The Thesis Committee for Prabhat Mahadevan Nagarajan
certifies that this is the approved version of the following thesis:

Nondeterminism as a Reproducibility Challenge for Deep Reinforcement Learning

Committee:

Peter Stone, Supervisor

Scott Niekum

Nondeterminism as a Reproducibility Challenge for Deep Reinforcement Learning

by

Prabhat Mahadevan Nagarajan, B.S. Mathematics

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

August 2018

Acknowledgments

First and foremost I would like to thank my advisor, Peter Stone, for his guidance and encouragement throughout this thesis. He always made himself available to provide help and I could always count on his lightning-fast email responses. He gave me autonomy in finding a research topic and provided the right amount of guidance to help me make progress when I felt stuck. It has been a privilege to work with him.

I am also grateful to Scott Niekum for serving on my thesis committee and for his helpful comments. My many discussions with Scott (along with his excellent courses) have played an important role in shaping my interests within artificial intelligence.

Garrett Warnell deserves a special thanks as a collaborator and mentor throughout my research. He has provided invaluable help, especially with regards to his extensive reviews of my writing. Discussions with him have helped me flesh out ideas fully, and my writing and research ability have improved through his mentorship.

I would like to thank several of my peers and colleagues for the help they've provided during my thesis. Darshan Thaker has reviewed several drafts of my work and we have had several fruitful discussions about my research. Brahma Pavse has also reviewed several drafts of my work and has provided extensive feedback. I have had many long discussions with Ewin Tang, which have been very helpful for

formalizing aspects of my research. Naren Manoj and Kapil Krishnakumar have provided useful feedback on prior versions of the work in this thesis. I would also like to thank Josh Kelle for first implementing deep Q-networks with me and for helping me build some of the experimental infrastructure that has been critical for this thesis. I would also like to thank Amy Bush, as her assistance made it possible for me to run the many experiments needed for this thesis.

Prabhat Mahadevan Nagarajan

The University of Texas at Austin
August 2018

Nondeterminism as a Reproducibility Challenge for Deep Reinforcement Learning

Prabhat Mahadevan Nagarajan, M.S.
The University of Texas at Austin, 2018

Supervisor: Peter Stone

In recent years, deep neural networks have powered many successes in deep reinforcement learning (DRL) and artificial intelligence by serving as effective function approximators in high-dimensional domains. However, there are several difficulties in reproducing such successes. These difficulties have risen due to several factors, including researchers' limited access to compute power and a general lack of knowledge of implementation details that are critical for reproducing results successfully. However, nondeterminism is a reproducibility challenge that is perhaps less emphasized despite being particularly relevant in DRL. DRL algorithms tend to have high variance, in no small part due to the fact that agents must learn from a nonstationary training distribution in the presence of additional sources of randomness that are absent from other machine learning paradigms. The high variance of DRL algorithms, combined with the low sample sizes used in research, makes it difficult to

match reported results. As such, the ability to control for sources of nondeterminism is especially important for achieving reproducibility in DRL. If we are to maximize progress in DRL, we need research to be reproducible and verifiable, ensuring the validity of our claims. Reproducibility is a necessary prerequisite for improving upon or comparing algorithms, both of which are done frequently in DRL research.

In this thesis, we take steps towards studying the impact of nondeterminism on two important pillars of DRL research: the reproducibility of results and the statistical comparison of algorithms. We do so by (1) enabling deterministic training in DRL by identifying and controlling for all sources of nondeterminism present during training, (2) performing a sensitivity analysis that shows how these sources of nondeterminism can impact a DRL agent's performance and policy, and (3) showing how nondeterminism negatively impacts algorithm comparison in DRL and describing how deterministic training can mitigate this negative impact. We find that individual sources of nondeterminism such as the random network initialization can affect an agent's performance substantially. We also find that the current sample sizes used in DRL may not satisfactorily capture differences in performance between two algorithms. Lastly, we make available our deterministic implementation of deep Q-learning.

Contents

Acknowledgments **iv**

Abstract **vi**

List of Figures **xi**

List of Algorithms **xiii**

Chapter 1 Introduction **1**

1.1 Motivation 3

1.1.1 The Benefits of Determinism 5

1.1.2 Nondeterminism in Deep RL 6

1.2 Research Questions and Contributions 8

1.2.1 Research Questions 8

1.2.2 Contributions 9

1.3 Thesis Outline 10

Chapter 2 Background **11**

2.1 Neural Network Learning 11

2.1.1 Artificial Neurons and Neural Networks 12

2.1.2 Convolutional Neural Networks 14

2.2 Reinforcement Learning 16

2.2.1 Markov Decision Processes 18
2.2.2 Value Functions and the Bellman Equations 20
2.2.3 Q-learning 22
2.2.4 Q-learning with Function Approximation 23
2.3 The Arcade Learning Environment 24
2.4 Deep Reinforcement Learning 26
2.4.1 Deep Q-learning 26
2.4.2 Other Deep Reinforcment Learning Algorithms 30
2.5 Statistical Formulation of Deep Reinforcement Learning Algorithms 32

Chapter 3 Nondeterminism and Reproducibility in Deep Reinforcement Learning 34

3.1 Nondeterminism in Deep Reinforcement Learning 35
3.1.1 Sources of Nondeterminism 35
3.1.2 Implementation: Eliminating Nondeterminism 37
3.1.3 Replicability vs. Determinism 39
3.1.4 Experimental Conditions and Replicability 40
3.2 A Statistical Interpretation of Reproducibility and Deterministic Implementations 41
3.3 Experiments 44
3.3.1 Training 45
3.3.2 Evaluation Protocol: Average Score 47
3.3.3 Evaluation Protocol: Average Maximum Q-Values 49
3.3.4 Evaluations: Results 49
3.3.5 Policy Similarity 62
3.3.6 Type I and Type II Error 70

Chapter 4 Related Work **75**

4.1 Reproducibility in Computer Science and Artifical Intelligence 75

4.1.1 Reproducibility vs. Replicability 76

4.1.2 Replicability and Experimental Conditions 76

4.1.3 Reproducibility Efforts 77

4.2 Reproducibility in Deep Reinforcement Learning 78

4.3 The Arcade Learning Environment 79

Chapter 5 Conclusion and Future Work **81**

Appendix A Supplemental Material **84**

Bibliography **90**

List of Figures

1.1 Key differences between deep reinforcement learning and supervised deep learning. 6

1.2 The cascading effect. The stochastic policy π_0 causes two agents to have differing experiences at time step 1. This results in a different learned policies π_1 and $\pi_1^{'}$. These differing policies cause the agents to have different experiences and policies throughout the training process. 7

2.1 An artificial neuron. Each element of an input vector $\mathbf{x}$ is mutiplied by a corresponding weight in the weight vector $\mathbf{w}$. These are then summed together and a *bias* is added. This forms the output of the "summation" stage: a scalar $y = \mathbf{x} \bullet \mathbf{w} + b$. An *activation function* f is then applied to the scalar y, producing the final output of the neuron: $f(y)$. 13

2.2 A deep neural network $g : \mathbb{R}^5 \rightarrow \mathbb{R}^2$, with 2 hidden layers and a fully connected output layer. 14

2.3 A depiction of a 2D convolution. 15

2.4 An abstract depiction of RL (modeled after the diagram from Sutton and Barto [59]). At some point in time t, the RL agent experiences a state s_t and a reward r_t for his action in time $t-1$. In state s_t, the agent takes action a_t, for which the environment advances a time step, and the agent experiences a new state s_{t+1} and a reward r_{t+1} for action a_{t+1}. 17
2.5 A Soccer Player's MDP. 19
2.6 A frame from the game Pong. 25
2.7 The Deep Q-Network Architecture adapted from [39]. 31

3.1 Deterministic implementation vs. replicability. The learning curves of a deterministic implementation that is executed on two separate machines. 39
3.2 The game scores for our six experimental groups. Solid curves depict the mean score. Shaded areas represent values within one standard deviation of the mean score. The absence of a shaded area in the deterministic curves indicates that the results are identical across the five runs. 50
3.3 Values for Equation 3.1 plotted throughout training. We plot values within one standard deviation (shaded area) of the mean (solid). . . 51
3.4 The score of the deterministic agent on five individual start sequences. 52
3.5 Policy agreements on a held out set of states for different sources of nondeterminism. 65
3.5 Policy agreements on a held out set of states for different sources of nondeterminism. 66
3.5 Policy agreements on a held out set of states for different sources of nondeterminism. 67

3.6 The false negative rate for various percentage improvements upon deep Q-learning at $\alpha = 0.05$ and a sample size of 5. 74

A.1 Learning curves of the deterministic agent for individual start states/sequences. 85

A.1 Learning curves of the deterministic agent for individual start states/sequences. 86

A.1 Learning curves of the deterministic agent for individual start states/sequences. 87

A.1 Learning curves of the deterministic agent for individual start states/sequences. 88

A.1 Learning curves of the deterministic agent for individual start states/sequences. 89

List of Algorithms

1 Q-learning 22
2 Deep Q-learning with Experience Replay 30

Chapter 1

Introduction

The reproducibility of scientific results is critical both for the advancement of science and for society as a whole. Scientific results influence the medications that we are prescribed, the internal combustion engines in our vehicles, and the rockets we launch to space. In all of these scenarios, we have faith in the validity of the underlying scientific results behind these technologies. Shockingly, however, it is believed that a large amount of published research is not reproducible, and is often false [17, 28]. The inability to reproduce or verify published results places modern scientific knowledge on unstable footing, since foundational results may be called into question. As such, to maintain a trusted scientific body of knowledge, it is generally beneficial for the scientific community to move towards publishing reproducible research.

Nondeterminism is one of the reproducibility challenges that affects several scientific domains. For example, in psychology, variations in experimental context [64], such as the time of day or culture can impact the results of research studies. In other areas of science, we observe more extreme examples of nondeterminism impacting reproducibility, such as chaotic dynamical systems that exhibit *sensitive dependence to initial conditions*. In such systems, small variations in the initial

conditions can induce large variations in the outcomes or results [15].

Machine learning (ML) is not exempt from the impact of nondeterminism. In fact, nondeterminism is often explicitly embedded within ML algorithms, e.g. through stochastic gradient descent or Monte Carlo methods. However, as a computational field, ML is in a unique position amongst the sciences. In several ML research settings, nondeterminism can be controlled or even eliminated, enabling reproducibility. Since much of ML is software-driven, we can achieve *deterministic implementations*, which permit us to achieve a stricter form of reproducibility. This leads us to an important distinction — the distinction between *reproducibility* and *replicability*, which we define as follows [41]:

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

Replicability: a stricter form of reproducibility in which experimental results are able to be reproduced exactly, producing the same quantitative result.

For example, when reproducing a robotics study, it is difficult, if not impossible, to replicate exactly the original environment or setting in which the robot acted in the original study. Similarly, when reproducing a psychology study, it is impossible to have access to the exact same subjects from the original study. In such scenarios, we aim to achieve reproducibility, not replicability. We seek the same qualitative results, as opposed to achieving, say, an identical robotic grasp or an identical measurement of some psychological phenomenon.

In contrast, replicability is a much stricter form of reproducibility, and is more applicable to research that is conducted entirely through software or in simulation. In such scenarios, aspects of the environment or algorithm are often within control

of the programmer, enabling identical results to be achieved. In the context of DRL, achieving replicability requires a *deterministic implementation* to be run under identical *experimental conditions*. *Experimental conditions* refer the the software and hardware conditions under which a deterministic implementation or experiment is executed. A *deterministic implementation* is defined as:

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

It is critical to understand that deterministic implementations are not equivalent to replicable experiments. Having a deterministic implementation alone does not achieve replicability. A replicable experiment consists of a deterministic implementation *and* fixed experimental conditions. If a deterministic implementation is executed on different hardware or is compiled differently from the original experiment, the experimental results might not be replicated.

1.1 Motivation

In recent years, deep learning has begun to have a significant real world impact. Its impact spans areas such as autonomous cars [8], language translation [69], and image recognition systems [25]. In the future, as these technologies are commercialized and used to make critical decisions in our daily life [57], reproducibility becomes ever more important. In the world of ML research, though, we are primarily concerned with studying the validity of claims at a more abstract level rather than at the implementation level. As our algorithms begin to hold more important roles in our society, it is imperative that we are convinced of any results or insights we claim about our algorithms. Furthermore, if we are to have a wide adoption of our technologies across society, our systems should be rigorous and reliable as are other

mainstream technologies generally considered to be safe. Cars, roller coasters, and bridges are widely used by the general public because we consider them to be safe technologies with sound scientific underpinnings.

However, DRL faces several challenges that make reproducible research more difficult to achieve. For example, the large computational requirements of DRL algorithms often make it prohibitively expensive to have large enough sample sizes to adequately perform statistical tests. This lowers our confidence in published results and may make it more difficult, if not impossible, for others to reproduce results if the reported result is an anomaly that goes undetected due to the small sample size. Another reproducibility challenge arises from the inconsistent evaluation methodologies in DRL [26, 37], which can lead to different reported results for the same algorithm. Perhaps the most well-known reproducibility challenge in DRL is the general lack of knowledge of specific implementation details critical for achieving state-of-the-art performance. Details such as the weight initialization scheme, network architectures, minibatch sizes, learning rates, and other hyperparameters often go unreported in published work.

Nevertheless, solving these challenges remains insufficient for achieving replicability. Even with access to an implementation, nondeterminism in the training process can cause large variation in results, making it difficult to reproduce results. For example, nondeterminism can come in the forms of random sampling, random initializations of parameters, or multithreaded programs, all of which are impediments to achieving full replicability. If researchers had access to replicable experiments, they need not painstakingly expend time, often fruitlessly, acquiring the tribal knowledge needed to reproduce algorithms. Recall that to achieve replicability, we

1. require the implementation of an algorithm to be deterministic, and

2. must run the experiment under identical experimental conditions.

In this thesis, we advocate for the use of deterministic implementations for its critical role in solving the reproducibility/replicability problem.

1.1.1 The Benefits of Determinism

Perhaps the most significant benefit of deterministic implementations is that they permit replicability. However, they provide several benefits beyond replicability that are relevant to the research community.

1. *Algorithm Comparisons.* Prior work [26] has shown that performance can be substantially impacted by random seeds. Comparing two algorithms under the same deterministic conditions provides a fairer comparison of their performances. In such scenarios, usual performance differences due to nondeterminism can be avoided. Furthermore, as we will discuss later in Section 3.2, deterministic implementations allow us to utilize paired difference tests for comparing algorithms, which generally have more power than unpaired difference tests.

2. *Ablation Studies.* A central tenet to ablation studies is the notion of understanding a phenomenon by studying it in isolation. Deterministic implementations control potential confounding sources of randomness in an algorithm. This allows researchers to more confidently analyze the components of an algorithm in isolation.

3. *Debugging.* A deterministic implementation reduces the complexity of debugging. This is because the state of the program can be efficiently tracked if it is identical on each run. Efficient debugging is generally beneficial for reproducibility in that researchers can implement (or reproduce) algorithms faster.

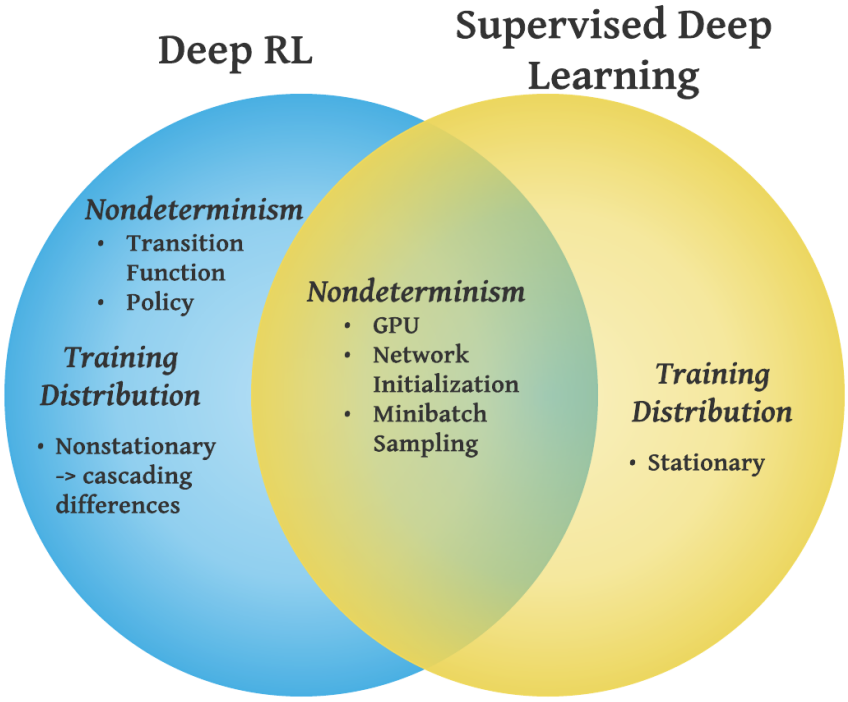


Figure 1.1: Key differences between deep reinforcement learning and supervised deep learning.

