The Impact of Nondeterminism on Reproducibility in Deep Reinforcement Learning

Prabhat Nagarajan
Department of Computer Science
The University of Texas at Austin
prabhatn@cs.utexas.edu

Garrett Warnell
Computational and Information Sciences Directorate
U.S. Army Research Laboratory
garrett.a.warnell.civ@mail.mil

Peter Stone
Department of Computer Science
The University of Texas at Austin
pstone@cs.utexas.edu

Abstract

While deep reinforcement learning (DRL) has enjoyed several recent successes, results reported in the literature are often difficult to reliably reproduce. Difficulties in reproducibility can arise due to many factors, including the lack of access to computational resources or the lack of knowledge of specific implementation details. Another factor of particular importance in the specific case of DRL is the ability to control for sources of nondeterminism during the training process. This is because DRL is faced with the challenges of a nonstationary training distribution and additional sources of randomness that are absent from other areas of machine learning. In this paper, we (1) enable deterministic training in DRL by identifying and controlling for all sources of nondeterminism present during training, and (2) perform an ablation study that shows how these sources of nondeterminism can impact the performance of a DRL agent. We find that even simple sources of nondeterminism such as those stemming from nondeterministic GPU operations can lead to large differences in performance between training runs. Lastly, we make available our deterministic implementation of deep Q-learning [14].

1 Introduction

Progress in DRL relies on the reproducibility of state-of-the-art algorithms. In particular, reproducible algorithms enable researchers to more easily improve upon and measure against state-of-the-art research. However, reproducing DRL algorithms can be quite difficult due to several factors including inconsistent evaluation methodologies [10, 12], computational limitations, and perhaps most commonly, the lack of knowledge of the implementation details necessary to achieve a performance similar to a published result. Details such as the weight initialization scheme, network architectures, learning rates, minibatch sizes, and other hyperparameters are often imperative for success, yet go unreported in published work.

However, even with explicit knowledge of the implementation details, reproducing results *exactly* remains a challenge. This issue highlights the important distinction between *reproducibility* and *replicability* in machine learning [5]. While this distinction has been addressed in previous literature and has been defined in different ways [5, 6, 8], we define these terms here as follows:

Reproducibility: the ability of an experiment to be repeated with minor differences from the original experiment, while achieving the same qualitative result.

2nd Reproducibility in Machine Learning Workshop at ICML 2018, Stockholm, Sweden.

Replicability: a stricter form of reproducibility in which the results are reproduced exactly, achieving the same quantitative result.

Given a replicable experiment, researchers need not squander time acquiring the tribal knowledge necessary to reproduce results and can instead focus on the underlying techniques behind algorithms. Achieving replicability requires a *deterministic implementation* to be repeated under identical *experimental conditions*. *Experimental conditions* refer to the software and hardware conditions under which an experiment is executed, and we define a *deterministic implementation* as follows:

Deterministic implementation: a computer program that, when run under some fixed experimental conditions, will always produce identical outputs for a given input.

It is important to emphasize that, as we have defined the terms here, having a deterministic implementation does not necessarily imply replicability. That said, deterministic implementations do have several benefits relevant to the reproducibility community. For example, comparing two algorithms under deterministic conditions provides an impartial comparison of their performances since nondeterminism can be eliminated as a cause of difference. Furthermore, debugging is made easier because the determinism allows the state of the program to be more efficiently tracked, which enables researchers to develop (or reproduce) algorithms more quickly.

We believe that deterministic implementations can be particularly useful in DRL. To understand why, consider that DRL is more susceptible to nondeterminism than supervised deep learning (SDL). Traditional SDL encounters nondeterminism primarily through random network initialization and minibatch sampling from a stationary training distribution. On the other hand, DRL must contend with additional sources of randomness such as environment and policy stochasticity. Moreover, DRL also faces the challenge of learning from a nonstationary distribution of training data since the policy continuously evolves, changing the distribution of states that the agent visits. Imagine how a small difference between two agents' early experiences can proliferate throughout the training process as their experience distributions drift apart. The difference in experiences can result in drastically different outcomes for the two agents. It is this cascading effect that makes DRL particularly susceptible to nondeterminism. Furthermore, a recent empirical result shows that in a specific implementation of a policy gradient DRL algorithm, it is possible to achieve statistically different performances solely by altering the global random seed [10]. The cascading effect coupled with this compelling empirical result motivates a deeper study of the impact of nondeterminism in DRL.

