DERANGE: DICE-ENHANCED RANDOM NUMBER GENERATOR

Ravi Hassanaly
Sorbonne Université
Paris

Nemo Fournier
Sorbonne Université
Paris

Ghislain Vaillant
Paris

April 1, 2024

ABSTRACT

Upon review of the literature, it appears clear that Pseudo Random Number Generation (PRNG) used extensively in the Machine Learning is intrinsically problematic. We propose here to reintroduce *True* Random Number Generation in the ML field, and we publish a library which allows users to replace the default PRNG provided in PYTHON by the results of dice rolls that have been performed by the authors in very controlled conditions. This will ensure more sound theoretical foundations of any downstream ML algorithm using our source of randomness.

Keywords Dice · Random Number Generation · Paradigm Shift

1 Introduction

1.1 The role of Random Number Generation in Computing, and focus on Machine Learning

Rare are the ramifications of modern computing that do not rely on Random Number Generation (RNG). Cryptography of course comes to mind, of which many algorithms are centered around making encrypted data as close as possible to random bits of information. This while enforcing reversibility and determinism of the operation to reach its practical purpose (*i.e.* original data can be recovered from the seemingly random sequence of bits). One could even argue that practical cryptography is nothing but the science of pseudo-random number generation.

Yet, many other applications of RNG are commonly encountered. One is the study of computational statistics and machine learning algorithms. Algorithms developed in these fields often boil down to estimating and sampling parameters or data from a probability distribution in which relevant information for the problem of interest is embedded. Since such distributions are usually not practically observed nor computationally tractable, algorithms rely on stochastic estimations of these parameters, by treating data as a random realization of an underlying distribution and performing stochastic computations and estimations to estimate — hopefully likelihood-maximizing — model parameters. Random Number Generation therefore plays a significant part in how such algorithms behave. Many recent advances in generative machine learning, such as Diffusion Models [1] rely on the progressive rectification of a noise field (hence generated by RNG) towards data that looks as though it was sampled from a source distribution. Other machine learning algorithms also heavily rely on RNG at their core. One can for instance mention Genetic Algorithms [2], in which random mutations are introduced and further selected according to the *environmental pressure* enforced to improve performance in the learning task.

All of these intertwinements of computing and RNG mean that a source of RNG must be available in the implementation platform of many of those algorithms[3].

1.2 Pseudo-RNG

Because of the intrinsic deterministic nature of computers, randomness cannot be created out of sequential computing steps. Either an external physical device must be used as a source of randomness through either chaotic systems (we for instance quote the Lavarand device [4] or the double-pendulum based RNG [5]) or truly random processes, such as Quantum-physics based RNG [6]. Such external devices, even though providing solid grounds for RNG, are not the most practical and often space-consuming. Most machine learning laboratories would therefore have to choose between using an external devices and hosting interns during intern season.

Thankfully, RNG on determinstic machines can still be approached, thanks to the development of Pseudo-RNG (PRNG) algorithms. We refer to [7] for a recent survey about such PRNG algorithms. In a nutshell, these are based upon the use of an initial entropy source (which is often stated as the *seed* of the PRNG system), which is then transformed through a sequence of deterministic operations. The most common PRNG are the Linear Congruence Generators (LCG) [8], defined by a set of (a, c, m) integers through which an iterative sequence of numbers (x_n) is produced as the following recurrence relation, of which the initial x_0 is the seed.

$$x_{n+1} = (a \cdot x_n + c) \bmod m$$

The study of such PRNG (and especially LCG) has been extensive over the past decades, trying to define and ensure desirable properties of these generators (such as *uniformity*, *independance*, *large period*, *reproducibility*, *consistency*, *disjoint subsequentiality*, *portability*, *efficiency*, *coverage*, *spectral characteristc* and *cryptographic security* — we refer here again to [7] for more comprehensive definitions). Despite their apparent simplicity, LCG have been prone to very serious *ill-designs* even among very widespread implementations. Let us mention the RANDU LCG (characterized by the $\left(65539, 0, 2^{31}\right)$ triplet), which has been famously pointed as one of the worst generators ever used in production. Its bad properties can famously be observed when sampling the unit cubes (all samples can be found to lie in at most 15 planes) and has been used as a "negative ground-truth" for randomness testing suite since then [9].