1.1.2 Nondeterminism in Deep RL

Deterministic implementations can be especially useful in the context of DRL. To understand why this is the case, consider the unique aspects of DRL that make it more susceptible to nondeterminism than supervised deep learning (SDL), as shown in Figure 1.1. Standard SDL experiences nondeterminism primarily through random network initialization and random minibatch sampling from a stationary training distribution. DRL, however differs greatly. A DRL agent must learn in the face of additional sources of randomness that stem from its policy and the environment. Moreover, because the agent's actions combined with the environment dynamics jointly determine the distribution of states that the agent experiences, its learning distribution is nonstationary. The learning process causes the agent's

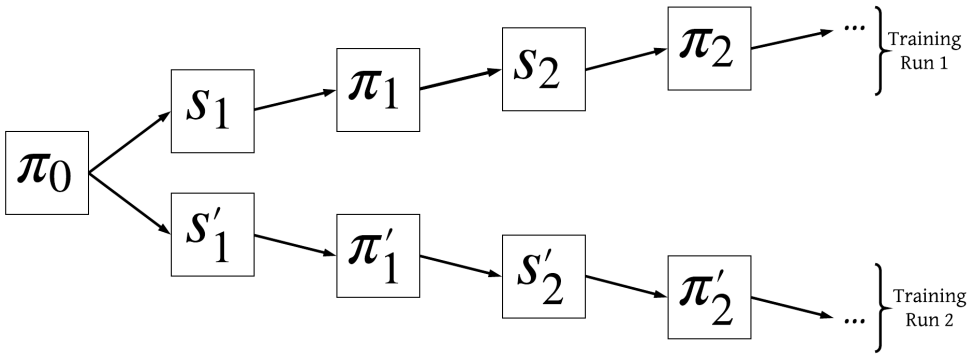


Figure 1.2: The cascading effect. The stochastic policy π_0 causes two agents to have differing experiences at time step 1. This results in a different learned policies π_1 and $\pi_1^{'}$. These differing policies cause the agents to have different experiences and policies throughout the training process.

policy to constantly change, and its state visitation distribution changes in tandem. Consider a scenario where two agents trained independently have slightly different experiences early in the training process. This small difference affects their policies, and all future policies thereafter. As a result, a small difference early in training can proliferate throughout training, causing the agents' distribution of experiences to stray apart. This *cascading effect* (see Figure 1.2) makes DRL especially vulnerable to nondeterminism. In fact, a recent empirical result highlights this susceptibility [26]. The result shows that in a specific implementation of Trust Region Policy Optimization (TRPO) [52], solely altering the global random seed can result in two groups of trained networks with statistically different performances — despite originating from the same algorithm implementation. These considerations motivate a more rigorous study on the impact of nondeterminism on achieving reproducibility in DRL.

1.2 Research Questions and Contributions

To better understand the reproducibility challenge posed by nondeterminism, we investigate the negative effects that nondeterminism has on reproducibility, as well as the positive effects of utilizing deterministic implementations. Two of our main foci are the reproducibility of an algorithm's performance (accumulated reward) and the reproducibility of its learned policies. The goal of DRL is to maximize reward, and thus naturally, the reproducibility of an agent's performance is of interest. However, in domains such as robotics, DRL may be used to train the agent, but the agent may be evaluated more qualitatively. For example, if a robot learns to walk, a pure performance-based view of reproducibility may study the distance that the robot covers, regardless of how the robot achieves this. However, we may be interested in reproducing the specific gait learned. As such, we also study the reproducibility of a policy, specifically the policy similarity between agents.

1.2.1 Research Questions

In an attempt to provide a comprehensive study of the impact of nondeterminism on reproducibility in deep reinforcement learning, we investigate the following questions:

1. **How can we produce deterministic deep reinforcement learning algorithm implementations?**
2. **To what extent does nondeterminism impact the reproducibility of performance in deep reinforcement learning?**
3. **To what extent does nondeterminism impact the reproducibility of policies in deep reinforcement learning?**

4. **To what extent does nondeterminism impact the quality of DRL research?**

1.2.2 Contributions

In addressing these research questions, this thesis makes the following contributions:

1. **Deterministic implementation of deep Q-learning**. In analyzing the impact of nondeterminism, we first produce a deterministic implementation of Deep Q-learning. To do so, in Section 3.1 we identify and eliminate all sources of nondeterminism from the training process, producing a fully deterministic implementation. In addition, we provide guidelines for producing deterministic implementations of deep reinforcement learning algorithms more generally.

2. **Sensitivity analysis of individual sources of nondeterminism.** Having identified the sources of nondeterminism in the training process, in Section 3.3 we perform a simple one-factor-at-a-time (OFAT) sensitivity analysis on the sources of nondeterminism. We allow an individual source of nondeterminism to affect the outcome of several deep Q-learning training runs, while holding other sources of nondeterminism fixed. We then measure the impact of each source of nondeterminism on the agent's performance and policy. We demonstrate that all sources of nondeterminism substantially hinder the reproducibility of the performance of an agent. Our results for the impact of nondeterminism on the reproducibility of the policy were inconclusive.

3. **Improved experimental methodology for DRL research.** Algorithm comparisons are performed quite frequently, given the fast pace of DRL research. By utilizing deterministic implementations, we describe how we can exploit the increased statistical power of *paired difference tests*, as opposed to the standard unpaired difference tests used to compare algorithms. In Sec-

tion 3.3, we measure the Type I (false positive) and Type II (false negative) errors caused by nondeterministic implementations, which can be more easily avoided through deterministic implementations.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the reader with foundational knowledge on neural networks and reinforcement learning. Chapter 3 discusses our formulation, experiments, and results. Chapter 4 discusses related work on reproducibility in machine learning and deep reinforcement learning. Chapter 5 summarizes our contributions, reflects on lessons learned, and proposes potential avenues for future work.

Chapter 2

Background

In this chapter, we provide an overview of neural networks, reinforcement learning, deep reinforcement learning, and convergent learning. These areas together constitute the essential background needed to develop our experiments in later chapters.

2.1 Neural Network Learning

Artificial Neural Networks are abstractions of biological neural networks that are used to compute complex functions. A certain class of neural networks, called *deep* neural networks, have had a significant impact in recent years across many areas of artificial intelligence including reinforcement learning, natural language processing, and visual recognition. Oftentimes, humans hand-design the *features* of a learning task: the elements of the training data that are relevant for learning. Hand-designed features can be too difficult to design by hand or too simple to be of major practical use in many problems. One of the biggest advantages of deep neural networks is that they can learn hierarchical representations [33] from raw data, ultimately *learning the features* as opposed to using hand-crafted features. They can approximate a diverse set of functions, and are powerful tools in modern artificial intelligence and

machine learning.

2.1.1 Artificial Neurons and Neural Networks

Before examining neural networks in their entirety, one must understand the core ingredient from which neural networks are built from: the *artificial neuron*. An artificial neuron (or simply, neuron) is a function $u : \mathbb{R}^n \to \mathbb{R}$, consisting of the following elements:

- A vector $x \in \mathbb{R}^n$ of *inputs*.
- A scalar $b \in \mathbb{R}$ called the *bias*.
- A *weight* vector $w \in \mathbb{R}^n$.
- An *activation function* $f : \mathbb{R} \to \mathbb{R}$, which can be nonlinear.

A neuron takes the dot product of the input vector x and the weight vector w, and adds the bias b, obtaining $z = x \bullet w + b$. It then applies the activation function to z, outputting $f(z)$. Formally, the neuron is

$$u(x) = f\left(\left(\sum_{i=1}^{n} w_i \cdot x_i\right) + b\right) \tag{2.1}$$

An *artificial neural network* (ANN), or just *neural network*, is a directed graph with weighted edges consisting of individual artificial neurons, or *units*. The nodes in the graph represent individual artifical neurons. An edge from unit a to b represents the output of neuron a entering neuron b as an input. Since each edge represents an input to a neuron, each edge has an associated weight.

In aggregate, a neural network represents a function $g : \mathbb{R}^n \to \mathbb{R}^m$. A vector $\mathbf{x} = [x_1 x_2 ... x_n]^T$ of inputs enters the network, undergoes several compositions of

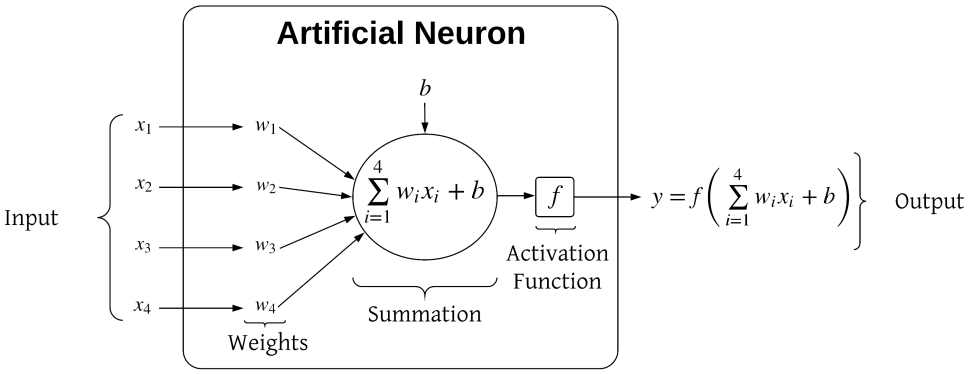


Figure 2.1: An artificial neuron. Each element of an input vector $\mathbf{x}$ is mutiplied by a corresponding weight in the weight vector $\mathbf{w}$. These are then summed together and a *bias* is added. This forms the output of the "summation" stage: a scalar $y = \mathbf{x} \bullet \mathbf{w} + b$. An *activation function* f is then applied to the scalar y, producing the final output of the neuron: $f(y)$.

functions as specified by the neurons within the network, and the network ultimately outputs some output vector $\mathbf{y} \in \mathbb{R}^m$.

A *feedforward neural network* is a network whose graph representation has no cycle. Such networks are typically organized in layers, with the inputs entering through the *input layer*, passing through a series of *hidden layers*, and the final *output layer* outputs the result. Neural networks with multiple hidden layers are known as *deep neural networks*. A layer is *fully connected* if every neuron in the layer has an input for each output of the previous layer. Figure 2.2 depicts a typical neural network.

Neural networks are trained through *supervised learning*, where there is a prespecified set of *labeled examples* as tuples $(\mathbf{x}, \mathbf{y})$. The goal of training neural networks is to minimize some *loss function* $E : \mathbb{R}^m \to \mathbb{R}$. The parameters of the neural network are its weights, and we train the networks through *stochastic gradient descent* and continuously modify the weights of the network to minimize E. On a given training example $(\mathbf{x}_i, \mathbf{y}_i)$, the network outputs $\mathbf{y}_{pred} = g(\mathbf{x}_i)$. The

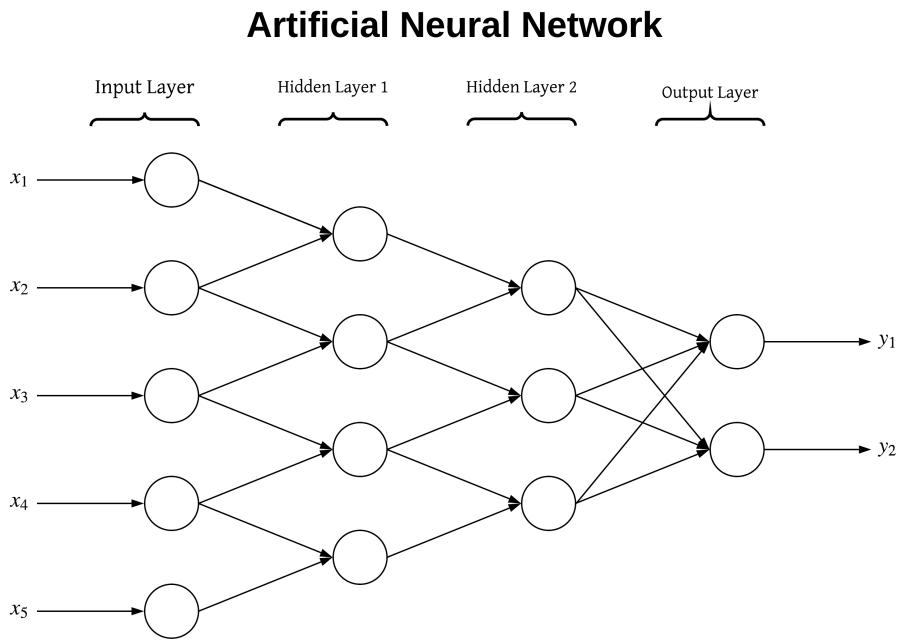


Figure 2.2: A deep neural network $g : \mathbb{R}^5 \to \mathbb{R}^2$, with 2 hidden layers and a fully connected output layer.

error $E(\mathbf{y}_{pred})$ is computed, and then through the *backpropagation algorithm* [33] the partial derivatives $\frac{\partial E}{\partial w_{ij}}$ can be computed (where w_{ij} is the weight of the edge from unit i to unit j). With the derivatives computed, we can perform gradient descent and learn the weights of the neural network.

2.1.2 Convolutional Neural Networks

Many of the recent advances in deep learning are due to a class of neural networks known as *convolutional neural networks* (CNNs), inspired by visual neuroscience [33]. CNNs are designed for array data such as images, and have played critical roles in successes including image classification [32], object detection [49], Atari gameplay [40], and visual question answering [36], to name a few.

Most CNNS consist of two types of layers, convolutional layers and pooling layers. In this thesis, our focus is on convolutional layers. Pooling layers are effective

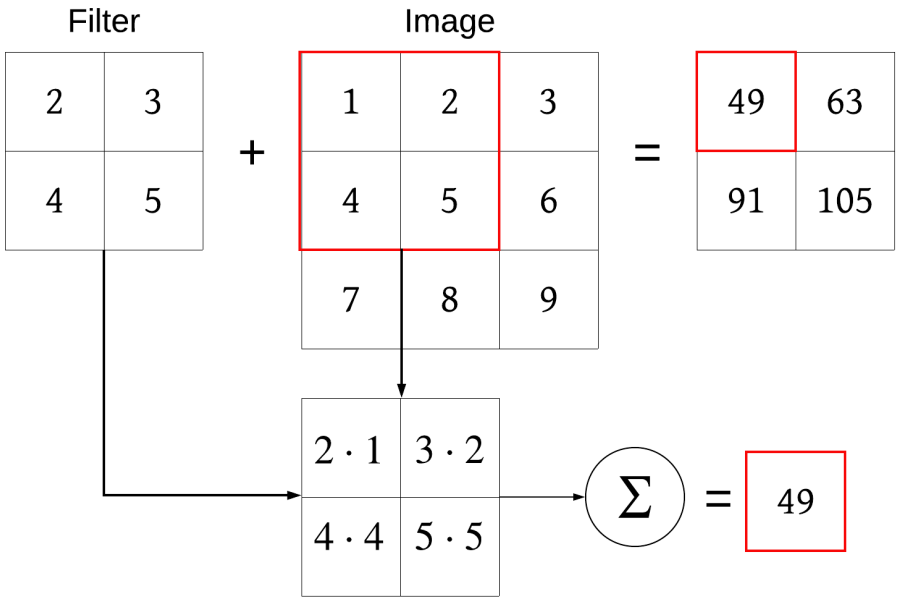


Figure 2.3: A depiction of a 2D convolution.

for achieving *spatial invariance*, in problems for which location is not as relevant. For example, when classifying an image as a 'tiger', the tiger can appear in different locations in the image. Pooling layers help solve this issue. For the tasks we consider in this thesis, we do not desire this property. As such, pooling layers are absent in our networks.

Convolutional layers consist of several *filters*, or *kernels*, which are typically square matrices when the inputs are images. Each filter "slides" across various positions of the image and performs an element-wise multiplication between the values in the filter matrix and the pixel values of the image. Once this element-wise multiplication is performed, the output is simply the sum of these element-wise products. The locations that the filter is placed at within the image is specified by the *stride*, which, as its name suggests, dictates how the filter "strides" across the image. Figure 2.3 depicts an example of a convolution.

Convolutional layers allows local correlations in image-like data to be ex-

ploited [33]. Further, by having several convolutional layers, hierarchies in the data can be harnessed. When convolutional neural networks are trained, the learned weights are the filters themselves. The goal during training is to learn the appropriate filters that are bested suited to the data.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm for solving sequential decision-making problems. In RL, learning occurs as an agent *interacts* with his environment. The agent observes the "state" of the environment, takes an action, and is given a *reward* by the environment. The agent then "transitions" to a new state, and the cycle repeats in a sequence of states, actions, and rewards. To provide a human analogy, we can consider our environment to be the world we live in. The states we observe through our senses corresponds to the state of the world at some point in time, and the actions we take are the decisions we make. Our actions affect the subsequent states we experience, and the consequences of our actions represent the reward we receive from the environment.