Motivated by these considerations, we propose in this paper a methodology for achieving a deterministic implementation of a popular DRL algorithm, deep Q-learning [14]. Our methodology consists of eliminating all sources of nondeterminism from the training process. In order to quantify the benefit of a deterministic implementation, we study the *impact that individual sources of nondeterminism have on the performance of DRL agents*. In particular, we study three sources of nondeterminism: nondeterminism originating from the GPU, from random exploration, and from random initialization. We find that all three sources induce large variance in the learning curves. We also find that variance is minimal early in the training process and grows large as training continues. In this work, our specific contributions are:

1. to identify sources of nondeterminism that, when controlled, give rise to deterministic deep Q-learning, and
2. to perform a systematic study of the effects of specific sources of nondeterminism.

In doing so, we demonstrate that our deterministic implementation, which we make publicly available, can be particularly beneficial for reproducibility in DRL.

2 Related Work

While reproducibility has been explored across artificial intelligence, machine learning, and robotics [5, 8, 3, 7], reproducibility in DRL remains relatively uncharted. In the context of DRL, the effects of hyperparameters, codebases, evaluation metrics, random seeds, and aspects of the environment have been studied to a degree [10, 11, 12]. However, existing work studies the aggregate effect of random seeds, whereas here we investigate individual sources of randomness. Further, our work studies value-based methods whereas previous research in reproducibility for DRL has focused on

policy gradient methods. Finally, previous work done in the context of DRL has focused primarily on the broader notion of reproducibility from Section 1, whereas we emphasize determinism, which is more closely related to replicability.

While not branded explicitly under reproducibility, other works have analyzed the impact of hyperparameters and randomness in DRL. For instance, it was shown [4] that the frame skip hyperparameter commonly used in DRL algorithms can strongly influence a learning agent's success in the Arcade Learning Environment (ALE) [2], a standard evaluation platform for DRL agents. The ALE was originally designed to be deterministic, allowing agents to succeed by simply memorizing action sequences. Consequently, several methods of injecting stochasticity into the environment [9, 12] have been proposed to address this issue. These methods of injecting stochasticity into the environment have been evaluated for their ability to cause memorizing agents to fail while simultaneously not harming the performance of non-memorizing agents. It should be noted that these studies of environment stochasticity are performed in the presence of other forms of nondeterminism, and do not isolate sources of nondeterminism as we do. Many of these findings have been synthesized into best practices for the DRL community [10, 12], with the hope of setting the standard for research to follow.

3 Background

We now provide a brief background of the Markov decision process formulation of reinforcement learning problems and of the deep Q-learning algorithm, which is our algorithm of interest in this paper.

3.1 Markov Decision Processes

Reinforcement learning (RL) problems are formulated in the context of *Markov decision processes* (MDPs). An MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, \gamma, \mathcal{R})$, where $\mathcal{S}$ denotes the set of *states* within the environment and $\mathcal{A}$ denotes the set of actions available to the agent within the environment. The agent acts at discrete timesteps where, at each timestep, it experiences a state, performs an action, and transitions to another state. This is formalized by the *transition model* P, where $P(s'|s, a)$ is the probability that the agent transitions to state s' when performing action a in state s. The *discount rate* $\gamma \in [0, 1]$ specifies the agent's preference for immediate rewards versus future rewards. The *reward function* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ provides the agent with reward $\mathcal{R}(s, a, s')$ as it transitions to state s' after performing action a in state s.