1.3 Fundamental Limits of PRNG and Derandomization.

Many problems, such as the ELECTIONOFALEADER problem (in which a system of identical distributed nodes need to collectively agree on a *leader*) can be shown to be insolvable if relying on classical pseudo RNG [10]. This is because determinism (which encompasses PRNG) does not allow for symmetry breaking in the network of nodes. Only algorithm that can access to true source of randomness can solve this problem in an exact manner [6]. More fundamentally, the use of random-number generators is of first order importance in the theory of computation. Indeed many decision and computation problems have been shown to belong to the BPP (*Bounded-error Probabilistic Polynomial* time) complexity class [11]. One historically notable example is the PRIMALITYTESTING decision problem, which admits a very simple BPP solution through the MILLER-RABIN algorithm [12]. Proving that algorithms of the BPP class behave equivalently when using a pseudo-RNG as a source of "randomness" would be sufficient to collapse the BPP class into P and thus a major step in mapping computational complexity. But so far this equivalence remains out of reach, justifying the need for true sources of randomness in the framework of random algorithms.

These limitations shows that it's quite a miracle that modern computing still holds on PRNG while their properties and equivalence to truly random RNG is out of the reach of the contemporary mind.

1.4 Limitations of PRNG use in Machine Learning.

As mentioned earlier, Machine Learning relies on RNG at its core in many aspects. Yet contrary to the cryptography field, the impact of the PRNG choice has not been studied extensively. Following the above-mentioned limits of PRNG, and embracing the popular sayings that "an ounce of prevention is worth a pound of cure" as well as the "better safe than sorry", our stance is that it is ill-advised to continue using PRNG in machine learning.

1.5 Dice: An Underrated source of Randomness

Dice are ancient analog mechanisms used to generate random numbers [13]. Their main advantage is that they are small, easy to use, and considered truly random. Their use for RNG is further detailled in the Methods section of this article.

1.6 Our Proposal: Bringing Back True Randomization using Dice Throws

We propose to tackle the issue of potentially bad PRNG properties by leveraging physical random process for number generation, in a manner that would be convenient for the overall community. We offer to perform and report the results of three thousands throws of 10 faces dice. These results are available in a PYTHON library that once imported replace the underlying source of randomness by a reading of the results of our throws. For retro-compatibility, we still offer the possibility to use a seed with this source of randomness, which influences the read-out order of the random number table. For convenience and overall improved learning performances, an interface to perform *seed-tuning* of ML experiments is also provided. This paper describes the methodological choices and methods used to generate these numbers, as well as some additional results for the occasional dice-throwing interested reader.

2 Methods

In order to bring true randomness in computer science, we use a true random process in order to sample a large number of numbers. At the genesis of this project, we had the issue of choosing a random value for a parameter of a deep learning model, in the process of running a random search. A computer scientist would simply use the Python `random`[1] package at the beginning of the script to initialize the model. However, at that particular moment, we had a deck of card on our desk, and decided to draw cards in order to select a value for this parameter. This later gave us the idea to sample random parameters using role play dices, that are numbered from 0 to 9 (it is the kind of dice that is used in Dungeons & Dragons). We bought three 10-sided dices with three different colors. And then, when needing to set a parameter between 0 and 1000, we just throw the three dices in order to have a value (with each color corresponding to a power of 10). This process was very satisfying as it was fun, but also truly random. As we strongly believe that true randomness is essential for many purposes, but most importantly for science, we initiated this project in order to share our work with the computer science community.

To this end, we developed a package that allows to randomly sample numbers between 0 and 1000 with true randomness. We made 1000 throws of three 10-sided dices, so 3000 throws in total, and reported the result in a CSV file. This file can then be read by the user in order to sample a number randomly generated. Variability can be added to the sampling: the user can choose a seed that simply correspond to a different way of reading the table. To make it more practical, we packaged this in a Python library, allowing to directly import the sampler in Python and replace the fake default random sampler.

3 Experimental setting

It may be surprising to have an "Experimental setting" here, as the experiment is very simple. Indeed, it only consists in throwing dices and reporting numbers. However, as it is a tedious and repetitive task, we optimized the process and would like to share the tricks with you, in case you would like to generate your own list of random number.