It is important to note that RL differs from both supervised learning and unsupervised learning [59]. In supervised learning, a machine learning algorithm uses a training set of *labeled examples*. The goal of the algorithm is to be able to learn from these examples and generalize the learned knowledge. That is, when given novel examples, the ML algorithm should be able to predict their corresponding labels. However, in RL, the agent does not learn from examples attached with labels of the correct output, but rather the agent learns from experiences. The reward signal is indeed a form of feedback for the agent, but it does not indicate the *correct* output for the agent, as is the case in supervised learning. It is the RL agent's responsibility to utilize its combined set of experiences and feedback to learn the optimal behavior.

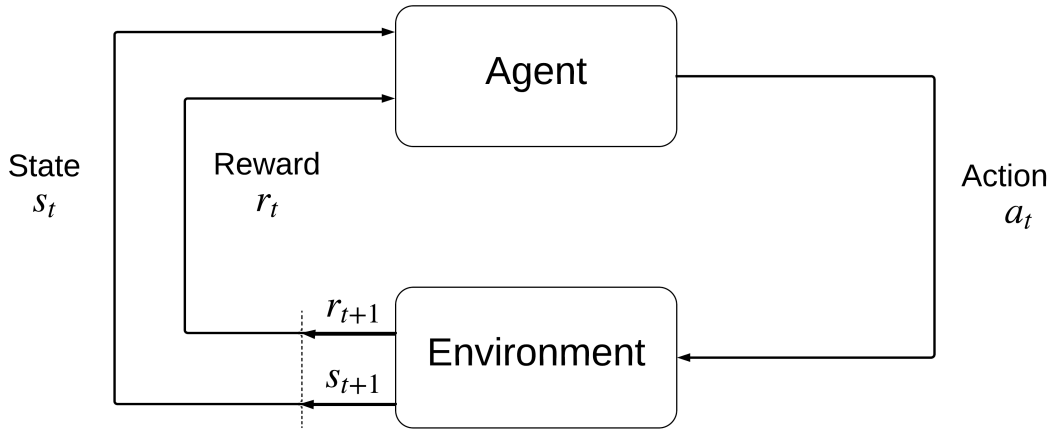


Figure 2.4: An abstract depiction of RL (modeled after the diagram from Sutton and Barto [59]). At some point in time t, the RL agent experiences a state s_t and a reward r_t for his action in time $t-1$. In state s_t, the agent takes action a_t, for which the environment advances a time step, and the agent experiences a new state s_{t+1} and a reward r_{t+1} for action a_{t+1}.

In unsupervised learning, a machine learning algorithm is given a set of *unlabeled* data, with the goal of discovering some patterns or structure within the dataset. While the reward signal is indeed a form of feedback that the RL agent receives, in unsupervised learning the agent does not receive any form of feedback. Additionally, the goal of finding hidden structure within a dataset is not what an RL agent seeks to do per se, though finding hidden structure in the environment can be useful. Rather, the RL agent's objective is to learn the behavior that will allow it to maximize its cumulative discounted reward over many time steps. In doing so, the agent does not necessarily need to perfectly discover hidden structure within the data, and as such unsupervised learning and RL simply have two different objectives.

The remainder of this section focuses on the formulation of RL problems as Markov Decision Processes, and discusses tabular and approximate methods for solving Markov Decision Processes.

2.2.1 Markov Decision Processes

The sequential decision-making problems posed by RL are typically formulated as *Markov Decision Processes* (MDPs). An MDP captures the Agent-Environment interaction depicted in Figure 2.4. Formally, an MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, T, \gamma, \mathcal{R})$ in which:

- $\mathcal{S}$ denotes the set of *states* in the environment.
- $\mathcal{A}$ denotes the set of *actions* the agent can take.
- T denotes the *transition function* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, a probability measure where $T(s, a, s')$ is the probability of the agent transitioning to state s' when in state s and taking action a.
- $\gamma \in [0, 1]$ is the *discount rate* that indicates how much an agent values future rewards.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function* that specifies the reward an agent receives during a transition. That is, when the agent takes action a in state s, and transitions to s', it receives a reward of $\mathcal{R}(s, a, s')$.

For the remainder of this thesis, the MDPs we work with are assumed to be *finite*. That is, $|\mathcal{S}|$ and $|\mathcal{A}|$ are finite.

When acting in an MDP, the goal of a learning agent is to maximize its expected *discounted return*. With each decision the agent makes, it attempts to maximize its expected discounted return. The focus of RL is to learn how the agent can achieve its goal of maximizing its expected discounted return.

To solve an MDP, the agent must learn the correct action or actions to perform at each state. Formally, the agent must learn a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maps a state-action pair (s, a) to the probability of taking action a in state s. Essentially, the policy π is the agent's decision-making function that specifies

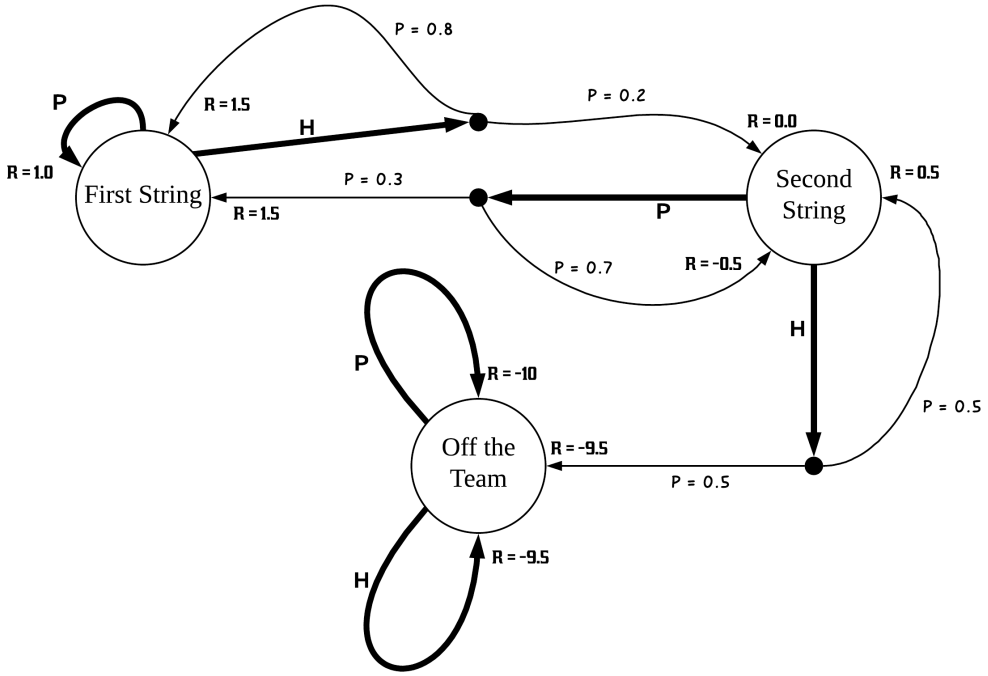


Figure 2.5: A Soccer Player's MDP.

how that agent should act, and can be either deterministic or nondeterministic. In its goal of maximizing its expected discounted return, the agent attempts to learn an *optimal policy* π^*, a policy that if followed, will achieve an expected discounted return greater than or equal to that of any other policy.

Consider the example MDP in Figure 2.5. In this example, we consider an MDP for a soccer player. The player can be in three possible situations, he can be a "First String" player that plays frequently, a "Second String" player that plays infrequently, or he can be "Off the Team" and never play. Thus the player's state space is $\mathcal{S} = \{$First String (FS), Second String (SS), Off the Team (OtT)$\}$. Now the player can do one of two things: "Practice" or "Horse around". Thus his action space is $\mathcal{A} = \{$Practice ($\mathbf{P}$), Horse around ($\mathbf{H}$)$\}$. The player likes to play in the games, but he also enjoys horsing around. He starts off as a first string player, but if he horses around, he runs the risk of playing less frequently as a second string

player. Similarly, if he horses around as a second string player, he might be forced off of the team, after which he can never play soccer for the rest of the season. In subsequent sections, we will study various methods to solve this and more complex MDPs.

2.2.2 Value Functions and the Bellman Equations

We have stated that the goal of the agent is to learn a policy that maximizes its expected discounted return. But how can we quantify the expected discounted return an agent will receive when following a policy π? This information is encoded in the *value function* $V_\pi : \mathcal{S} \to \mathbb{R}$, that maps a state s to the expected discounted reward the agent will receive if following policy π from state s.

Suppose we have an MDP $M = (\mathcal{S}, \mathcal{A}, T, \gamma, \mathcal{R})$. Suppose the agent is following some policy π. The *value* of state $s \in \mathcal{S}$ is the expected discounted return the agent receives if it starts in state s and follows policy π. That is:

$$V_\pi(s) = \sum_a \pi(s,a) \left(\sum_{s'} T(s,a,s')(\mathcal{R}(s,a,s') + \gamma V_\pi(s')) \right).$$

This is the Bellman equation [59]. $V^* = V_{\pi^*}$ denotes the value function of the optimal policy π^*. It satisfies the recursive relationship defined by the Bellman optimality equation:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')(\mathcal{R}(s,a,s') + \gamma V^*(s')) \tag{2.2}$$

Why do we care about the value function, when the goal of the agent is to find the optimal policy? As stated previously, the goal of the agent is to maximize its expected discounted return. If the agent can learn the value function of the optimal policy V^*, then the agent indirectly learns the optimal policy. Specifically,

if the agent learns V^*, then an optimal policy π^* can be deduced:

$$\pi^*(s,a) = \begin{cases} 1 & \text{if } a = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')(\mathcal{R}(s,a,s') + \gamma V^*(s')) \\ 0 & \text{otherwise} \end{cases}. \tag{2.3}$$

Just as we have defined $V_\pi(s)$ to be the expected discounted return of a state s when following a policy π, we can define a *state-action value function* [59] $Q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, that equals the expected discounted return when *taking action a in state s and following the policy π thereafter*. Formally,

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_\pi(s')).$$

Similarly, there is an optimality equation for Q as well:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s')). \tag{2.4}$$

Notice the relationship between equations 2.2 and 2.4, we see that

$$V^*(s) = \max_a Q^*(s,a).$$

We can rewrite our optimal policy as

$$\pi^*(s,a) = \begin{cases} 1 & \text{if } a = \underset{a}{\operatorname{argmax}}\, Q^*(s,a) \\ 0 & \text{otherwise} \end{cases}. \tag{2.5}$$

Consider the difference between equations 2.3 and 2.5. Observe that if an agent has knowledge of V^*, it must have access to the transition model T in order to execute an optimal action. However, if the agent has a knowledge of Q^*, it does not need access to the transition model T. This will be important when we study

Q-learning in the next section.

2.2.3 Q-learning

We have discussed the formulation of RL problems as MDPs, and have defined the value function V and the action-value function Q, and described their relationships. We have found that an optimal policy can be deduced from the optimal value functions V^* or Q^*. In this section, we will introduce a method for learning Q^*, called Q-learning [66].

Q-learning is an online algorithm where an agent continually interacts with the environment, and uses those experiences to learn Q^*. At some time step t, an agent is in state s_t, takes an action a_t, and observes a new state s_{t+1} while receiving a reward of $\mathcal{R}(s_t, a_t, s_{t+1})$. For this transition, the agent applies the update rule [59]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\mathcal{R}(s_t, a_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.6}$$

where α is a *step size*. The full algorithm is shown in Algorithm 1

Algorithm 1 Q-learning

$\forall s, a \in \mathcal{S} \times \mathcal{A}$, initialize $Q(s, a)$ arbitrarily
repeat
  $t \leftarrow 0$
  repeat
    In state s_t, select action a_t according to π.
    Take action a_t, observe state s_{t+1}, receive reward $\mathcal{R}(s_t, a_t, s_{t+1})$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\mathcal{R}(s_t, a_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
    $t \leftarrow t + 1$
  until episode terminates
until forever

If α is set appropriately, and all state-action pairs are visited infinitely, Q-learning converges and learns Q^*. Note that the policy π that the agent uses is

irrelevant as long as it visits all state-action pairs infinitely. This characterizes Q-learning as an *off-policy* algorithm, where the policy that dictates which states and actions are visited differs from the policy that is learned. In the case of Q-learning, the action-value function of the optimal policy is learned while the visited states can be generated from a different policy. Also note that this algorithm is *model-free* in that it does not make assumptions about or make use of a transition model.

2.2.4 Q-learning with Function Approximation

In the previous section we introduced Q-learning in the tabular setting, where we have access to every possible state and action pair, and store a table of all state-action pairs, and update them throughout the algorithm. However, in practical scenarios, it is often infeasible to store a table of the entire state-action space. In such scenarios, we have methods to approximate the value function, using some form of function approximator.

Suppose that instead of storing a table of all state-action pairs, we could represent a state action pair (s, a) with a *feature vector*

$$\phi(s, a) = [\phi_1(s, a)\ \phi_2(s, a)\ ...\ \phi_m(s, a)]^T.$$

Let us also suppose that $Q(s, a)$ is linear function of its features ϕ_i, $i \in [1, m]$. Then instead of learning the value of each state-action pair as in the tabular setting, we can learn a *weight vector* $\boldsymbol{\theta} = [\theta_1\ \theta_2\ ...\ \theta_m]^T$ such that $Q(s, a) = \boldsymbol{\theta}^T\phi(s, a)$. Instead of directly learning Q^*, we learn the optimal weight vector $\boldsymbol{\theta}^*$ such that

$$Q^*(s, a) = \boldsymbol{\theta}^{*^T}\phi(s, a).$$

The update rule for *approximate* Q-*learning* is on the weights [38] rather than the

tabular Q-values from Equation 2.6:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} Q(s_t, a_t)[\mathcal{R}(s_t, a_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.7}$$

$$= \boldsymbol{\theta} + \alpha \phi(s_t, a_t)[\mathcal{R}(s_t, a_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.8}$$

In this subsection we studied the simple example of using a linear function approximator to approximate Q. It is important to note that Q-learning with function approximation sometimes diverges [58], even in the simple case of linear function approximation. In contrast, there are on-policy algorithms, i.e. algorithms where the agent learns Q_π rather than Q^*, that are known to converge under linear function approximation. However, despite this drawback, Q-learning with linear function approximation is still used frequently and often works well. Unfortunately, Q-learning faces similar challenges when we move to the case of nonlinear function approximation. Nevertheless, neural networks can serve as effective nonlinear function approximators in practice, as we will see when we introduce deep reinforcement learning in Section 2.4. In fact, we will study an extension of Q-learning with function approximation that allows us to train a neural network to approximate Q.

For a comprehensive treatment on reinforcement learning, see the text by Sutton and Barto [59].

2.3 The Arcade Learning Environment

The Arcade Learning Environment (ALE) serves as a platform for evaluating artificial intelligence agents by providing an interface to play Atari 2600 games [7, 37]. The ALE allows agents to be tested on dozens of different games that are difficult for humans and diverse in their nature. Consequently, the ALE is accepted as a platform for evaluating the general competency of algorithms and has been frequently used in recent years for testing RL agents. In the ALE, an agent receives video input

of the game screen and takes actions within the game, with the goal of maximizing its game score. As the agent selects an action, the ALE executes it and feeds the agent a reward and a new frame, until the game ends.

More formally, the ALE's MDP formulation is as follows. At each time step, the agent observes a single frame. For example, observe Figure 2.6, a single frame from the game of PONG. Notice how the direction of the ball cannot be determined

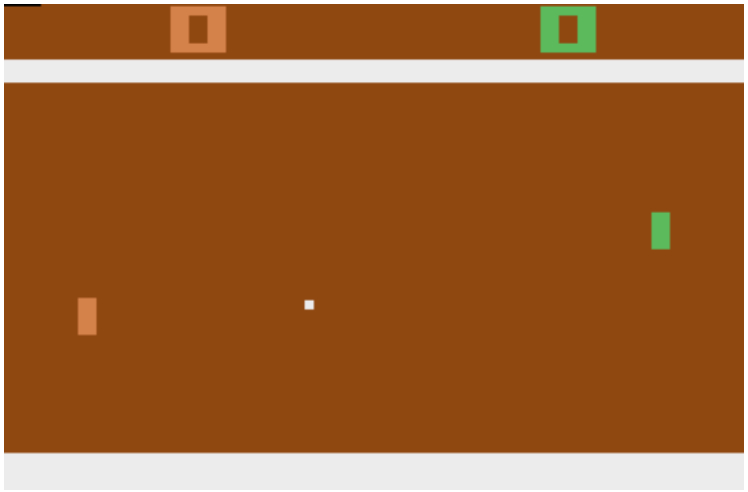


Figure 2.6: A frame from the game Pong.

simply by observing this frame. We typically infer the direction of the ball by having a knowledge of the previous frames in the sequence of observed frames. The problem here is known as *perceptual aliasing*, where two different states can be perceived identically. This makes several Atari games *partially observable*, because an agent's observation (a single frame) does not capture the full state, i.e. the state of the 1024-bit RAM. Thus the state space of the ALE consists of the set of possible RAM states. However, in the algorithms we will discuss, the agent's observations consist solely of the 210x160 frames.