Given an MDP, an RL agent's objective is to learn a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ mapping a state-action pair (s, a) to the probability that the agent performs action a in state s. Specifically, the agent tries to learn an *optimal policy* π^*, a policy that maximizes the agent's expected cumulative discounted reward $\mathbb{E}[\mathcal{R}(s_0, a_0, s_1) + \gamma \cdot \mathcal{R}(s_1, a_1, s_2) + ... + \gamma^t \cdot \mathcal{R}(s_t, a_t, s_{t+1}) + ...]$. Oftentimes, rather than directly learning an optimal policy π^*, the agent learns the *optimal state-action value function* $Q^* : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which maps a state-action pair (s, a) to the expected cumulative discounted reward the agent receives if action a is taken in state s and optimal actions are performed thereafter. If the agent learns Q^*, then an optimal policy can be to perform action $\underset{a}{\operatorname{argmax}}\, Q^*(s, a)$ in state s.

3.2 Deep Q-learning

Deep Q-learning [13, 14] uses a deep neural network to approximate the state-action value function Q and trains the network with Q-learning [18], an algorithm for learning Q^*. Deep Q-learning can achieve human-level performance in many Atari games in the ALE while learning directly from pixel representations of the state. As the agent interacts with its environment, it maintains a *replay buffer* $\mathcal{D}$ of its last N transitions (typically $N = 1$ million). Each entry in this replay buffer contains a tuple (s_t, a_t, r_t, s_{t+1}), representing the state, action, reward, and subsequent state, respectively. The network representing the state-action value function being learned is termed a deep Q-network (DQN), where $Q(s, a; \boldsymbol{\theta})$ represents the predicted state-action value under the DQN parameters $\boldsymbol{\theta}$. The algorithm also maintains a separate *target network* $Q(s, a; \boldsymbol{\theta}^-)$, where $\boldsymbol{\theta}^-$ represents the parameters of a network from a prior training iteration. Periodically, the target network parameters are reset to equal the DQN parameters: $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$. To train the DQN at iteration i, the agent minimizes the loss [14]:

$$L_i(\boldsymbol{\theta}_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}_i^-) - Q(s, a; \boldsymbol{\theta}_i)\right)^2\right]. \quad (1)$$

Following techniques from stochastic optimization, the agent randomly samples minibatches of transitions uniformly from the replay buffer, uses the DQN and the target network to compute the loss, and then updates the DQN's weights.

4 Nondeterminism in Deep Reinforcement Learning

In order to investigate the challenge that nondeterminism poses for reproducibility, we must identify all the sources of nondeterminism that are present when implementing deep Q-learning. Once identified, controlling or eliminating all of these sources from the learning process is sufficient for obtaining a deterministic implementation of deep Q-learning.

4.1 Sources of Nondeterminism

While the exact sources of nondeterminism depend on the algorithm, problem domain, libraries, etc., we identify those sources common to most DRL algorithm implementations.

- **GPU** Neural networks are typically trained on graphics processings units (GPUs). However, under certain experimental conditions, numerical operations carried out on the GPU yield nondeterministic outcomes.
- **Environment** The environment in reinforcement learning can be stochastic. That is, the transitions can be random.
- **Policy** During training, reinforcement learning agents typically employ a stochastic policy. That is, the agent's action is drawn from a non-degenerate distribution over the available actions.
- **Network initialization** Prior to training, the weights of the neural network are randomly initialized.
- **Minibatch sampling** When training neural networks, several algorithms sample random minibatches of training data from some dataset.

In deep Q-learning, the neural network is typically trained using a GPU. Consequently, deep Q-learning has nondeterminism stemming from both the network initialization and the GPU. The algorithm also has randomness in the form of minibatch sampling of transitions from the replay buffer during training. Furthermore, during training, the agent uses an ϵ-greedy policy, performing a random action with probability ϵ at each timestep. The ALE was originally deterministic, and while a new version of the ALE [12] permits stochastic environments, we use the deterministic environment as in the original deep Q-learning papers [13, 14]. Furthermore, while the ALE exposes a random seed for the environment, the seed takes no effect in most games, yielding deterministic dynamics given a fixed action sequence. For completeness, however, we fix the random seed throughout our experiments. Lastly, deep Q-learning introduces an additional source of randomness not listed above, in the form of *no-op* (or "do nothing") actions. The agent performs a random number of no-op actions at the beginning of each episode in order to randomize the initial state within the deterministic environment.