The experiment will be realized with two operators: one will throw the dices, and read the result loudly, while the other one will write it. Then, we strongly advise to directly report the results on a numeric spreadsheet that can be saved and exported as a CSV file, rather than a blackboard that can be erased, or a paper notebook that can burn. It would be really unfortunate to lose such valuable data.

The main material is three 10-sided dices: one brown, one purple and one orange. We used a table in a room, as we believe that outside perturbation such as wind and rain could impact the results. The table needs to be large enough to put three boxes on it. To optimize the reading of the results, even if the dices were of different colors, we throw each dice in its own box, each box corresponding to a power of 10. It is indeed quite difficult and prone to errors to throw the three dices at one time. It is trivial to prove that throwing the three dices separately in strictly equivalent in terms of probabilities than throwing the three of them all at once. Moreover, the use of boxes reduces the risk of a die falling off the table.

The protocol can be resumed as follows:

- operator A throws the brown dice in the left box and tells to operator B what number have been drawn,
- operator B notes the number on its spread shit,
- operator A throws the purple dice in the middle box and tells to operator B what number have been drawn,
- operator B notes the number on its spread shit,
- operator A throws the orange dice in the right box and tells to operator B what number have been drawn,
- operator B notes the number on its spread shit,
- operator B presses the key (usually `enter`) to go to the next line.

[1] `https://docs.python.org/3/library/random.html`

Table 1: Dices' mass

Dice	Mass (in g)
Brown	data point lost while preparing the paper
Purple	data point lost while preparing the paper
Orange	data point lost while preparing the paper

Table 2: Boxes' dimension

Box	Length (in cm)	Width (in cm)	Height (in cm)
1	29.5	29.9	7.0
2	29.8	29.9	6.8
3	30.	30.3	8.1

This is then repeated 1000 time. The operations of operator A are illustrated in Figure 1. To avoid boredom, operator A and B inverse their role every 100 iterations. In addition, a coffee break is suggested and the half of the experiment.

Figure 1: The tasks of operator A when the brown die gives a 5, the purple one a 2 and the orange one a 7.

In order to improve the reproducibility of the experiment, we report in Table 1 the exact mass of the three dices that we used, and in Table 2 the dimensions of the three boxes. Unfortunately, we had written down the exact mass (measured to the 10^{-4} grams) in a yellow notebook that we have since lost, please feel free to contact the corresponding author if you run into it. We also tracked the temperature of the room over the time in order to enable any scientist that would like to reproduce our results to reproduce the same environment. We also believe that the atmospheric pressure in the room is a key component for the reproducibility of the experiment, but unfortunately it was very difficult to lend a device to measure it, so we cannot report this data.

In total, including the pause, it took exactly 1 hour and 59 minutes to complete the experiment. We estimate that using this protocol, if the operators are focused, it requires 1 minutes to sample 10 numbers, which corresponds to 30 dice rolls. If they are awakened, it may require only 10 second to sample 10 number, but none of the operators could achieve this state, despite their high level of experience.

4 Results

4.1 Statistical analysis

As curious math enjoyers, having a list of 1000 randomly sampled numbers gives us the desire to carry out a statistical study.

In total, 633 number out of 1000 have been drawn. In other words, 367 numbers have not been drawn. We unfortunately did not draw any fixed point (meaning that the throw N give us the number N). The most obtained numbers are 394, 575, 911 and 828 with 5 occurrences for each of them. In Figure 2 we display the density of number obtained using a count plot, to check if it is similar to the uniform distribution and make sure that no pattern emerges. To prove it mathematically and show that our method surpasses the PRNG available in Python, we measure the Wasserstein distance between our list of value and the uniform distribution and compared it to the Wasserstein distance between a pseudo-random list generate with `random.randint` and the uniform distribution. We obtain respectively Wasserstein distances of 10.68 and 11.09, proving that the proposed list is closer to a uniform law than the Python method (fun fact: we did not even have to generate several lists for this result).