The action space consists of 18 actions [7], formed as a combination of joystick motions and a button. Some games have action sets that are subsets of these 18

actions.

The transitions in the ALE are deterministic. However, a recent version of the ALE adds a version of stochasticity in the form of "sticky-actions" in which the agent's previous action is repeated with some parameterizable probability [37]. Other methods to address the determinism of the ALE exist, such as the use of no-ops, which we will discuss in subsequent sections.

The reward that the agent receives at a particular time step is usually its change in game score. That is, the reward the agent receives at time step t is the difference between the score at time t and time $t-1$.

Given that an agent's objective is to maximize its total game score, we keep the discount rate γ near 1, so that the agent learns to value rewards for the entire episode.

2.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) refers to reinforcement learning with deep neural networks as function approximators. The foundation of this thesis focuses on a specific neural network architecture, called a Deep Q-Network (DQN), used to approximate the state-action value function Q.

2.4.1 Deep Q-learning

One of the original and successful DRL algorithms is *Deep Q-Learning with Experience Replay* [40] (or just deep Q-learning), which is used to train a Deep Q-Network (DQN). Deep Q-learning is very successful at Atari game play, and was one of the first algorithms to exhibit general competency, as it was able to achieve human or even superhuman performance on several domains with high dimensional state-spaces, using only the pixels and game score as inputs to the algorithm. It is a black-box algorithm in that it does not exploit domain-specific knowledge or fea-

tures. It must learn the appropriate representation for each game autonomously, without modifying the network architecture or hyperparameters on a task-by-task basis.

Deep Q-learning uses a convolutional neural network as a function approximator for Q (hence the term deep Q-network). The architecture for the DQN that we use follows [39] and is depicted in Figure 2.7. When using a neural network as a function approximator, RL has been known to be unstable [63]. Deep Q-learning addresses three issues causing this instability:

1. When the agent observes trajectories of states, the states within the trajectory are highly correlated.

2. Small updates to Q can change the policy which changes the distribution of the experiences the agent will receive.

3. There are correlations between $Q(s, a)$ and the Q-learning target $\mathcal{R}(s, a, s') + \gamma \max_{a'} Q(s', a')$.

The stochastic gradient descent methods that are used to train neural networks often assume that the minibatches used to train the networks are independent and identically distributed. However, when an agent experiences a trajectory or episode, each state is highly correlated with the subsequent state. Consequently, online updating of the Q-network as states are experienced breaks any independence assumptions we have about the states used to update the network. To resolve this, we use an *experience replay* buffer[35]. This replay buffer $\mathcal{D}$ stores the last N transitions (s, a, r, s') (we use $N = 1000000$), and we sample minibatches uniformly from this replay buffer when training the network. In effect, this makes the distribution of training data more stationary, and attenuates the correlations between the minibatches used for updates. In contrast, online updating of the weights follows the distribution of data being generated by the agent's current policy, which is

constantly changing, making the data distribution much more nonstationary. Recall that in subsection 2.2.3 we stated that Q-learning is an off-policy learning algorithm. Since the experience replay buffer stores transitions generated from old policies, an off-policy algorithm like Q-learning is suited to learning from such data.

In tabular Q-learning, the representation of the value $Q(s, a)$ and the target value $\mathcal{R}(s, a, s') + \gamma \max_a Q(s', a')$ are distinct, because the state-action pairs (s, a) and (s', a') each have their own entries in the table. However, when using function approximation, as in a neural network, we find that $Q(s, a; \boldsymbol{\theta})$ and $\mathcal{R}(s, a, s') + \gamma \max_a Q(s', a'; \boldsymbol{\theta})$ share the parameter $\boldsymbol{\theta}$. As a result, when we update $\boldsymbol{\theta}$ to move $Q(s, a; \boldsymbol{\theta})$ towards the target $\mathcal{R}(s, a, s') + \gamma \max_a Q(s', a'; \boldsymbol{\theta})$, we are *also* modifying the target, since it is also a function of $\boldsymbol{\theta}$. To see how this can potentially be a problem, suppose the RL agent experiences the transition (s, a, s'), and a large reward $\mathcal{R}(s, a, s')$. Then it is likely that the agent will increase its estimate of $Q(s, a)$. However, since the state-action value is aliased with the network representation $Q(s, a; \boldsymbol{\theta})$, and in Atari games, the state s is perceptually similar to s', we can unintentionally increase the value of $Q(s', a'; \boldsymbol{\theta})$. Next time the transition (s, a, s') is experienced, the target $\mathcal{R}(s, a, s') + \gamma \max_a Q(s', a'; \boldsymbol{\theta})$ will be larger. If this transition is repetitively experienced, this target can grow larger and larger, leading to instability. To mitigate this issue, we use a *target network*, whose parameters are denoted $\boldsymbol{\theta}^-$. The target network, as its name suggests, is used to compute the target $r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}^-)$ for a transition (s, a, r, s'). We keep the target network $\boldsymbol{\theta}^-$ fixed (i.e. we do not change $\boldsymbol{\theta}^-$) while learning $\boldsymbol{\theta}$, thereby reducing the correlations between the state-action value and the target. Periodically, we perform the update $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$, because we need the targets to change in order for learning to occur, though not as frequently as each update. Given the replay buffer $\mathcal{D}$ of

transitions, the loss function obtained at iteration i is [40]

$$L_i(\boldsymbol{\theta}_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}_i^-) - Q(s, a; \boldsymbol{\theta}_i)\right)^2\right]. \tag{2.9}$$

That is, to minimize the loss is to minimize the expected squared temporal difference error of a training example drawn uniformly from $\mathcal{D}$.

In implementing the deep Q-learning algorithm to play Atari games, there are several critical techniques employed. First, all observed frames are preprocessed by being grayscaled and rescaled from 210×160 to 84×84. Further, to combat the partial observability of the environment, the agent uses a function ϕ that stacks the previous m frames to construct the agent's input to the neural network. However, the full state at time t is the sequence of frame-action pairs preceding t in that episode $s_1, a_1, ..., s_t$. In order for the agent to experience more transitions, we employ *frame-skipping*, where an agent's action is repeated for k frames. Further, since the ALE is deterministic, the agent performs a random number of no-op (or "do nothing") actions, in order to randomize the initial state, preventing the agent from taking advantage of the ALE's determinism.

The agent uses an ϵ-greedy policy, whereby it selects the action $\operatorname{argmax}_a Q(s, a; \boldsymbol{\theta}_i)$ with probability $1-\epsilon$, and selects a random action with probability ϵ, where $\epsilon \in [0, 1]$. The motivation behind this policy is that the agent *exploits* its current knowledge of the environment with probability $1-\epsilon$, and *explores* the environment with probability ϵ, so as to obtain new experiences that can improve its estimate of Q, improving future exploitation.

The network architecture we employ, adapted from [39], is depicted in Figure 2.7. The network takes as input an 84 x 84 x 4 image consisting of 4 stacked and preprocessed 84 x 84 frames. The first hidden convolutional layer consists of 16 8x8 filters with stride 4 followed by a rectifier nonlinearity [43]. The second hidden convolutional layer consists of 32 4x4 filters with stride 2 followed by a rectifier

nonlinearity. The final hidden layer is a full connected layer of 256 neurons with ReLU activations. The output layer is fully connected with a neuron for each valid action in the game (ranges from 4 to 18).

Algorithm 2 Deep Q-learning with Experience Replay

$\mathcal{D} \leftarrow \emptyset$, with capacity N
Initialize $\boldsymbol{\theta}$ randomly
$\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
for episode $e \leftarrow 1$ **to** M **do**
  Initialize sequence $s_1 = \{o_1\}$, and $\phi_{s_1} = \phi(s_1)$
  for time step $t \leftarrow 1$ **to** T **do**
    $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \underset{a}{\text{argmax}}\, Q(s, a; \boldsymbol{\theta}) & \text{with probability } 1 - \epsilon \end{cases}$
    Execute(a_t) and receive reward r_t and frame o_{t+1}
    Let $s_{t+1} = s_t, a_t, o_{t+1}$ and let $\phi_{s_{t+1}} = \phi(s_{t+1})$,
    Store the transition $(\phi_{s_t}, a_t, r_t, \phi_{s_{t+1}})$ in $\mathcal{D}$
    Sample a minibatch of transitions $(\phi_{s_j}, a_j, r_j, \phi_{s_{j+1}})$ uniformly from $\mathcal{D}$
    Target $y_j = \begin{cases} r_j & \text{if episode terminates at } j+1 \\ r_j + \max_{a'} Q(\phi_{s_{j+1}}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform gradient descent with respect to $\boldsymbol{\theta}$ with loss $(y_j - Q(\phi_{s_j}, a_j; \boldsymbol{\theta}))^2$
    Every C steps, let $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
  end for
end for

Detailed overviews of the Deep Q-learning algorithm can be found in [39, 40].

2.4.2 Other Deep Reinforcment Learning Algorithms

While our focus in this thesis is on the deep Q-learning algorithm, it is worthy to note the numerous advances in DRL that have followed since deep Q-learning's inception. Recall that deep reinforcement learning generally refers to reinforcement learning using a deep neural network as a function approximator. Since deep Q learning was introduced, there have been several advances that improve upon it, all of which use a neural network to represent a value function. For example, double DQN [65] addresses overestimation issues of DQNs. Prioritized double DQN [51]

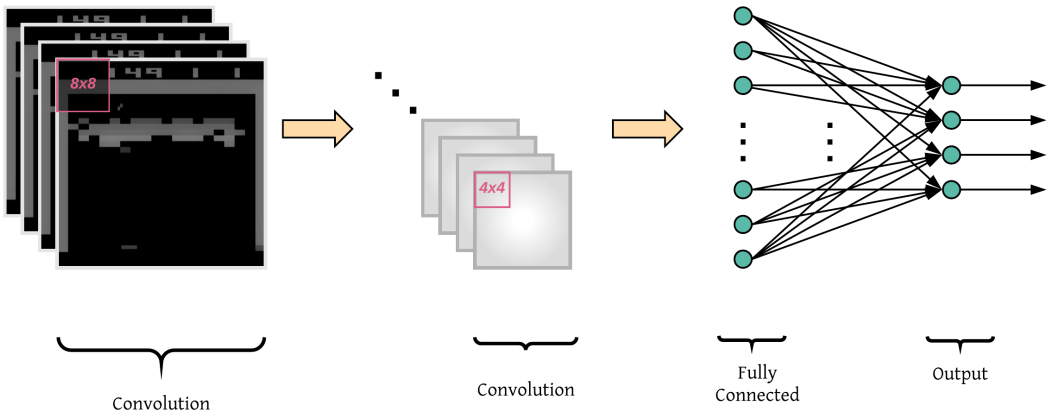


Figure 2.7: The Deep Q-Network Architecture adapted from [39].

improves upon DQNs by prioritizting certain experiences from the replay buffer over others. Categorical DQN [6] models a distribution of over value functions, or a value distribution, and exhibits superior performance to the previously mentioned algorithm. Rainbow [27] combines several improvements upon DQNs into a single agent, achieving state-of-the-art results. In spite of these significant advancements upon DQNs, deep Q-learning remains a relevant algorithm in modern DRL research, and continues to serve as a popular baseline for algorithms. Given our focus on reproducibility, we chose to study deep Q-learning because of its critical role to the DRL community.

Thus far we have discussed value based deep reinforcement learning algorithms, where a deep neural network is used to approximate a value function. However, there remains another class of deep reinforcement learning algorithms, *policy gradient* methods, which use deep neural networks to directly approximate the policy, instead of using it to approximate the value function. Several policy gradient algorithms [34, 52, 53] have been developed for deep reinforcement learning, such as trust region policy optimization (TRPO) [52], proximal policy optimization (PPO) [53], and deep deterministic policy gradient (DDPG). Such policy gradient

algorithms have been particularly effective at continuous control tasks and robotics tasks. While we do not delve into policy gradient algorithms in this thesis, the reader should note that the reproducibility of policy gradient DRL methods has been studied extensively by Henderson et al. [26].

2.5 Statistical Formulation of Deep Reinforcement Learning Algorithms

This section describes the statistical problem of comparing the performance of two DRL algorithms, and is based off of work by Colas, Sigaud, & Oudeyer [12].

Since DRL environments are often stochastic in nature, the performance of an algorithm can be modeled as a random variable X. Training and evaluating an algorithm n times results in a sample of performances $x = x^1, ..., x^n$. Suppose we are comparing two algorithms A and B, whose performances are modeled by random variables X_A and X_B. Subtracting these random variables gives us the random variable $X_{A-B} = X_A - X_B$. When we compare two DRL algorithms, we are typically trying to show that one algorithm achieves a better performance, on average, than another. Since we are comparing averages, we can test whether the mean $\mu_{A-B} = \mu_A - \mu_B > 0$. To do so, we can perform a type of hypothesis test known as a *difference test*. Suppose we want to show algorithm A performs better than algorithm B, then we have the following null and alternative hypotheses (*one-tailed*):

- $H_0 : \mu_{A-B} \leq 0$
- $H_a : \mu_{A-B} > 0$

There are two *unpaired* difference tests that are commonly used to test such null hypotheses: the two-sample t-test and the Welch's t-test [67]. Both tests assume

normally distributed or bell-shaped data. The two-sample t-test assumes that both X_A and X_B have equal variances, which is not an assumption that holds in practical DRL scenarios. However, the Welch's t-test can be used in the scenario where X_A and X_B have unequal variances, and it is the test that we use in this thesis.

When performing hypothesis testing, we wish to avoid two types of errors: Type I and Type II errors. Type I errors, or *false positives*, occur when we incorrectly reject the null hypothesis. Type II errors, or *false negatives*, occur when we incorrectly *fail* to reject the null hypothesis. In the context of DRL algorithm comparisons, a Type I error is analogous to finding that one algorithm is better than another when in reality it is not [12]. The analogy for a Type II error is failing to detect that one algorithm is better than another when it is indeed the case.

For a given sample size n, we can empirically estimate the Type I error, using the following procedure given by Colas et al. [12]:

1. For a given algorithm, generate $2n$ trials.
2. Randomly split the $2n$ trials into two groups a and b, each of size n. We view group A as a sample from some algorithm A, and we view group b as a sample from some algorithm B. Note that since the data in each group comes from the same algorithm, the null hypothesis holds true.
3. Perform a difference test (e.g. Welch's t-test) and tally the result.
4. Repeat steps 1-3 T times.
5. The empirical Type I error is the proportion of times that the null hypothesis was rejected.

This procedure will be used in Section 3.3, to measure the Type I error when trials are generated by the deep Q-learning algorithm.

Chapter 3

Nondeterminism and Reproducibility in Deep Reinforcement Learning

In this chapter[1], we explore the impact of nondeterminism on the reproducibility of the performance and policy of a DRL agent. Theoretical and empirical observations point towards DRL being susceptible to nondeterminism in the training process. However, there has been no comprehensive study that isolates all individual sources of nondeterminism and studies their impact on an algorithm. It is common knowledge that exploration in reinforcement learning is important and that initializing the neural network properly is essential for adequate performance [19]. Such sources of nondeterminism as exploration and random initialization play a central role in our algorithms. In this chapter, we perform a sensitivity analysis on individual sources of nondeterminism and provide the first study in DRL to isolate these sources of nondeterminism individually and assess their impact on the policy and performance of an agent.

[1]Much of this chapter discusses ideas and results from [41].

This chapter is organized as follows. In Section 3.1, we discuss the sources of nondeterminism present in DRL, and use this to produce a deterministic implementation of deep Q-learning. Section 3.2 discusses 1) how to formulate the performance of a deterministic implementation of an algorithm as a random variable, and 2) how we can exploit deterministic implementations to improve experimentation. We conclude the chapter with Section 3.3, which discusses our experiments and findings regarding the impact of nondeterminism on the reproducibility of performances and policies.

3.1 Nondeterminism in Deep Reinforcement Learning

In order investigate the challenge posed by nondeterminism on 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.

3.1.1 Sources of Nondeterminism

The exact sources of nondeterminism depend on the algorithm, domain, libraries, and other factors. For example, a multithreaded algorithm may have additional sources of nondeterminism that are not present in a single-threaded program, due to the nondeterministic order of thread completion. That said, we identify here the sources common to most DRL algorithm implementations.

- **GPU** Neural networks are typically trained on graphics processing 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.

Deep Q-learning experiences all of the above sources of nondeterminism. A DQN is typically trained on a GPU and thus experiences GPU nondeterminism. Additionally, by virtue of being a neural network, the DQN is randomly initialized. Deep Q-learning 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 time step. The ALE was originally deterministic, however the new version of the ALE [37] introduces a form of stochasticity known as *sticky actions*. With sticky actions, the agent performs an action, but the environment (ALE emulator) executes the agent's *previous* action with probability $p = 0.25$. Thus, when the agent executes an action, it is not always the one carried out in the emulator. Lastly, deep Q-learning introduces an additional source of randomness not listed above, in the form of *no-op* (or "do nothing") actions. When the environment is deterministic (i.e. $p = 0$), the agent performs a random number of no-op actions at the beginning of each episode. This serves to randomize the initial state within the deterministic environment.