4.2 Implementation: Eliminating Nondeterminism

Our implementation of Deep Q-learning is written in Python using the PyTorch library [16]. PyTorch[1] exposes a modifiable boolean variable that allows us to enable or disable deterministic numerical computations on the GPU. Furthermore, PyTorch permits us to set the seed used to initialize the

[1] We selected PyTorch for its ease of controlling GPU nondeterminism. However, the sources of nondeterminism can depend on the deep learning library used. For example, as of this writing, Tensorflow has certain nondeterministic functions that require workarounds and enabling GPU determinism is not straightforward. In fact, we were not able to do so.

weights of the deep Q-network, allowing us to obtain identical networks on separate runs. To control for no-ops, exploration, and minibatch sampling, we assign each of these sources of nondeterminism its own seeded random number generator. Any random operations required for no-ops, exploration, or minibatch sampling is implemented using the assigned random number generator. Thus, across training runs, the same "random" number of no-ops are performed at the beginning of episodes. Exploratory actions are identical and occur at consistent timesteps across runs. Similarly, the same minibatches are sampled across runs.

When all these sources of nondeterminism are held fixed, as well as the implementation details and experimental conditions (as is the case in our experiments), we achieve identical results on separate training runs, as desired. To validate the equivalence of separate runs, we verify that the learned weights of the neural network are identical at intervals throughout training.

The network architecture for our DQNs is two hidden convolutional layers, a hidden fully connected layer, and a fully connected output layer with an output for each action [13]. The agent's policy during training is an ϵ-greedy policy, where at each timestep, it either performs a random action with probability ϵ, or the greedy action $\operatorname{argmax}_a Q^*(s, a; \boldsymbol{\theta})$ with probability $1 - \epsilon$. The value ϵ is initially set to 1.0 and is linearly annealed to 0.1 over the first million frames, remaining at 0.1 thereafter. We make our deterministic implementation[2] available with all hyperparameters, implementation details, and experimental conditions recorded.

5 Experiments

To quantify the benefit of controlling nondeterminism, we use our deterministic implementation as a baseline and systematically allow individual sources of randomness to influence the training process. We train multiple networks in the presence of individual sources of nondeterminism and evaluate their performance under two different evaluation metrics: the game score and the maximum predicted Q-values on a set of states. We use the standard deviations of the evaluations as a qualitative measure of the impact of a source of nondeterminism on achieving reproducibility. The sources of nondeterminism that we focus on are: GPU nondeterminism, exploration randomness, and random weight initializations.

5.1 Training

In our experiments, we train four groups of networks using our deterministic implementation. For each group, we train five networks, as is often done in DRL papers [12, 15]. The first group, which we call the "deterministic" group, consists of five networks trained with the same random seeds and with deterministic GPU operations enabled across all runs. The "initialization" group consists of five networks each trained with a different set of randomly initialized weights and all other settings identical to the deterministic group. The "exploration" group consists of five networks trained with different random exploration seeds and all other settings identical to the deterministic group. The final group is the "GPU" group, which has all settings identical to the deterministic group except with deterministic GPU operations disabled. All of our agents were trained for 10 million timesteps in the ALE [2] on the Atari game BREAKOUT, a domain where the agent uses a paddle to hit a moving ball while trying to eliminate rows of bricks from the game screen. The agents are evaluated every 100K timesteps of training, and we compute the mean and standard deviation of the agent's performance after each of these evaluation intervals. The hardware and software conditions are held constant for all experiments.