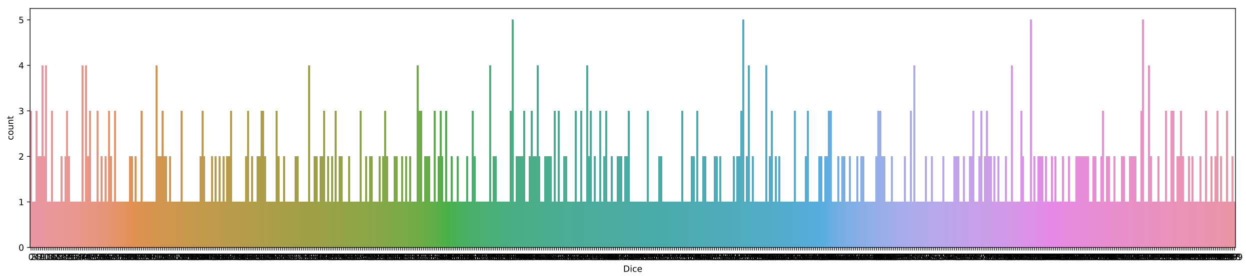

Figure 2: Distribution of numbers obtained from the 1000 dice rolls.

We illustrate in Figure 3 the different number over dice rolls. It would be a nice experiment to use it as an audio signal and check if it sounds like a white noise in order to confirm the random nature of the experiment.

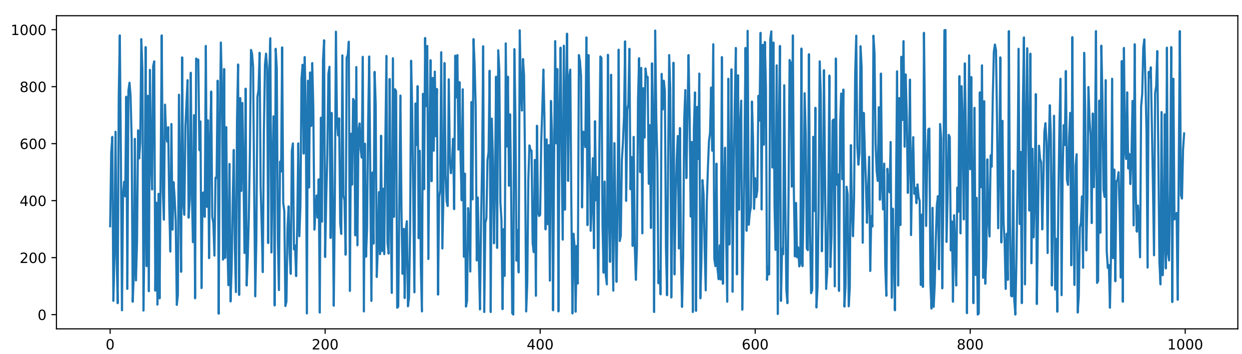

Figure 3: Numbers obtained over the 1000 dice rolls (in chronological order).

When throwing the dice, we realized that some pattern were much more exciting to obtain than the others, such as hundreds or triples. We counted a total of 15 hundreds, which is way above the expectancy (that is 10), and 11 triples, which is just above the expectancy. Can we consider our experiment as a lucky experiment? Actually not really. Indeed, in Figure 4, we show the occurrences of each number for the three dice. We can see that the 0 is the most obtained number with 349 occurrences, explaining why the hundreds are more probable (as the condition for a hundred is that the 2nd and 3rd dice give a 0). If we consider that 3000 throws are enough to conclude, 0 is more probable than the other numbers. This can be critical, especially when playing a role game, where a 0 outcome will probably lead you to a defeat. Another interesting point is that the 7, usually considered in many cultures as a "lucky number", is the number with the lowest frequency, with only 0.085% of appearance. Therefore, we would not recommend playing the 7 when betting with these dices, especially if you gamble money.

In Figure 5, we separated the 3 dices. There is nothing much to add using this plot, except that the color code follows the colors of the dice.