3.1.2 Implementation: Eliminating Nondeterminism

Our implementation of Deep Q-learning was written in Python using the PyTorch[2] library [46]. To control each source of nondeterminism, we did the following:

- **GPU** PyTorch exposes a modifiable boolean variable that allows us to enable or disable deterministic numerical computations on the GPU. Using this variable, we simply enabled deterministic numerical computations.

- **Environment** To remove environment stochasticity, we set the sticky action probability $p = 0$, ensuring that the agent's actions are deterministic. Take note that the original deep Q-learning algorithm [40] was trained and evaluated with $p = 0$. This conforms with the original version of the ALE [7]. However, in order to introduce stochasticity into the environment in a controlled fashion, we wrote a wrapper for the ALE, setting set $p = 0$ in the actual ALE, and using our own seeded random number generator to implement sticky actions in our ALE wrapper.

- **Policy** During training, the agent executes a stochastic policy. To control this, we designate a seeded random number generator to perform any random operations required for implementing the policy. For example, this random number generator is used to decide when to take a random action as well as which random action to take.

- **Network initialization** PyTorch permits us to set the seed for the random number generator that is used to initialize the weights of the deep Q-network, allowing us to obtain identical initial networks on separate training runs.

[2]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.

- **Minibatch sampling** Just as we do for the policy, we designate a seeded random number generator to perform any random operations required for minibatch sampling. That is, we use this random number generator to decide which transitions from the replay buffer should be in a minibatch.

No-ops are controlled in a similar way to the policy or minibatch sampling. We simply assign a seeded random number generator to generate the number of no-ops to perform at the beginning of an episode. When all these random number generators are used in tandem, we find that 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.

We utilize the network architecture from Mnih et al. [40], consisting of three hidden convolutional layers followed by a hidden fully connected layer and then the output layer. 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 $\underset{a}{\operatorname{argmax}} Q^*(s, a; \boldsymbol{\theta})$ 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. For agents trained in a stochastic environment, we anneal ϵ to 0.01 over the first million frames, after which it remains at 0.01. We make our deterministic implementation[3] available with all hyperparameters, implementation details, and experimental conditions recorded.

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

3.1.3 Replicability vs. Determinism

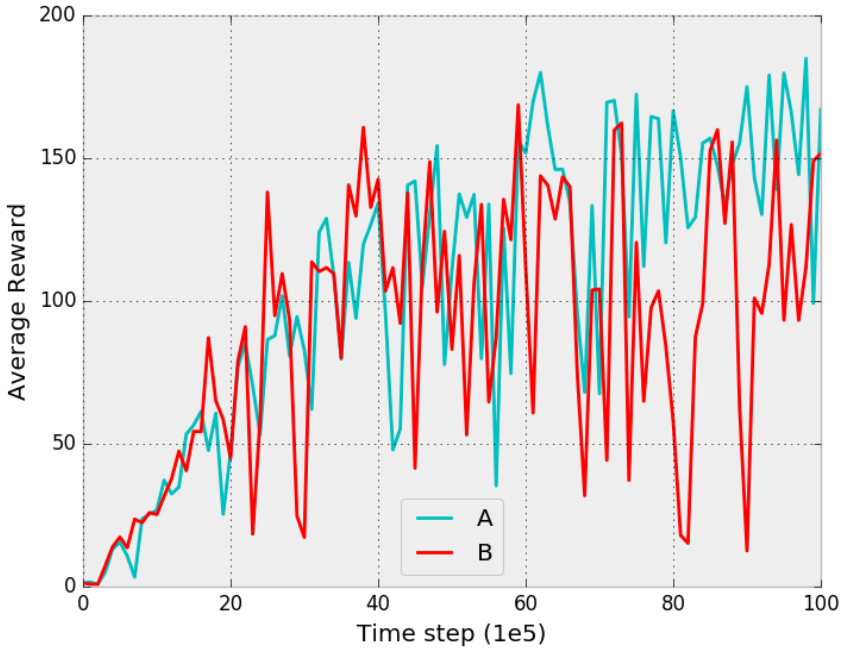


Figure 3.1: Deterministic implementation vs. replicability. The learning curves of a deterministic implementation that is executed on two separate machines.

Recall that having access to a deterministic implementation does not necessarily imply that one is able to replicate an experiment. Deterministic implementations are necessary but not sufficient for replicating results. Factors such as the software package versions, the hardware, the operating system, and other experimental conditions can all influence whether results can be replicated. The distinction is illustrated by Figure 3.1. It contains two runs of a deterministic implementation of deep Q-learning, with identical network initializations, identical random number generators, and deterministic GPU operations. However, they are run on separate machines, and thus the learning curves are very different. However, when run repeatedly on a *fixed* machine, the implementation produces identical curves. Given

the identical network initialization, we see that the curves overlap in the earliest stages of the training process but diverge soon thereafter.

3.1.4 Experimental Conditions and Replicability

The goal of this thesis is not to identify the hardware or software conditions that must necessarily be controlled to achieve replicability. However, we identify here a few representative examples of conditions that may have led to discrepancies such as the one we observe between the curves in Figure 3.1. On the software side, the deep learning library versions can influence replicability. Certain releases of TensorFlow [2] have library functions that are nondeterministic. Furthermore, to achieve determinism, in some scenarios the library must be forced to be single-threaded. According to the cuDNN documentation (cuDNN underlies many deep learning libraries), bit-wise reproducibility cannot be ensured, as implementations for some routines are changed across versions [45]. From the hardware side, we find that running the same deterministic implementation on a CPU can yield different results from running deterministically on a GPU. This can be due to several reasons [68], including differences in available operations and in precision between the CPU and GPU. Further, when a deterministic implementation is run on two different GPU architectures, it may produce different results, since code generated by the compiler is then compiled at run-time for a specific target GPU [44, 45].

It is important to note that we have identified but a few of the many experimental conditions that can impact the ability of a deterministic implementation to replicate results. The specific experimental conditions that can impact the replicability of an implementation depend on several factors and may not be limited to those catalogued here.

3.2 A Statistical Interpretation of Reproducibility and Deterministic Implementations

When studying deterministic implementations and their benefits for DRL, it may be helpful to view them from a statistical point of view. First, consider an algorithm A with k sources of nondeterminism that are modeled as the random variables $X_{A,1}, ..., X_{A,k}$. When each source of nondeterminism is fixed to a certain value, the deterministic implementation produces a single output, i.e. it implements a function f whose inputs are realizations of random variables $X_{A,1}, ..., X_{A,k}$ and whose output is a performance. As a result, we can view the performance of algorithm A as the random variable $X_A = f(X_{A,1}, ..., X_{A,k})$. To draw from X_A, we first draw $(x_{A,1}, ..., x_{A,k})$ from $X_{A,1}, ..., X_{A,k}$, and then execute the deterministic implementation f, producing an output performance $f(x_{A,1}, ..., x_{A,k})$.

To concretize this formulation, let us consider the case of a deterministic implementation of deep Q-learning. We can model the performance of deep Q-learning as the random variable X_{DQN}. We can view the sources of nondeterminism from Section 3.1.1 as the random variables $X_{ENV}, X_{POLICY}, X_{INIT}, X_{MINI}$, representing nondeterminism in the environment, policy, initialization, and minibatch sampling, respectively. Prior to running the deterministic implementation, we can view the selection of unique random seeds for these sources of nondeterminism as draws from these random variables. GPU nondeterminism is an exception in that we refrain from modeling it as a random variable. Our deterministic implementation forces the GPU to perform deterministic operations, in some sense truly *eliminating* the GPU as a source of nondeterminism. In contrast, the other random variables are *controlled* (not eliminated) with random seeds.

It seems natural that deterministic implementations have the ability to improve the quality of statistical hypothesis tests when comparing algorithms. De-

terministic implementations allow us to utilize paired difference tests such as the paired t-test and the Wilcoxon signed-rank test, which generally have more power than unpaired difference tests such as the two-sample t-test or the Welch's t-test. For simplicity, let us consider two DRL algorithms A and B with a single shared source of nondeterminism X_{INIT}, the network initialization. This means that both algorithms share the same network architecture and can draw a network initialization from X_{INIT}. Note that in practice, we find that the network initialization is a shared source of nondeterminism for several algorithms. For example, DQNs [40], Double DQNs [65], Prioritized DQNs [51], and Averaged-DQNs [4] all share the same network architecture, and thus can share the same random initialization. When we have a common source of nondeterminism such as the network initialization, we can often use paired tests, which can reduce variance and increase statistical power.

Returning to our example, suppose we have two algorithms A and B, each with a single source of nondeterminism X_{INIT}, shared by both algorithms. Let algorithms A and B have deterministic implementations g and h, respectively, such that $X_A = g(X_{INIT})$ and $X_B = h(X_{INIT})$, with means μ_A and μ_B, respectively. We define $X_{A-B} = X_A - X_B$, which has mean $\mu_{A-B} = \mu_A - \mu_B$. Recall the *one-tailed* null and alternative hypotheses for the unpaired difference tests from Section 2.5:

- $H_0 : \mu_{A-B} \leq 0$
- $H_a : \mu_{A-B} > 0$

Here, the null hypothesis states that *the difference between the mean performance of algorithms A and B is less than or equal to 0.* The standard way of performing a difference test with this null hypothesis is to obtain n realizations of algorithm A and n realizations of algorithm B: $\mathbf{x}_A = (g(x^1_{INIT}), ..., g(x^n_{INIT}))$ and $\mathbf{x}_B = (h(x^{1'}_{INIT}), ..., h(x^{n'}_{INIT}))$. We then take the samples $\mathbf{x}_A$ and $\mathbf{x}_B$ and perform an unpaired difference test (e.g. the two-sample t-test or the Welch's t-test) with the null hypothesis $H_0 : \mu_{A-B} \leq 0$. Notice how we obtained $2n$ draws from X_{INIT} ($2n$

network initializations), half of which we ran through deterministic implementation g, and half of which we ran through h. In an unpaired difference test, we do not exploit the fact that algorithms A and B share a source of nondeterminism, and thus we obtain $2n$ network initializations instead of n. Intuitively, we can see how it may be desirable to compare algorithms A and B using the *same* network initializations, as opposed to having them run under different sets of n initializations. Allowing the random network initialization to differ between the algorithms introduces noise into our comparison. This is a core idea behind *paired difference tests*. In paired difference tests, we can use only n network initializations, potentially improving our comparisons of algorithms A and B. Define the random variable $X'_{A-B} = (g - h)(X_{INIT})$, with $\mu'_{A-B} = \mu_{A-B}$. Though $\mathbb{E}[X'_{A-B}] = \mathbb{E}[X_{A-B}]$, the random variable X'_{A-B} has lower variance than X_{A-B} in many cases, in particular when $g(X_{INIT})$ is positively correlated with $h(X_{INIT})$. To exploit this property, we perform a paired difference test. Our null and alternative hypothesis are:

- $H_0 : \mu'_{A-B} \leq 0$
- $H_a : \mu'_{A-B} > 0$

In this scenario, the null hypothesis has a subtle difference from the unpaired null hypothesis. This null hypothesis states that *the mean difference in paired performances of algorithms A and B is less than or equal to 0.* We first draw n network initializations $\mathbf{x}_{INIT} = (x^1_{INIT}, ..., x^n_{INIT})$ and construct a paired sample $\mathbf{x}_{A,B} = ((g(x^1_{INIT}), h(x^1_{INIT})), ..., (g(x^n_{INIT}), h(x^n_{INIT})))$. We then perform a paired difference test (e.g. a paired t-test) on this sample of pairs with the null hypothesis $H_0 : \mu'_{A-B} \leq 0$. As long as the intra-pair correlation is positive, paired difference tests have more power than unpaired ones.

There are a few important details to be aware of. If the positive correlation within pairs does not exist, we may find that a paired test has *less* statistical power than an unpaired test. However, this typically does not occur in practice. We usually

design algorithms in an initialization-agnostic fashion. As such, we have no reason to suspect that there is a pairwise negative correlation induced by the identical network initialization. Also note that our discussion regarding the paired and unpaired tests extends to several sources of nondeterminism. For simplicity, we discussed two algorithms with a single shared source of nondeterminism. However, if several sources of nondeterminism are shared we can use deterministic implementations to control them and construct pairs between algorithms.

In this section, we have described how deterministic implementations can be utilized to improve algorithm comparisons in DRL. We can do so by identifying shared sources of nondeterminism and pairing runs of two algorithms by fixing these shared sources of nondeterminism. In doing so, we reap the advantages of paired tests over unpaired tests. While these advantages are well-known, we include them here to stress their connection to deterministic implementations and reproducibility, as studied in the remainder of this chapter.

3.3 Experiments

In this section, we perform several experiments to study the effect of nondeterminism on reproducibility. The focus of our experiments are three-fold. First, we analyze the impact of nondeterminism on the reproducibility of performance by varying a single source of nondeterminism while holding others fixed. Second, we study the impact of nondeterminism on the reproducibility of policies by measuring the frequencies with which different networks trained in the presence of nondeterminism agree on action decisions. Lastly, we empirically estimate the power of current hypothesis testing practices in DRL.

To study the impact of nondeterminism on the reproducibility of performance, we perform one-factor-at-a-time (OFAT) sensitivity analysis on the individual sources of nondeterminism identified in Section 3.1. 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 these 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.

3.3.1 Training

In our experiments, we train six groups of networks using our deterministic implementation. For each group, we train only five networks, since this is the sample size commonly used in DRL papers [37, 42]. The six groups of networks are as follows:

- **Deterministic** The "deterministic" group consists of five networks trained with identical random number generators and with deterministic GPU operations enabled across all runs. All networks in this group will be identical throughout training. The environment is kept deterministic, i.e. we set the action repeat probability to $p = 0$.

- **GPU** The "GPU" group consists of five networks trained with settings identical to the deterministic group, except with nondeterministic GPU operations enabled. All networks begin training with identical network initializations, policies, and minibatch samples. However the nondeterministic GPU operations causes differences in computations resulting in different networks upon completion of training.

- **Environment** The "environment" group consists of five networks trained with the same random number generators as the deterministic group and with deterministic GPU operations. However, we introduce stochasticity into the environment through sticky actions [37], where the agent's previous action is

repeated with probability $p = 0.25$, regardless of the agent's chosen action. Our internal implementation of sticky actions is powered by a seeded random number generator. In order to ensure that we introduce nondeterminism in the environment across our five runs, we use a different sticky action random seed for each run.

- **Exploration** The "exploration" group consists of five networks trained with different random exploration seeds and all other settings identical to the deterministic group. The DQN agent uses the seeded random number generator to perform an ϵ-greedy exploration policy.

- **Initialization** 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.

- **Minibatch** The "minibatch" group consists of five networks trained with settings identical to the deterministic group, however each of the five agents had a different random number generator used for minibatch sampling.

All of our agents are trained 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. We choose this domain due to its widespread popularity in the DRL research community. All of our agents were trained for 20 million time steps in the ALE [7] and are evaluated after every 250K time steps of training. At each of these evaluation intervals, we measure the mean and standard deviation of the agent's performance. The hardware and software conditions are held constant for all experiments.

3.3.2 Evaluation Protocol: Average Score

Typically, in the (deterministic) ALE, agents are evaluated by taking their average score over several episodes of performing an ϵ-greedy policy with some low ϵ [39, 40]. This exploration is done in order to produce diversity in the episode trajectories. Diversifying the evaluation trajectories is a way of testing 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 can be in different states, and the seeded random 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 (except when action repeat probability $p > 0$), each greedy evaluation results in the same episode trajectory, which poses an issue. We must ensure that our evaluation measures the agent's ability to generalize to different states, because a single episode trajectory is not representative of the agent's overall performance. Note that this issue specifically applies when the environment is deterministic. When evaluating agents trained in a stochastic environment, i.e. the environment group, the evaluation is simple. We have 100 random seeds which we use to implement sticky actions. Each agent in the environment group is evaluated by executing a greedy policy for 100 episodes (each episode is capped at five minutes of play), where each episode uses a unique sticky action seed to implement stochasticity. This ensures that in each episode, differences between two agents' trajectories are caused solely by differences in their decisions made within that episode. Simultaneously, by using a different random seed for each episode to implement sticky actions, we ensure that we obtain 100 different episodes.