5.2 Evaluation Protocol: Average Score

Typically, evaluation protocols [13, 14] for the (deterministic) ALE involve averaging the agent's game score over several episodes during which the agent performs an ϵ-greedy policy with some low ϵ. This exploration is done in order to produce diversity in the episode trajectories which tests the agent's ability to generalize to different states. However, suppose we are evaluating two different agents. If we have the customary random exploration during the evaluation phase, then after a single deviation between the policies, the agents could be in different states, and the seeded random

[2] `https://github.com/prabhatnagarajan/repro_dqn.git`

exploration would not have the same effect on the agents' trajectories, confounding the results. Thus, in alignment with our goal of measuring differences in performance that stem solely from agents' policies, we utilize a greedy policy during evaluations. Consequently, any variation in the agents' performance can be attributed to differences between their Q-networks alone. However, since the ALE is deterministic, each greedy evaluation results in the same episode, which poses an issue. We must ensure that our evaluation tests the agent's ability to generalize to different states, because a single episode is not representative of the agent's overall performance.

Our solution to the above issue is to evaluate each agent for 100 episodes, where each episode is capped at five minutes of play. Each of these episodes begins with a predetermined action sequence, followed by the agent's greedy policy for the remainder of the episode. This idea is similar to using "human starts" at the beginning of episodes, where the agent completes an episode starting from a trajectory of human expert play [15]. We obtained our start sequences by generating 1000 random sequences, each with a random number (chosen uniformly between 55 and 95) of random actions (performed without frame skipping). However, since the sequences were generated randomly, some of these sequences may end in poor states. To address this, we used a trained DQN to evaluate the Q-values for these states, and selected 100 start sequences randomly from those sequences that ended in states with reasonable Q-values. While this method of generating start sequences is biased towards those sequences that a DQN might perform well in, it enables us to generate longer sequences of random actions, which improves the diversity of our start states. These same 100 start sequences were used at each evaluation interval for all training runs. In this way, we are able to have a diverse set of start states that remain constant across all evaluations, while still ensuring that differences in performance are caused by the agents' policies alone.

5.3 Evaluation Protocol: Average Maximum Q-Values

Because the above protocol yields noisy learning curves (discussed further in Section 5.4.1), our second evaluation protocol uses a more stable metric that is commonly used [1, 13, 14, 17], based off of the agent's estimated action-value function Q. In a fashion similar to prior work [13], we run a random policy for 50K timesteps, and sample 500 states from the 50K initial states. Denoting the set of 500 held out states as $\mathcal{S}_Q$, we compute the average maximum predicted Q-value for each state over that set, i.e.,

$$\frac{1}{|\mathcal{S}_Q|} \sum_{s \in \mathcal{S}_Q} \max_a Q(s, a).$$

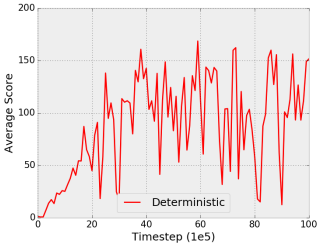
Plotting the average maximum Q-value shows smoother learning not seen in curves that plot the average score.

5.4 Results

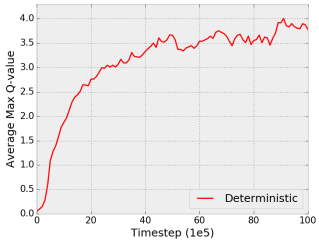
Our results are summarized in Figure 1. For each experimental group and evaluation metric, we plot the mean and population standard deviation of the metric. We discuss the results for each group individually and provide additional analysis.

5.4.1 Determinism

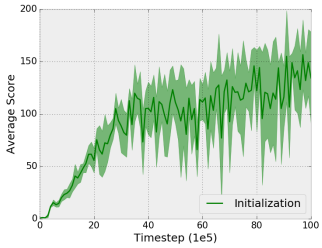
Figure 1(a) depicts the mean and standard deviation of the average scores from five deterministic runs with all sources of nondeterminism controlled. The extreme volatility of the curve, with rising and falling performance, is consistent with prior results [13]. This volatility is attributed to minor changes in the policy's weights that change the agent's state distribution. In fact, when we inspected the learning curve for each individual start sequence, we found that the score fluctuates wildly — far more than in Figure 1(a). Thus, even for individual start sequences, we do not observe a stable improvement in performance, which is consistent with the agent's state distribution changing in tandem with minor changes to the policy. If the agent's policy is only equipped to perform well in a subset of the state space, then that provides an explanation for the lack of stable learning for individual start sequences. Each individual start sequence exists in a different part of the state space. If the set of states for which an agent performs well on is constantly changing, then it is unlikely that there is a start sequence for which there is consistent improvement. Figure 1(b) shows the more stable learning of the agent as its estimated Q-values steadily increase throughout training.