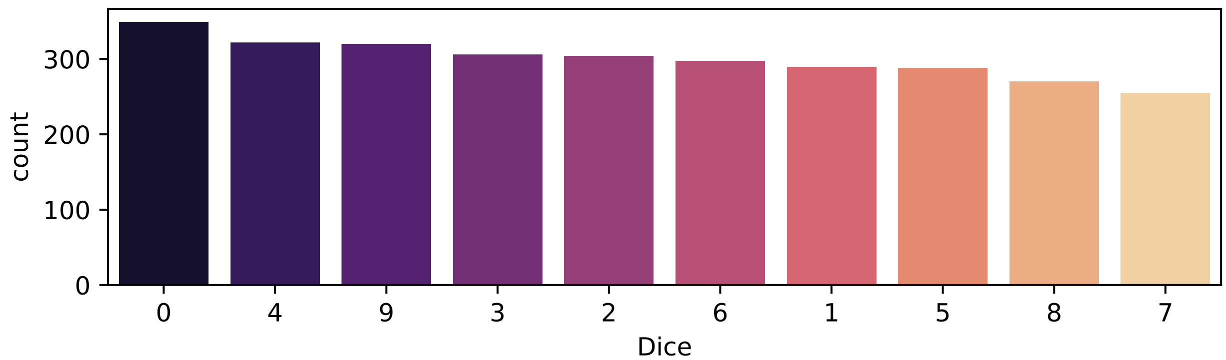

Figure 4: Distribution of digits obtained from the 3000 dice rolls of the three dice used (summed over the dice).

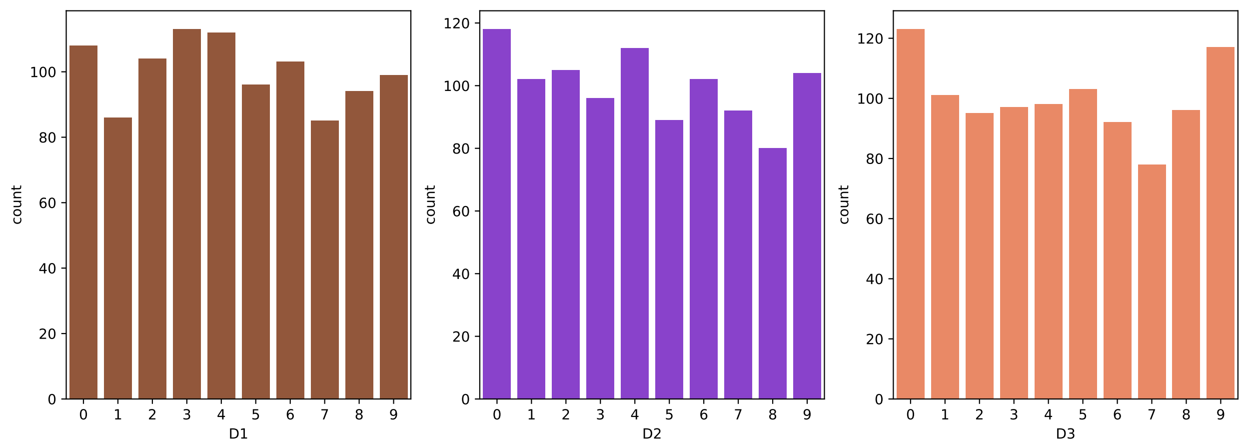

Figure 5: Distributions of digits obtained from the 1000 dice rolls for each of the die used.

We also had the feeling that many numbers had a double inside (for instance 101, 110 or 001 are numbers with double). We checked and in total we have 295 numbers with doubles, where the expectancy is 270.

It is interesting to notice that no number outside the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ have been reported, probably meaning that the operator did not try to make us believe that the dice are magical.

Finally, we report in Figure 7, the evolution of the temperature during the experiment. We can observe that the temperature increase as the number of dice rolls augment. However, even is there is a clear correlation, it is not enough to conclude that throwing dices causes the augmentation of the agitation of the gas molecule in the atmosphere of the room. The most probable explanation for this temperature increase is the effect of the sun warming the earth's atmosphere, or maybe due to global warming.

To end this statistical analysis, we present an interesting 3D(ice) visualization of the number generated in Figure **??**.

4.2 Table

The final table of generated numbers in available in Appendix A.

5 Discussion and future direction

As a result of all this intense work, we built a Python package, the link will be shared soon. Currently, the code and CSV file can be found on the following repository: `https://github.com/ravih18/DERaNGe`.

We are currently planning on an online platform in which everybody that throws a die can report the result of the throw. We will manually check that each contributed number is random enough and add it to the next release of the library.