However, for the agents in the other five groups, the question still remains: *How do we allow agents to perform greedy policies during evaluations in a deterministic environment while still testing the agents in a diverse set of conditions?* Our solution to this question is to evaluate each agent for 100 episodes, where each episode begins at a unique starting point and is capped at five minutes of play. To start each episode at a unique starting point, we assign each episode a unique predetermined action sequence for the agent to perform at the beginning of the episode to arrive at the unique starting point. After performing this predetermined action sequence, the agent then executes a greedy policy for the remainder of the five minute episode. The use of a predetermined action sequence 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 [42]. However, rather than generate our start sequences from human trajectories, we generate them computationally. To do so, we first produce 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 them may end in poor states. To remedy this, we used a trained DQN to rank our 1000 generated starts states by their maximum Q-value. We then select 100 start sequences randomly from the top 250 start sequences. While this method of generating start sequences is biased towards generating sequences that a DQN may perform well on, it enables us to generate longer sequences of random actions, which improves the diversity of our start states. We use the same 100 start sequences at every evaluation interval for all agents (except for the environment group). In this way, we are able to have a diverse set of start states that remains constant across all evaluations, while still ensuring that differences in performance are caused by the agents' policies alone.

3.3.3 Evaluation Protocol: Average Maximum Q-Values

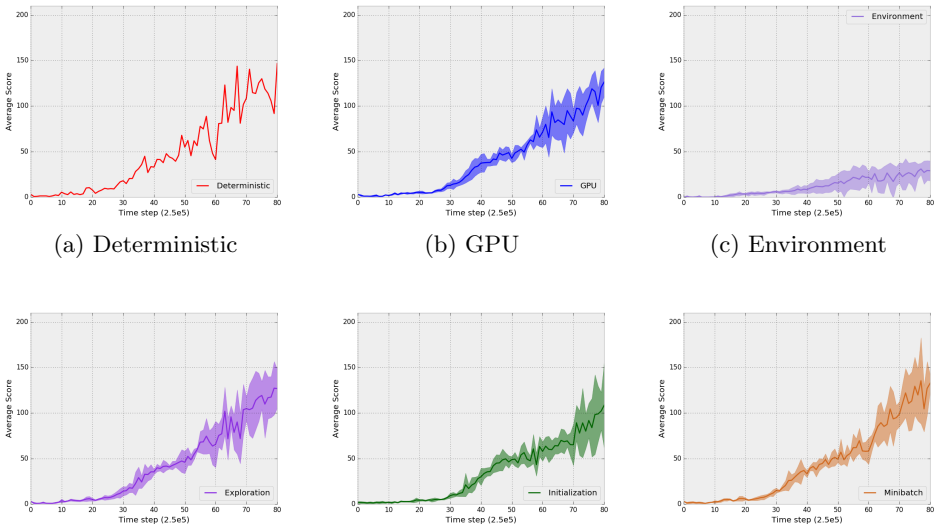
Because the learning curves for the average score can be noisy, (discussed further in Section 3.3.4), we have a second evaluation protocol that uses a more stable metric that is commonly used [4, 39, 40, 65], based off of the agent's estimated action-value function Q. In a fashion similar to prior work [39], we run a random policy for 50K time steps, 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.,

$$\frac{1}{|\mathcal{S}_Q|} \sum_{s \in \mathcal{S}_Q} \max_a Q(s, a). \tag{3.1}$$

We compute this value at every evaluation interval, using the same 500 states for all agents trained in a deterministic environment. We generate another held out set of 500 states in a nondeterministic environment, which is used for the environment group. The plots for the average maximum Q-values indicate smoother learning not seen in the plots for average score.

3.3.4 Evaluations: Results

For each experimental group of five networks, we evaluate its members under the average score and average maximum Q-value metrics. For each group and evaluation metric, we record the mean and standard deviation of the networks within that group under under the evaluation metric. Our results are summarized in Figure 3.1, Figure 3.2, Table 3.1, and Table 3.2. In figures 3.1 and 3.2, we plot the mean and standard deviation for each evaluation metric and experimental group as the agent learns. In tables 3.1 and 3.2, we report the average score and average maximum Q-values, respectively. Each table reports the performance of two types of networks, the best-scoring network and the final network. In DRL, the reported performance



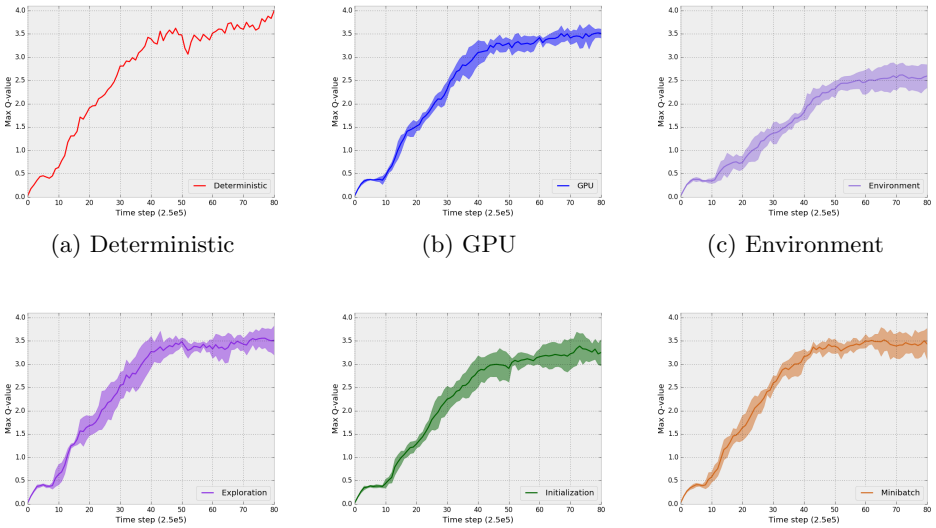
(a) Deterministic (b) GPU (c) Environment

(d) Exploration (e) Initialization (f) Minibatch

Figure 3.2: The game scores for our six experimental groups. Solid curves depict the mean score. Shaded areas represent values within one standard deviation of the mean score. The absence of a shaded area in the deterministic curves indicates that the results are identical across the five runs.

Metric	**Det**	**GPU**	**Env**	**Exp**	**Init**	**Mini**
Average Score (*Best*)	146.7	141.9	33.6	148.6	131.2	153.38
Standard Deviation (*Best*)	0.0	8.8	8.7	17.0	31.0	32.96
Relative Standard Deviation (*Best*)	0.0%	6.22 %	25.96%	11.42%	23.61%	21.49%
Average Score (*Final*)	146.7	126.5	29.0	126.9	108.6	132.84
Standard Deviation (*Final*)	0.0	15.7	10.9	21.4	47.4	8.89
Relative Standard Deviation (*Final*)	0.0%	12.41%	37.65%	16.85%	43.61%	6.69%

Table 3.1: The mean, standard deviation, and relative standard deviation of scores in BREAKOUT for six experimental groups. Note that the deterministic group has no variance.



(a) Deterministic (b) GPU (c) Environment

(d) Exploration (e) Initialization (f) Minibatch

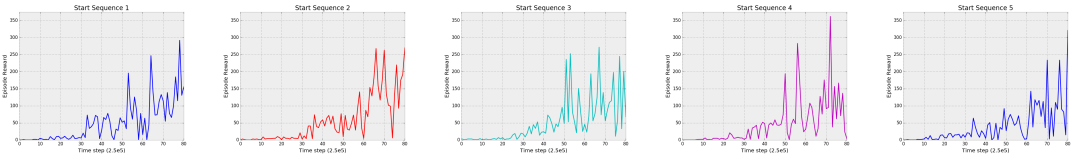
Figure 3.3: Values for Equation 3.1 plotted throughout training. We plot values within one standard deviation (shaded area) of the mean (solid).

Metric	**Det**	**GPU**	**Env**	**Exp**	**Init**	**Mini**
Average Maximum Q-value (*Best*)	4.00	3.45	2.49	3.53	3.29	3.44
Standard Deviation (*Best*)	0.0	0.081	0.269	0.305	0.204	0.36
Relative Standard Deviation (*Best*)	0.0%	2.34%	10.83%	8.63%	6.19%	10.5%
Average Maximum Q-value (*Final*)	4.00	3.51	2.60%	3.51	3.25	3.43
Standard Deviation (*Final*)	0.0	0.096	0.245	0.315	0.284	0.34
Relative Standard Deviation (*Final*)	0.0%	2.73%	9.45%	8.98%	8.73%	9.97%

Table 3.2: The mean, standard deviation, and relative standard deviation of Equation 3.1 on a set of BREAKOUT states. Values are reported for the six experimental groups.

is often the performance of the *best*-scoring network during training [40, 42, 51]. That is, the reported score is produced using the network parameters that had the highest score during training as determined by the periodic evaluations performed throughout training. In the interest of reporting results in a similar fashion to other DRL research, we too include the evaluations obtained from the best-scoring networks in tables 3.1 and 3.2. Note that we determine the best-scoring networks using the average score metric. We then use those best-scoring networks (as determined by game score) to produce the average maximum Q-values. In addition to the evaluations of the best-scoring networks, we report the evaluations of the *final* networks. That is, we report the evaluations of the final network parameters of each training run. Additionally, since the magnitude of the score and Q-values are specific to the domain (in our case, BREAKOUT), we also report the relative standard deviation to give a domain-agnostic measure of variance. We discuss the results for each experimental group individually and provide additional commentary.

Determinism



(a) Sequence 1 (b) Sequence 2 (c) Sequence 3 (d) Sequence 4 (e) Sequence 5

Figure 3.4: The score of the deterministic agent on five individual start sequences.

Figure 3.2(a) depicts the mean and standard deviation of the average scores from five deterministic runs with all sources of nondeterminism controlled. Notice how the agent's learning is not steady, with performance rising and falling for millions of time steps, oftentimes significantly. This volatility in the curve is at-

tributed to minor changes in the policy's weights, which then change the agent's state distribution [39]. In fact, when we inspected the learning curves of the agent for individual start sequences, we found that the agent's performance fluctuates drastically throughout training. In Appendix A, Figure A.1, we have included the learning curves of the deteterministic group for all start sequences. We have included Figure 3.4 here to show the average score of the deterministic agent for five example start sequences. Notice how we do not observe a stable improvement in performance, even for individual start sequences. This observation 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 offers an explanation for the lack of stable improvement on 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 on which the agent consistently improves. Figure 3.3(a) shows the more stable learning of the agent as its estimated Q-values steadily increase throughout training.

The key takeaway from figures 3.2(a) and 3.3(a) 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. This is also confirmed by tables 3.1 and 3.2, which indicate no variance in reward.

Environment

The results showing the impact of environment stochasticity are shown in figures 3.2(c) and 3.3(c), as well as tables 3.1 and 3.2. First, note that injecting stochasticity into the environment changes the transition function T of the MDP. Therefore, the environment group is performing a different task (though with the same general goal) from the remaining groups. Furthermore, we evaluate the environment

group differently from other groups, and thus we must be careful when comparing environment stochasticity to other forms of nondeterminism.

Recall that all the networks in the environment group are trained with deterministic GPU operations and share the same network initialization, minibatch random seeds, and exploration seeds. However, we allow the stochasticity in the environment to vary. As a consequence, we do not expect to obtain much variance reduction from the identical exploration and minibatch seeds, because the stochastic environments will cause each agent to have different trajectories. Two agents that have different trajectories will sample different minibatches, nullifying the benefit of sharing the minibatch seed. The common exploration seed ensures that different agents choose the same exploratory actions at the same time steps. However, since the agents experience different trajectories, they will choose the same exploratory action in different states, again nullifying the benefit of the exploration. As a result, the only enduring benefit of controlling nondeterminism in a stochastic environment comes from the common network initialization. Consequently, we expect a large amount of variance in the performance and Q-value predictions for the environment group. In fact, this is indeed the case. The relative standard deviation is 25.96% of the mean score for the best networks, which is higher than any other source of nondeterminism. While it is true that the standard deviation in score is lower than other sources of nondeterminism, it is more appropriate to consider the relative standard deviation, since the stochastic Breakout task is more difficult than the deterministic Breakout task. The large variance caused by environment stochasticity is also observed in the average max Q-value predictions, where the standard deviation in the predictions is 10.83% for the best scoring network — more than any other source of nondeterminism. Another key observation to take note of is the growing standard deviation depicted in both Figure 3.2(c) and Figure 3.3(c). The standard deviation begins at 0, as the networks are identical, and then

grows as the networks undergo different experiences and their identical initializations slowly diverge to adapt to their unique sets of experiences. Ultimately, our results show that nondeterminism in the environment can substantially impact the spread of both the score and the predicted Q-values amongst identical initial agents.

Network Initialization

The results for the initialization group are shown in Figure 3.2(e) and Figure 3.3(e). Excluding the environment group, we expect the initialization and exploration groups to have the largest variation in performance and predicted Q-values amongst all the sources of nondeterminism. Recall from Section 3.1 that DQN agents perform a large amount of exploration early in training (ϵ is annealed from 1 to 0.1 over the first million frames). Since the agents in the initialization group have identical random exploration seeds, they share many early experiences from which to learn. However, given that their Q-networks start off at different initializations, these shared experiences may affect their respective policies differently. As a result, we expect the variance to be large even in the early stages of training. However, in Figure 3.2(e), we observe that the variance is quite low at the beginning of training.

Figure 3.3(e) 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 for the low variance is that their network weights are quite different, but perform similarly in terms of game score. This may well 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 zero than there are policies that

can score 150. Consequently, different random initializations may cause different policies to perform similarly poorly early in training. The differences between these networks' weights may only be reflected in the game score in the later stages of training when the networks improve and individual differences between policies have more consequential impacts on performance.

While we do not see much variance in performance in the early stages of training, figures 3.2(e) and 3.3(e) show the usual pattern of growth in the standard deviation of performance and predicted Q-values. Furthermore, we find that the standard deviation in the scores of the best-scoring networks and the final networks is quite large. The standard deviation is 23.61% of the mean score for the best scoring networks and 43.61% for the final networks (Table 3.1). These are the largest standard deviations in score we observe for any source of nondeterminism (excluding environment). As mentioned, this is expected, because the network weights most directly influence the policy, which in a deterministic environment, is the chief determinant of an agent's experience distribution. Since experience distributions influence future policies, we get a feedback loop where the policies and experiences of different agents diverge from one another.

The random initialization of a neural network is a source of nondeterminism that is fairly simple to control with modern libraries. However, when uncontrolled, it can lead to large variations in performance, as shown in Figure 3.2(e), Figure 3.3(e), Table 3.1, and Table 3.2. These results yet again demonstrate the benefit of deterministic implementations, which can avoid these variations in performance.

Exploration

Recall that all networks in the exploration group are trained with the same network initialization, minibatch sampling seed, and with deterministic GPU operations. The exploration seed is the only varying element when training this group

of networks. While the network initialization is a source of nondeterminism whose influence on the policy takes effect throughout training, exploration takes a more immediate effect on the agent's policy as soon as training begins. Since the agent begins training with an ϵ-greedy policy with $\epsilon = 1.0$, the exploration random seed is the source of nondeterminism with the largest immediate impact on the policy. Our observations from Section 3.3.4 suggest that the portions of the state space that DQNs perform well on is constantly shifting. As such, having a large difference in policies early in training immediately cause two agents to have different experience distributions. Since this occurs so early in training, we expect this small difference to cascade through the training process.

The results for the exploration group are shown in figures 3.2(d) and 3.3(d), as well as tables 3.1 and 3.2. Indeed, these results corroborate our hypothesis. The tables show that exploration has a large amount of variance in performance. The relative standard deviation of the performance of the best networks is 11.42% of the mean score. For the final networks, we see a relative standard deviation of 16.85% of the mean score. Both from a relative standard deviation and raw standard deviation point of view, we find exploration to be more variable than other sources of nondeterminism, excluding environment stochasticity and random initialization. Interestingly, we see a larger relative and raw standard deviation of Q-values in exploration than in the network initialization. The reason behind this is unclear, and requires further investigation.

The results from Table 3.1 suggests that, as an individual source of nondeterminism, the network initialization causes more variance in the output than exploration. Intuitively, this makes sense, because the network initialization is has a longer lasting effect throughout training than a differing exploration seed. Unless the initial exploration over the first million frames is enough to change the network weights so drastically as to overcome any effects of having an identical net-

work initialization, we would expect that varying the initialization would result in longer-lasting and more apparent differences in performance.

As in with the previous sources of nondeterminism, note that figures 3.2(d) and 3.3(d) exhibit the recurring signs of cascading differences: 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.

Minibatch Sampling

The minibatch group consists of five networks, all with the same random network initialization, exploration seeds, and with deterministic GPU operations. The only difference in the settings of each of these training runs is that each run uses a unique minibatch random number generator for sampling from the replay buffer. With the exception of the GPU, we expect the minibatch group to have the lowest variance in performance between training runs. Each training run starts with the same policy which generates the same initial experiences. Furthermore, since exploration is quite large at the beginning, we expect the replay buffer to be filled with plenty of identical experiences. Of course, once learning begins, the different minibatch seeds will cause separate training runs to sample different experiences from the replay buffer. However, in expectation, each experience in the replay buffer is sampled eight times for training. Thus we believe that, despite the fact that minibatches contain different experiences across runs, that the large replay buffer will ensure that many of the same experiences are sampled across runs, although in a different order. The purpose of the large replay buffer is to make the learning problem more stationary, which directly ameliorates the cascading effect. Given the large size of the replay buffer, we can loosely think of the minibatch nondeterminism as similar to a supervised learning problem, where different training runs sample different

minibatches from a stationary dataset. Of course, this analogy does not truly hold, because the replay buffer is still nonstationary to some degree.

