang, L.; Zhang, W.; He, X.; Zha, H. Supervised Reinforcement Learning with Recurrent Neural Network for Dynamic Treatment Recommendation. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18), New York, NY, USA, 19–23 August 2018; pp. 2447–2456. [CrossRef]

14. Chakraborty, S. Capturing Financial markets to apply Deep Reinforcement Learning. *arXiv* **2019**, arXiv:1907.04373.

15. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef]

16. Bengio, Y.; Simard, P.; Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* **1994**, *5*, 157–166. [CrossRef] [PubMed]

17. Zahavy, T.; Haroush, M.; Merlis, N.; Mankowitz, D.J.; Mannor, S. Learn what not to learn: Action elimination with deep reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2018**, 3562–3573. Available online: https://proceedings.neurips.cc/paper/2018/hash/645098b086d2f9e1e0e939c27f9f2d6f-Abstract.html (accessed on 23 November 2020).

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.