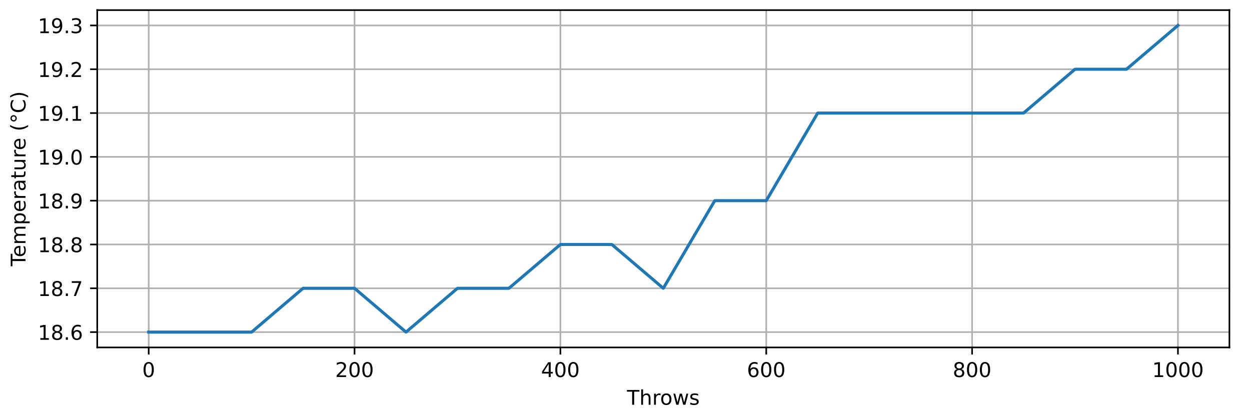

Figure 6: Evolution of the temperature during the experiment.

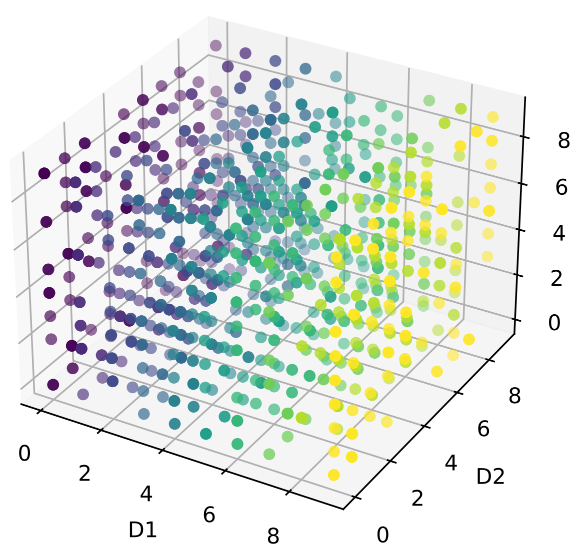

Figure 7: 3D scatter plot of the obtained numbers, with each axis representing one die.

Anonymity of contributions will be ensured using dice-based cryptographic schemes, allowing any user willing to participate in this project to do so without any consequence other than overall better science.

6 Conclusion

To wrap-up our work, our investigation has provided compelling evidence to support the use of traditional dice-rolling as a viable method for Random Number Generation (RNG) in machine learning experiments. The stochastic nature of dice outcomes demonstrated a level of randomness comparable to contemporary computational RNG algorithms. Thus, it appears that this age-old practice may offer a feasible alternative for researchers seeking reliable RNG sources for their experimental designs in Machine Learning. In light of these findings, further studies are warranted to explore the full potential and applicability of dice-rolling as a legitimate RNG methodology in scientific research and policy making in general.

Acknowledgments

The research leading to these results has received the help of:

- Maëlys Solal that generously lent us the dices.
- Fanny Namysl that helped us to use the high precision weighing machine.
- Agnes Rastetter that provided us an accurate thermometer.
- Gaia Gentile for her kind supervision during the dice rolls.

Ethics

The operators (that are also the authors) were fully cooperative and agreed to spend 2 hours of their Saturday morning to throw 1000 times three dices. They have rewarded themselves with a cup of coffee and a biscuit.

References

[1] Florinel-Alin Croitoru, Vlad Hondru, Radu Tudor Ionescu, and Mubarak Shah. Diffusion Models in Vision: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(9):10850–10869, September 2023. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[2] Kenneth De Jong. Learning with genetic algorithms: An overview. *Machine Learning*, 3(2):121–138, October 1988.