The results for the minibatch group are shown in figures 3.2(f) and 3.3(f). As per usual, we can visualize the low variance between the learning curves early in the training process, followed by greater variance later on. However, the results we observe for the minibatch group in Table 3.1 are quite surprising. We see that the standard deviation in performance is 32.96 for the best scoring network, dwarfing the standard deviations of the remaining sources of nondeterminism. However, upon closer inspection of the data, we found that one of the best-scoring networks for the minibatch group was an outlier, scoring over 200. Notice how the average score of the best scoring network for the minibatch group is larger than any other experimental group. This is due to the outlier. Furthermore, notice how the standard deviation of the final networks is much smaller, at 8.89. However, despite this outlier in score, the average maximum Q-value metric from Table 3.2 suggests that minibatch sampling has a greater impact than hypothesized. The relative standard deviation in the final network's Q-value predictions is in fact greater than that of exploration or initialization.

The outlier we observed in the data illustrates another important reproducibility issue in DRL. Individual runs of DQNs tend have noisy curves, generally speaking. When learning curves fluctuate up and down, we may find the best scoring network to be an outlier as in this scenario. As a result, we observe large variance in just five samples, for a relatively innocuous source of nondeterminism.

GPU

The results for the GPU group are shown in Figure 3.2(b), Figure 3.3(b), and our two tables. 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 anticipate that it will be minimal. We find that when the weights of a neural network are updated with nondeterministic GPU operations, the weights between two identical networks to desynchronize after that single update of the weights. This is expected, since the neural network has thousands of parameters. However, it takes time for the small differences in these weights to aggregate to a point where the differences affect decision-making. Early in training, the weights are similar enough across training runs to where the same exact actions are performed for tens of thousands of time steps. Even after the policies deviate for the first time, it takes more time before these policies are substantially different. As such, given the identical initial policies and random seeds, the agents' replay buffers will contain highly similar experiences, at least early in training. Consequently, we would expect the learned policies to be similar, resulting in similar performance.

Viewing Table 3.1 and Figure 3.2(b), we see that indeed the variation caused by the GPU is minimal relative to other sources of nondeterminism. We find that the relative standard deviation in score of the best scoring networks is near 6%, and the relative standard deviation in score of the final networks is 12.41%. Figure 3.2 and Table 3.3(b) show that the relative standard deviation in the predicted maximum Q-values is small, near 2% for both network types.

However, as anticipated, we observe the cascading effect in Figure 3.2(b) and Figure 3.3(b), where the variation is extremely small for several million frames before growing larger and larger. This results in the 12% deviation from the mean final score, which is quite a large deviation considering the starting conditions are identical and the variation can be attributed to nondeterministic operations. The GPU is an excellent example of the cascading effect: minor differences in computa-

tion cause differences that compound as training progresses, resulting in outcomes that become increasingly different with time.

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 underscores the benefit of deterministic implementations. Even if implementation details are successfully reproduced, sources of nondeterminism outside of the algorithm can impact results.

Conclusions

Our experiments on the reproducibility of performance due to individual sources of nondeterminism have led to several conclusions, considerations, and hypotheses, some of which we list here.

- **Conclusion** We have demonstrated that deterministic implementations can produce evaluations with zero variance and can enable replicability if the experimental conditions are fixed.
- **Conclusion** Random initializations, network initializations, and environment stochasticity are sources of nondeterminism that can cause large variations in the results.
- **Conclusion** Minor sources of nondeterminism can start a cascading effect throughout the training process.
- **Conclusion/Observation** For all individual sources of nondeterminism, the learning curves exhibit low variance early in training. The variance tends to grow as training progresses.
- **Consideration** Given the complex interactions between sources of nondeterminism, it may be safe to permit certain sources of nondeterminism in the

training process without substantially impacting the performance. However, given that we found that certain sources of nondeterminism cause a large variance in performance, minor sources of nondeterminism may serve to swell the variance we observe from major sources of nondeterminism.

- **Consideration** Given the observed trends in the learning curves, we may find that the variance in outputs will grow if we run these algorithms for a greater number of time steps.

- **Consideration** Our results are in some sense an optimistic perspective of what we might see in practice. It is important to realize that our evaluation methodology is designed so that any observed variation in performance is attributable to variations between policies and policies alone. It may be the case that individual sources of nondeterminism have higher variance in performance if we utilize traditional ϵ-greedy evaluations, which introduce nondeterminism to the evaluation process.

- **Hypothesis** There may be several causes of the increasing variance in performance we observe as training progresses. One cause may be the cascading effect, as in the case of the GPU. However, another cause may be that lower performing agents are quite different decision-makers, but the variation in decisions is not reflected in performance.

3.3.5 Policy Similarity

To study how individual sources of nondeterminism affect the reproducibility of a policy, we study the policy *similarity* within a group of agents. To do so, we generated a held out set of 100K states from Breakout, and have each of our groups of networks from Section 3.3.1 select the greedy action for each state in this held out set of states. For each state, we record the maximum number of selections

given to any one action, and tally up the results. Suppose for a particular state we find that, between five agents, the maximum number of selections given to any one action is two. We refer to this scenario as a 2/5 *agreement*. Through this section, we will refer to such scenarios in this fashion. For each source of nondeterminism, we produce pie charts representing the distributions of agreements on the held out set of states.

State Generation

We generate two held out sets of states. One set of held out states is generated in a deterministic version of BREAKOUT, and the second is generated in the stochastic version of BREAKOUT. To generate the deterministic set of states, we first train a DQN using a set of random seeds different from any used by the networks from our experimental groups. The different seeding is done in order to ensure that our held out set of states is not biased towards areas of the state space that our agents may visit. This DQN is trained in a deterministic environment using the same training specifications from Section 3.3.1. To generate the held out set of states, we have this agent play episodes using an ϵ-greedy policy with $\epsilon = 0.05$, beginning from the start states described in Section 3.3.2. We record the states that the agent visits until it has observed 100K states, which we use as our held out set of states to evaluate the policy similarity of the experimental groups trained in a deterministic environment. The held out of states for the stochastic version of BREAKOUT is generated in a similar fashion. We train an agent in the stochastic environment using different random seeds from any used in the environment group. We then have this agent perform several episodes using a greedy policy, until it experiences 100K states, which we record and use to construct our second set of held out states.

One issue that arises from our method of generating held out sets of states is that the states generated within the same episode are correlated. As a result,

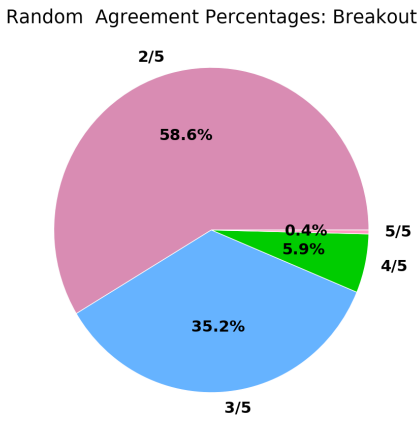
whether or not the agents in an experimental group agree on an action in a particular state is highly correlated with whether they agree on the action in the subsequent state in the episode. Despite this drawback, our method ensures that we obtain states from all portions of an episode. Perhaps in the future, we can have several policies generating states, and only select states for our held out set that are temporally far from one another within an episode.

Computing Similarity

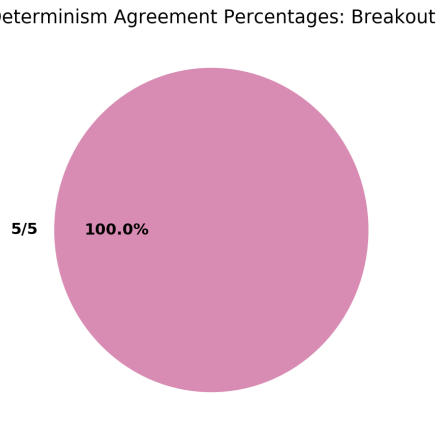
To construct our pie chart depicting the policy similarity of agents within a single group of five networks, we iterate through all 100K states from the held out set of states, and for each state we record the highest number of selections any one action gets from the five networks. For example, suppose for 50K of the states, we have a 2/5 agreement, for 40K of the states we have a 3/5 agreement, and for the remaining 10K states, all agents agree on the same action. This results in a pie chart with three categories: $\{2/5 : 0.5, 3/5 : 0.4, 5/5 : 0.1\}$. In this fashion, we construct two pie charts for each experimental group: one pie chart for the best scoring networks and another for the final networks.

Similarity Results

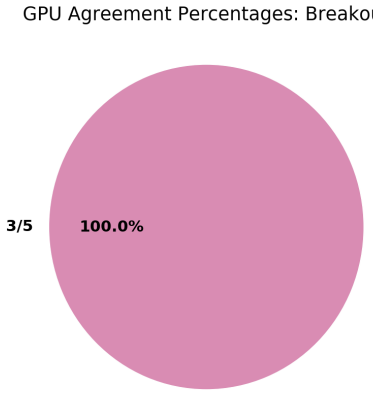
Notice Figure 3.5(a), which shows the expected agreement for five random policies. Since the minimal action set in Breakout has four actions, and we are measuring the agreement of five networks, then by the pidgeonhole principle, we will always have at least one action with two or more selections. We find that under five random policies, we have a majority agreement 41.4% of the time, which is quite often. Given that random agents perform quite poorly on Breakout, this suggests that five trained agents should have a majority agreement on actions far more than 41.4% of the time. However, it may be the case that trained agents, given their differing



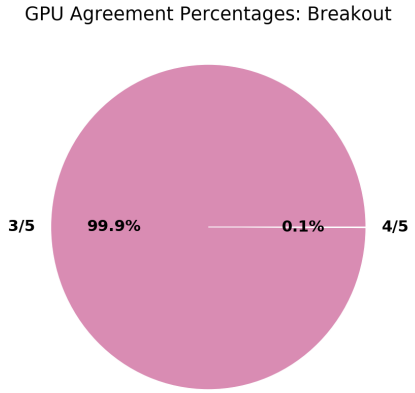
(a) Expected Policy Agreement: Random



(b) Policy Agreement: Deterministic

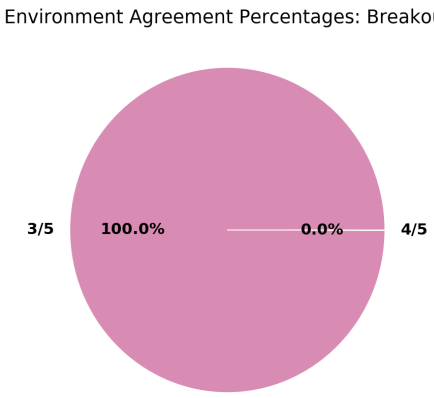


(c) Policy Agreement: GPU - Best

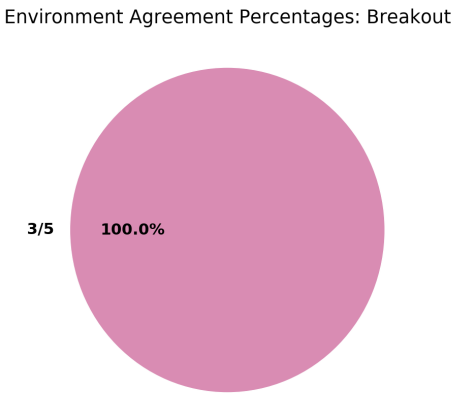


(d) Policy Agreement: GPU - Final

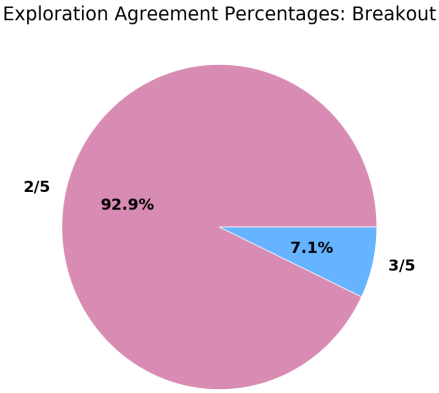
Figure 3.5: Policy agreements on a held out set of states for different sources of nondeterminism.



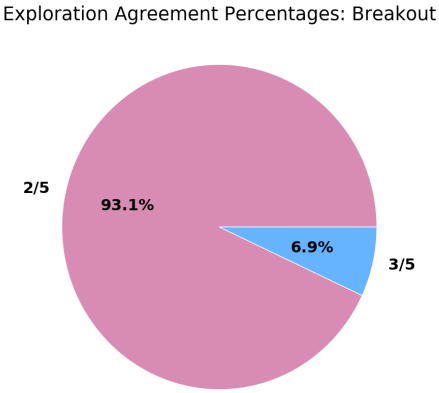
(e) Policy Agreement: Environment - Best



(f) Policy Agreement: Environment - Final

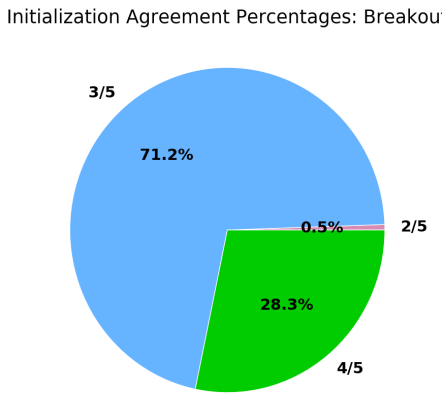


(g) Policy Agreement: Exploration - Best

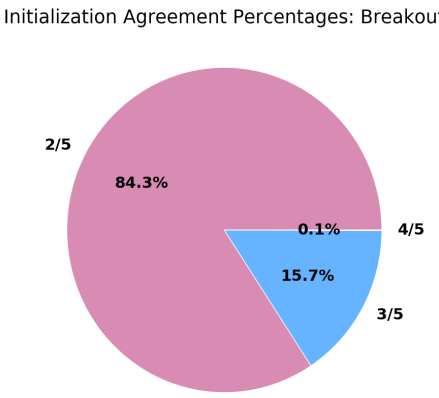


(h) Policy Agreement: Exploration - Final

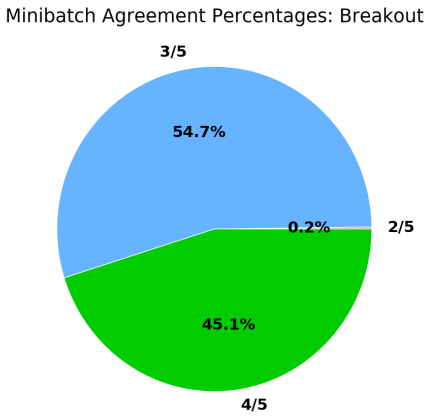
Figure 3.5: Policy agreements on a held out set of states for different sources of nondeterminism.



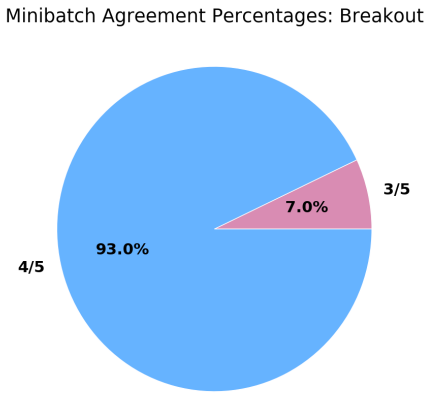
(i) Policy Agreement: Initalization - Best



(j) Policy Agreement: Initalization - Final



(k) Policy Agreement: Minibatch - Best



(l) Policy Agreement: Minibatch - Final

Figure 3.5: Policy agreements on a held out set of states for different sources of nondeterminism.

policies, only agree on actions in very important states (e.g. a life-or-death scenario in a game). In considering the reproducibility of a policy, on one end of the spectrum we have a deterministic agreement pie chart as in Figure 3.5(b), with full agreement on all 100K states. This is the ideal scenario of reproducing a policy. The random pie chart, in contrast, depicts the other end of the spectrum, a non-reproducible policy. Given that even random networks achieve a majority agreement much of the time, we expect trained policies to have a majority agreement for most states. If a source of nondeterminism hinders policy reproducibility, we expect more 2/5 agreements.