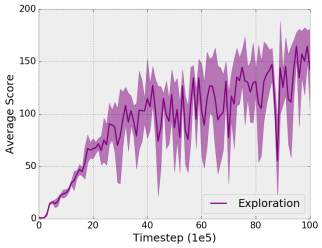
(a) Deterministic: score



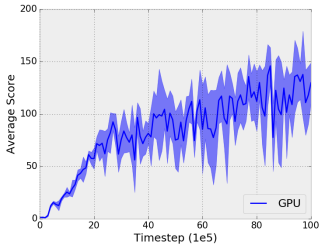
(b) Deterministic: $\max_a Q$



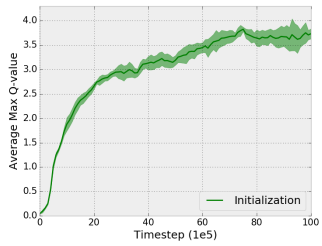
(c) Initialization: score



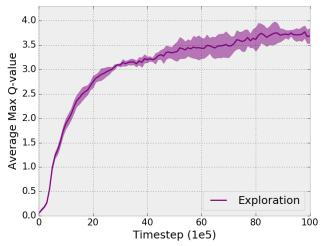
(d) Exploration: score



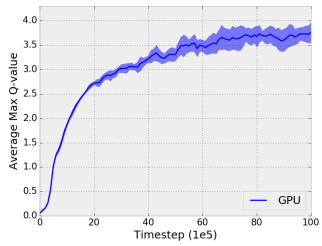
(e) GPU: score



(f) Initialization: $\max_a Q$



(g) Exploration: $\max_a Q$



(h) GPU: $\max_a Q$

Figure 1: The learning curves for four groups of agents trained with our deterministic implementation. A single group was trained deterministically with identical settings, while the remaining three groups permitted a single source of nondeterminism. Each curve depicts either the mean score or the mean maximum Q-value achieved from five different agent evaluations. The shaded area represents values within one population standard deviation of the mean. The absence of a shaded area in the deterministic curves indicates that the results are identical across the five runs.

The key point from Figures 1(a) and 1(b), however, is that both graphs lack any shaded area. That is, there is no variance in the evaluations, confirming that each of the five deterministic training runs produces identical evaluations.

5.4.2 Random Initialization

The results for the initialization group are shown in Figure 1(c) and Figure 1(f). Random initialization is a unique source of nondeterminism in that it is the only source in which the networks in the group do not have identical policies to begin with. However, recall from Section 4.2 that the agents perform a large amount of exploration early in training. Since the agents in this group have identical random exploration seeds, they have roughly the same early experiences from which to learn. However, given that their Q-networks start off differently, these common experiences affect their respective policies differently. As a result, we anticipated the variance to be nontrivial even in the early stages of training. However, in Figure 1(c), we observe that the variance is quite low at the beginning of training.

Figure 1(f) offers a possible explanation for this low initial variance. The Q-values provide a finer-grained view of the network's predictions than the game score. We find that, even under different initializations, the predicted Q-values are similar early in training. Since these networks predict similar Q-values, it is not unreasonable to expect them to have similar policies in the early stages of learning, resulting in similar performances. Another possible explanation is that their policies are quite different, but perform similarly in terms of game score. This may be the case because the set of policies that perform poorly is less restrictive than the set of policies that perform well. That is, in BREAKOUT there are more policies that can achieve a score of 0 than there are policies that can score 150. Consequently, different random initializations may cause different policies to perform similarly poorly during the early stages of training. The differences between these policies may only be reflected in the game score in the later stages of training when the networks improve and individual differences between policies have larger impacts on performance.