[3] Benjamin Antunes and David R. C. Hill. Reproducibility, energy efficiency and performance of pseudorandom number generators in machine learning: a comparative study of python, numpy, tensorflow, and pytorch implementations, February 2024. arXiv:2401.17345 [cs].

[4] Landon Curt Noll, Robert G. Mende, and Sanjeev Sisodiya. Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system, March 1998.

[5] Chokri Nouar and Zine El Abidine Guennoun. A Pseudo-Random Number Generator Using Double Pendulum. *Applied Mathematics & Information Sciences*, 14(6):977–984, November 2020.

[6] Marcin M. Jacak, Piotr Jóźwiak, Jakub Niemczuk, and Janusz E. Jacak. Quantum generators of random numbers. *Scientific Reports*, 11(1):16108, August 2021. Publisher: Nature Publishing Group.

[7] Kamalika Bhattacharjee and Sukanta Das. A search for good pseudo-random number generators: Survey and empirical studies. *Computer Science Review*, 45:100471, August 2022.

[8] GEORGE Marsaglia. The Structure of Linear Congruential Sequences. In S. K. Zaremba, editor, *Applications of Number Theory to Numerical Analysis*, pages 249–285. Academic Press, January 1972.

[9] George S. Fishman and Louis R. Moore. A Statistical Evaluation of Multiplicative Congruential Random Number Generators with Modulus 231 — 1. *Journal of the American Statistical Association*, 77(377):129–136, March 1982. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/01621459.1982.10477775.

[10] Dana Angluin. Local and global properties in networks of processors (Extended Abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 82–93, New York, NY, USA, April 1980. Association for Computing Machinery.

[11] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, August 1995.

[12] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, February 1980.

[13] Alex de Voogt. The role of the dice in board games history. January 2015.

A Table of dice based random number generator

310	569	624	48	202	642	220	40	764	980	553	15	422	466	415	765	90	786	814	768	618
45	140	617	120	255	647	548	604	967	861	14	296	939	170	768	82	859	554	439	872	889
84	393	35	424	57	488	980	431	333	737	640	607	658	378	221	669	298	465	410	327	34
70	772	459	150	903	393	350	659	739	823	340	402	849	432	254	700	57	899	893	895	577
678	93	427	429	343	943	378	682	198	612	783	343	321	207	481	479	821	3	664	955	783
193	862	199	658	257	103	529	46	160	395	578	232	79	682	878	69	760	434	743	500	216
793	102	199	345	570	929	916	868	422	64	265	764	788	919	446	308	149	638	877	916	866
251	848	970	218	520	696	32	933	864	265	86	537	502	938	393	363	30	46	304	380	213
143	575	601	229	243	135	309	601	275	382	828	877	565	904	711	4	821	446	850	662	883
723	298	367	418	340	475	7	691	326	892	963	202	407	561	844	870	269	708	411	31	396
993	683	629	689	321	283	910	418	400	210	902	910	958	83	635	456	757	749	278	869	243
644	671	579	551	905	11	600	203	303	682	906	456	48	499	249	852	740	132	237	430	212
628	223	442	212	611	908	253	214	827	344	76	900	343	791	784	24	28	389	769	368	143
301	58	297	328	29	59	290	891	711	390	78	741	595	802	268	575	171	11	775	440	971
732	943	195	616	893	468	831	575	853	626	792	70	411	436	921	232	414	883	491	397	381
826	545	496	539	617	370	909	762	911	579	816	544	401	791	203	495	28	49	374	323	151
746	404	967	880	733	190	411	190	18	321	528	942	9	322	338	542	571	856	9	688	555
250	463	835	234	928	855	394	332	19	300	136	952	590	835	452	703	436	5	0	932	684
189	297	148	998	802	716	897	840	510	11	109	474	594	578	575	251	219	543	66	663	359
346	350	609	724	860	686	299	615	317	595	119	750	636	15	569	960	601	862	11	710	937
540	263	944	368	775	986	469	832	860	394	4	284	240	10	241	109	911	890	837	376	642
640	588	973	314	911	565	294	394	549	233	679	401	483	70	556	906	901	311	147	467	140
106	912	348	185	842	326	84	602	366	115	396	930	258	277	542	605	695	959	399	719	738
933	440	553	241	624	240	122	236	752	900	500	866	285	795	621	864	832	834	459	665	304
882	683	9	997	690	538	110	153	71	845	720	821	784	237	68	532	911	794	60	377	884
141	407	542	627	232	655	205	27	202	685	788	479	690	911	704	344	606	9	652	780	13
729	546	846	57	191	472	422	230	85	394	512	608	638	797	575	949	196	170	530	160	124

243	123	904	108	458	586	279	45	828	246	416	861	80	503	739	936	141	840	368	661	751
17	253	560	936	294	996	17	253	560	936	294	996	315	356	378	748	617	371	479	412	436
768	681	989	369	947	596	957	795	122	169	141	967	994	516	955	380	227	875	2	652	943
48	231	212	805	394	87	40	409	894	883	449	980	426	204	392	201	236	169	181	934	170
381	777	630	232	226	914	302	75	86	624	364	726	25	96	365	889	775	222	400	858	474
90	150	397	839	167	275	681	685	99	899	456	492	83	668	776	475	790	28	100	449	423
29	94	595	379	275	411	844	979	591	526	567	942	869	252	708	571	465	322	440	554	153
346	309	979	916	600	504	469	728	403	846	454	369	530	167	66	512	449	429	606	59	372
189	15	446	854	102	760	314	905	683	960	589	843	697	427	538	825	279	527	623	424	445
391	457	412	399	104	351	97	989	673	311	531	650	653	111	21	375	27	103	261	384	416
159	669	620	92	401	998	999	255	419	631	623	225	327	45	349	238	102	446	361	837	285
802	674	334	882	625	5	557	910	510	834	425	40	726	139	410	0	4	780	447	875	129
749	108	201	545	334	274	559	519	763	922	948	928	671	303	517	903	253	680	328	257	90
284	122	995	323	66	64	505	51	0	193	460	933	318	571	40	731	973	106	655	935	619
365	916	180	582	271	420	774	37	298	593	547	535	299	451	642	672	604	216	579	735	631
102	378	698	88	266	10	404	661	828	68	497	665	595	855	473	455	622	708	173	974	325
104	521	563	7	65	306	322	415	164	919	711	225	409	799	684	630	321	706	447	570	995
111	118	751	287	944	765	438	413	823	223	164	174	24	213	828	198	509	117	468	472	500
357	130	890	45	936	586	546	780	513	564	458	556	750	313	949	424	291	383	323	201	731
774	937	966	820	640	165	852	833	868	660	452	208	777	802	925	539	186	106	711	138	249
703	162	937	215	190	459	939	44	828	334	346	357	52								

B Easter (l)egg of lamb

As today (the 1st of April 2024) is also Easter, here is the Chat-GPT recipe for the Easter leg of Lamb:
Ingredients:

- 1 leg of lamb (about 5-7 pounds)
- 4 cloves of garlic, minced
- 2 tablespoons of fresh rosemary, chopped
- 2 tablespoons of fresh thyme, chopped
- 1 tablespoon of fresh parsley, chopped
- 1/4 cup of olive oil

- Juice of 1 lemon
- Salt and pepper to taste

Instructions:

Preheat your oven to 350°F (175°C).

In a small bowl, mix together the minced garlic, chopped rosemary, thyme, parsley, olive oil, lemon juice, salt, and pepper to form a marinade.

Place the leg of lamb in a roasting pan and use a sharp knife to make several small incisions all over the surface of the lamb.

Rub the marinade all over the lamb, making sure to work it into the incisions you made.

Cover the roasting pan with foil and let the lamb marinate in the refrigerator for at least 2 hours, or overnight for best results.

Once the lamb has finished marinating, remove it from the refrigerator and let it come to room temperature for about 30 minutes.

Remove the foil from the roasting pan and place the lamb in the preheated oven.

Roast the lamb for about 20 minutes per pound, or until it reaches your desired level of doneness. For medium-rare, aim for an internal temperature of 135-140°F (57-60°C) when measured with a meat thermometer inserted into the thickest part of the meat.

Once the lamb is cooked to your liking, remove it from the oven and let it rest for at least 15 minutes before carving.

Carve the lamb into slices and serve with your favorite side dishes, such as roasted vegetables, mashed potatoes, or a fresh salad.