Random initialization is a source of nondeterminism that is fairly trivial to control. However, when uncontrolled, it can lead to large variations in performance, as shown in Figures 1(c) and 1(f).

5.4.3 Exploration

The results for the exploration group are shown in Figure 1(d) and Figure 1(g). We expected that altering exploration seeds would have the largest effect on performance. At the beginning of training, the agents perform large amounts of exploration, and different exploration seeds cause the experience distributions to diverge from the beginning, yielding different policies. Despite the Q-networks having identical weights at the beginning of training, we felt the differing experiences would substantially impact the policy. Indeed, Figure 1(d) exhibits the two key characteristics also shown in the curves for weight initialization: the low variance early in training followed by larger variance later in training. Again, these results illustrate the benefit of a deterministic implementation in that even with identical initial networks, uncontrolled exploration can cause remarkably different results.

5.4.4 GPU

The results for the GPU group are shown in Figure 1(e) and Figure 1(h). Initially, all networks in the GPU group have identical starting conditions. If the GPU performs operations deterministically, then the agents will have identical results throughout training. Given that the starting conditions are identical, we expect the GPU computations to differ only slightly across runs. While we still expect to observe a cascading effect from these small differences, we anticipated that it would be minimal. Since the agents share early experiences and have identical initial policies, we would expect the learned policies to be similar, again resulting in similar performance. However, Figure 1(e) shows that this is not the case. While the variance in game score is low for the first two million frames, it quickly increases and the variance in the learning curve looks similar to the curves for exploration and weight initialization. A similar effect is observed in Figure 1(h), where the variance is near zero for the first two million frames, but then subsequently grows. This experiment best demonstrates the cascading effect. All agents have identical starting conditions, but small deviations during training compound and lead to substantial differences as training progresses.

The GPU is of special interest since it is the only source of nondeterminism that exists *outside* of the algorithm itself. As a consequence, the GPU nondeterminism best underscores the benefit of deterministic implementations. Even if implementation details are successfully reproduced, sources of nondeterminism outside of the algorithm can considerably impact results.

6 Conclusion

In this paper, we quantified the benefit of a deterministic implementation by studying the impact of individual sources of nondeterminism. For all sources of nondeterminism studied, we found that the variance in performance is low early in training and grows larger as training progresses. Our core result is that, even with a fixed implementation run under identical experimental conditions using a standard style of evaluation, a single source of nondeterminism can substantially impact results. This illustrates the benefit of deterministic implementations. Under the general notion of reproducibility, a typical nondeterministic implementation may not reproduce results of similar quality to published results, solely due to the large variance between runs. We hope that our results spur the DRL community to provide deterministic implementations of algorithms when possible.

Acknowledgments

The authors would like to thank Darshan Thaker, Ewin Tang, Naren Manoj, Kapil Krishnakumar, and Brahma Pavse for their useful comments, suggestions, and reviews of earlier drafts.

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (IIS-1637736, IIS-1651089, IIS-1724157), Intel, Raytheon, and Lockheed Martin. Peter Stone serves on the Board of Directors of Cogitai, Inc. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

References

[1] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-DQN: Variance reduction and stabilization for deep reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 176–185, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[3] Fabio Bonsignorio. A new kind of article for reproducible research in intelligent robotics [from the field]. *IEEE Robotics & Automation Magazine*, 24(3):178–182, 2017.

[4] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. *Space*, 1600:1800, 2000.

[5] Chris Drummond. Replicability is not reproducibility: nor is it good science. 2009.

[6] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. What does research reproducibility mean? *Science translational medicine*, 8(341):341ps12–341ps12, 2016.

[7] Eugenio Guglielmelli. Research reproducibility and performance evaluation for dependable robots [from the editor's desk]. *IEEE Robotics & Automation Magazine*, 22(3):4–4, 2015.

[8] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. 2017.

[9] Matthew J Hausknecht and Peter Stone. The impact of determinism on learning atari 2600 games. In *AAAI Workshop: Learning for General Competency in Video Games*, 2015.

[10] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *AAAI*, pages 3207–3214, 2018.

[11] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

[12] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[15] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[17] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[18] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.