Figure 3.5 contains the results for all groups, and its implications are unclear. Previously, we found the GPU group to have low variance in performance and predicted Q-values. Consequently, we expect that given similar value predictions, similar performances, *and* identical initial conditions, that there would be many 4/5 or 5/5 agreements. However, figures 3.5(c) and 3.5(d) have virtually no 4/5 or 5/5 agreements. On the other hand, they do indicate that the GPU group achieves a majority agreement nearly 100% of the time. The exploration results from figures 3.5(g) and 3.5(h) are quite shocking. In fact, they suggest that exploration strongly impacts the reproducibility of a policy, to the point where the policy similarity of the group is worse than random. While we expect that exploration may impact the policy similarity, surely as group of trained agents we would expect them to show some amount of similarity better than random. The environment stochasticity results from figures 3.5(e) and 3.5(f) are somewhat counterintuitive as well. We saw earlier that the environment stochasticity caused plenty of variance in both performance and in Q-value predictions. It may be the case that a stochastic environment forces the agents to learn a more specific optimal policy, driving agents towards similar policies, even if their performances differ. However, this hypothesis cannot be validated without further study. Minibatch sampling shows a very high

degree of policy similarity, which we expected in spite of the high variance we observed in performance and Q-value predictions. Members of minibatch group share initial weights and initial exploration, which leads to similar replay buffers. Since the replay buffer is large, it minimizes the nonstationarity of the distribution over which we sample minibatches. This enables the policies to remain roughly the same across runs throughout training. The policy similarity results for the initialization group, shown in figures 3.5(i) and 3.5(j), are somewhat contradictory. The best networks show a high degree of policy similarity, while the final networks exhibit a policy similarity worse than random. When we studied the variance in performance, the final networks of the initialization group had an extremely large standard deviation in performance, at 43.61% of the mean, which was the largest of all the groups. Naturally, this drastic a difference in performance corroborates the fact that we observe a low policy similarity within the group. However, the best scoring networks in the initialization group *also* had high variance, and yet we observe a high degree of similarity.

In summary, we find the policy similarity results to be inconclusive. It seems many aspects of policy similarity remain black boxes that require further investigation. The policy similarity/reproducibility results we show here appear to be uncorrelated to our results on the reproducibility of performance. Furthermore, even within the same source of nondeterminism, we have somewhat contradictory results, such as with the network initialization group. There may be several causes for these counterintuitive results. For one, we have observed that DQNs perform well on certain start sequences but poorly on others. It may the be case that the 100K states we generate come from a policy and start sequences that either happen to align well with certain networks and poorly with others. Furthermore, correlations between states in the held out set may cause the similarity results to skew in favor of similarity or dissimilarity, without any middle ground. Furthermore, it

is difficult to reason about the magnitude of the results we expect. For example, we do not have a strong notion of what makes one pie chart exhibit a greater degree of policy similarity than another. In the future, a more in depth study can utilize a more formal diversity measure [60], as opposed to a visual metric such as a pie chart. Additionally, we can construct larger and more diverse held out data sets, without using entire episode trajectories in our set of states. The larger data set provides more diversity in state space, and removing episode trajectories can decorrelate states within our data.

3.3.6 Type I and Type II Error

An important component of analyzing the impact of nondeterminism on the reproducibility of research results involves studying its effect on statistical hypothesis tests. In this subsection, we empirically estimate the Type I and Type II errors that result from a nondeterministic implementation of deep Q-learning, while using a sample size of five, as is customary in DRL [37, 42]. To empirically estimate the Type I error, we adapt the procedure described by Colas et al. [12], which we described in Section 2.5. In that procedure, to estimate the Type I error, we produce $2n$ trials of a single algorithm, and split them into two groups a and b. We then perform a difference test on the two groups as though the groups were generated by different DRL algorithms. If the difference test finds group a to have statistically significantly (for some significance level α) higher performance than group b, then it is a false positive result, since we know the null hypothesis to be true. We repeat these steps thousands times, and the proportion of false positives we find is our empirical estimate of the Type I error.

When the Welch's t-test's assumptions are met, the probability of type I error is the desired significance level α [12]. However, oftentimes we employ statistical tests while deviating from their underlying assumptions (e.g. having non-normal

data). While we expect that performing a Welch's t-test with deep Q-learning satisfies the test's assumptions, we would like to measure the Type I error empirically and check whether it matches our significance level α, using samples consisting of five runs of the deep Q-learning algorithm. However, due to the large computational requirements of DRL, it is prohibitively expensive to repeatedly perform the steps from Colas et al.'s procedure thousands of times. Consequently, we modify this procedure to make estimating the Type I error more tractable. First, we train on smaller DQNs, following the architecture outlined by Mnih et al. [39], with two convolutional layers, a fully connected layer, and an output layer. Additionally, we train the DQNs for 10 million frames as opposed to 20 million frames. To simulate nondeterministic DQN training runs, we allow the GPU to perform nondeterministic operations, and seed each DQN's random number generators with a unique set of seeds. Each DQN represents a draw from the random variable X_{DQN} modeling the performance of the deep Q-learning algorithm.

However, we still run into the issue of having to repeat the procedure thousands of times, and even with our smaller network architectures, we cannot do this. To resolve this, we run 20 nondeterministic DQNs with performances $(x^1, ..., x^{20})$. Since we are measuring the Type I error when $n = 5$, we sample 10 DQNs from our group of 20 *without replacement* at each iteration of the procedure. This differs from traditional bootstrapping with replacement, but each subset of size 10 from our group of 20 DQNs is a sample from the true distribution of X_{DQN} [48], which achieves the goal of the original procedure.

We then perform the following modified procedure to produce our Type I error estimate:

1. Sample 10 DQNs without replacement from our original sample of 20 DQNs.

2. Randomly split the 10 DQNs into two groups a and b, each of size 5.

	Best Network	Final Network
Type I error	4.46%	4.3%

Table 3.3: The Type I error (false positive rate) for deep Q-learning.

3. Perform a Welch' t-test with signifiance level $\alpha = 0.05$ on the two groups, testing the alternative hypothesis: $H_a : \mu_{A-B} > 0$. Record whether or not the t-test rejected the null hypothesis.

4. Repeat steps 1-3 30K times.

5. The empirical Type I error is the proportion of times that the null hypothesis was rejected.

Table 3.3 contains the results. We observe for both network types ("best" and "final") that the false positive rate is slightly less than the significance level of $\alpha = 0.05$, though it is generally near our desired significance level. These results are consistent with those of Colas et al., who had reported a similar empirical Type I error (though slightly higher) for another DRL algorithm, DDPG [34], using the Welch's t-test, at a sample size of five.

We are also interested in measuring probability of a Type II error, or false negative rate. We typically want the false negative rate to be $\beta = 0.2$. That is, if a difference exists between algorithms, we wish to detect it at least 80% of the time. However, the false negative rate depends on the magnitude of the difference between the performances of two algorithms. For example, if the difference between the performances of two algorithms is large, then it may be easier to detect without error. The magnitude of the difference between the mean performances of the algorithms is called the *effect size*. We empirically measure the false negative rate given an effect size ϵ, sample size, and significance level α, using the following procedure:

1. Sample 10 DQNs without replacement from our original sample of 20 DQNs.

2. Randomly split the 10 DQNs into two groups a and b, each of size 5. To each DQN in group a, add the effect size ϵ to its performance. This ensures that "Algorithm A" performs ϵ better on average than "Algorithm B".

3. Perform a Welch's t-test with significance level $\alpha = 0.05$ on the two groups, testing the alternative hypothesis: $H_a : \mu_{A-B} > 0$. Record whether or not the t-test rejected the null hypothesis.

4. Repeat steps 1-3 30K times.

5. The empirical Type 2 error is the proportion of times that the test *failed* to reject the null hypothesis.

In our experiments, we use a sample size of 5, $\alpha = 0.05$, and we test for effect sizes $\epsilon \in \{0.04\widetilde{\mu}, 0.08\widetilde{\mu}, ..., \widetilde{\mu}\}$, where $\widetilde{\mu} = \frac{\sum_{i=1}^{20} x^i}{20}$, the mean of our 20 DQN performances. We considered other alternatives for adding an effect size to group a. For example, we considered drawing five numbers from a Gaussian distribution with mean ϵ and adding these drawn values to the scores of group a. The result of this method is that the performance of "Algorithm A" has more variance (due to the Gaussian noise added to group a), making it more difficult to detect a statistically significant difference. Our method of adding the effect size ensures that "Algorithm A" and "Algorithm B" have the same variance. Essentially, the performance distribution of "Algorithm A" is the performance distribution of "Algorithm B" shifted by effect size ϵ.

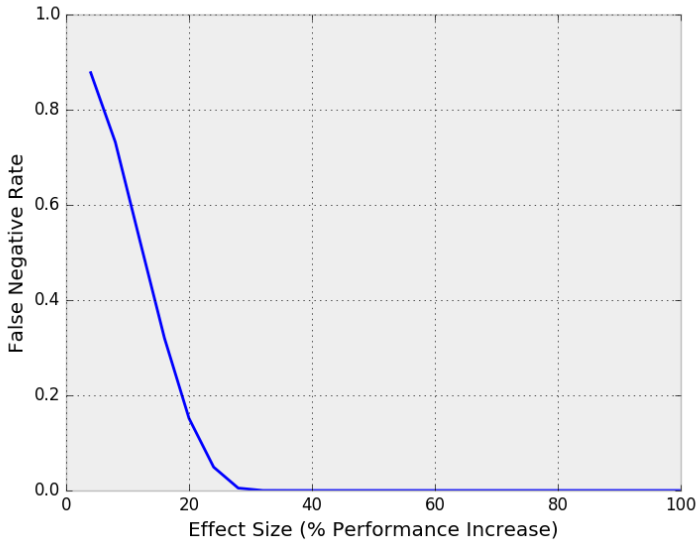


Figure 3.6: The false negative rate for various percentage improvements upon deep Q-learning at $\alpha = 0.05$ and a sample size of 5.

Figure 3.6 depicts our results. We find that at a significance level of $\alpha = 0.05$, using a sample size of 5 on the BREAKOUT domain, we are not able to achieve the desired false negative rate $\beta = 0.2$ unless Algorithm A exhibits an 18% improvement upon the original DQN algorithm. BREAKOUT is a relatively popular domain used in research, and it may be the case that due to the low power at the sample size of 5 and effect size of less than 18%, researchers may fail to detect their own success. Perhaps more shockingly, if an algorithm surpasses deep Q-learning by 10% on this domain, there is over 50% chance that a researcher will fail to detect any statistically significant improvement!

Deterministic implementations can offer a solution to the low statistical power we observe at smaller effect sizes. As discussed in Section 3.2, paired t-tests generally have greater power than unpaired t-tests, allowing us to detect smaller effect sizes at the same sample size.

Chapter 4

Related Work

In this chapter, we discuss prior work on reproducibility in computer science, AI, and DRL, as well as related research on the ALE.

4.1 Reproducibility in Computer Science and Artifical Intelligence

AI is currently suffering from a reproducibility crisis. Recent work by Gundersen and Kjensmo [22] reviews the state of reproducibility in AI. Gundersen and Kjensmo identify a comprehensive set of variables that are reflective of reproducible papers, including the availability of datasets used, source code, hyperparameters, hardware specifications, software dependencies, and research method. Using these variables, they analyze the documentation practices of hundreds of papers published at top AI conferences in recent years. Their study finds that most papers do not provide these variables, leading them to conclude that most research papers in AI are not reproducible.

4.1.1 Reproducibility vs. Replicability

There has been some debate within the AI community regarding reproducibility and replicability. This debate has been addressed by Drummond [16], who argues that replicability is a poor substitute for reproducibility in ML. He believes that replicability is not a worthy goal in the context of scientific reproducibility and that achieving replicability would be too time-consuming for the community. He asserts that a major point of reproducible research is that irrelevant details are deliberately not replicated. That is, results should be able to be reproduced by fixing the relevant details and allowing irrelevant details to vary. However, the problem with this argument is that in the modern day, we find it difficult to distinguish between the relevant and irrelevant details, as evidenced by the results of Henderson et al. [26], who show that different implementations can vary substantially in performance. In fact, replicability can be *less* time-consuming than reproducibility, since it can be much more difficult to comprehensively identify all the relevant details required for success than to share a deterministic implementation and experimental conditions.

4.1.2 Replicability and Experimental Conditions

There have been some efforts to identify or study the experimental conditions that can affect the outcome of computational experiments. For example, Gronenschild et al. [20] analyze the results produced by a software package called FreeSurfer [18], which is used to make measurements in studies of neuroanatomical structures. They test measurements produced while using FreeSurfer under different experimental conditions. They vary the FreeSurfer version, the operating system, and the workstation (hardware), and find that results can vary significantly when experimental conditions are changed. They conclude that within a single study, a consistent set of experimental conditions should be used.

4.1.3 Reproducibility Efforts

In spite of the reproducibility crisis that the AI and scientific community faces, there have been efforts made towards performing reproducibility research. For example, researchers and academic journals have encouraged wider dissemination of code and datasets with publications [47, 50, 54, 55, 56]. Furthermore, several ML labs and institutions across the world have made datasets and environments publicly available, enabling better benchmarking, comparison, and reproducibility of algorithms. For example, OpenAI has released OpenAI gym [11], which has several reinforcement learning and robotic tasks that researchers can use to develop and compare algorithms. The ALE [7] was one of the first DRL evaluation platforms, making dozens of Atari games available for use by researchers. Google Deepmind has released Deepmind Lab [5], a system that has several three-dimensional environments that can be used for AI research. They have also released the Deepmind Control Suite [61], an environment for control tasks built off of the MuJoCo engine [62]. However, in addition to simulated environments, there have been efforts to release datasets for research. The Imagenet dataset [14] has played a central role in computer vision research. Other datasets such as the CIFAR-10 dataset [31] and the UCI Machine Learning Repository have played important roles in ML research as well.

We have also seen the development of reproducibility-friendly software — software that minimizes the difficulty that researchers face in achieving reproducibility. For example, AWS Docker containers [3] and CodaLab [1] can be used to achieve replicability by packaging the experimental conditions with the code. Sumatra [13] is a tool that can be used to control dependencies and assist with version control in reproducible research. CDE (Code, Data, Environment) [23] packages software dependencies that are needed to rerun Linux-based experiments on other machines. Lastly, Jupyter notebooks [30] enable researchers to readily have code with explanations packaged together. Further, these notebooks can run inside containers (e.g.

Docker), permitting replicability.

Overall, there have been several efforts within AI, computer science, and other computational fields to achieve reproducibility. These include efforts pushing for broader sharing of code and data, efforts to release datasets and environments for research, and efforts to develop software tools that enable experimental conditions to be more easily controlled and shared universally.

4.2 Reproducibility in Deep Reinforcement Learning

While reproducibility has been explored across artificial intelligence, machine learning, and robotics [9, 16, 21, 22], 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 [26, 29, 37]. Henderson et al. [26] show that the choice of hyperparameters, network architecture, reward scale, and random seeds can have a dramatic effect on the performance of an agent. They show that some algorithms perform better than others in environments with stable dynamics, while performing worse in environments with unstable dynamics. They also find that differences in implementation details between codebases implementing the same DRL algorithm can result in drastically different performances. Perhaps their most shocking result is that two groups of networks trained from the same algorithm implementation can yield statistically significantly different performances solely due to differences in global random seeds.

The work in this thesis differs from prior research in that it investigates individual sources of randomness, as opposed to the aggregate effect of random seeds. Furthermore, this thesis studies the impact of injecting environment stochasticity into a task as opposed to comparing a stochastic task to a deterministic task when the task semantics are inherently different. Further, we study value-based methods as opposed to policy gradient methods, which have been the focus in prior work.

Finally, previous work done in the context of DRL has focused primarily on the broader notion of reproducibility, whereas we emphasize the benefit of determinism, which spans both reproducibility and replicability.

Reproducibility in DRL has also been examined from the perspective of statistical hypothesis testing by Colas et al [12] (as in Section 2.5). Their focus is on describing the best statistical practices for algorithm comparison (e.g. showing the superiority of a new algorithm). This is particularly relevant in the modern day when the state-of-the-art in DRL is continually moving forward. They formulate the performances of DRL as random variables and address how to perform hypothesis tests to compare two DRL algorithm performances. They go on to review the different statistical tests that can be used, discuss their assumptions, and derive general guidelines. Additionally, they discuss the influence of deviating from the assumptions of the statistical tests, which often occurs in practice. Lastly, they address the important problem of selecting the appropriate sample size when performing a DRL hypothesis test.

While this thesis focuses on how nondeterminism in a program can cause instability or variance in practice, there have been other efforts to identify and address sources of variance or instability in DRL from more theoretical or algorithmic perspectives. For instance, Double DQN [65] proposes an extension to DQN that helps mitigate overestimations of DQNs. Averaged-DQN [4] also proposes an extension to DQN that reduces the variance in the target approximation error which helps with stabilization.

4.3 The Arcade Learning Environment

While not necessarily in the domain of reproducibility, there has been research centered around the effects of hyperparameters and determinism within the ALE. For instance, it was shown that the frame skip hyperparameter commonly used in

DRL algorithms can strongly influence a learning agent's success in the ALE [10]. Since the ALE was originally designed to be deterministic, agents can succeed simply by memorizing action sequences. Naturally, we only want agents that learn robustly to be able to succeed. As a result, several methods of injecting stochasticity into the environment [24, 37] have been proposed to address this issue. The goal of stochasticity injection is to thwart a memorizing agent (causing it to fail) without harming an agent that learns and generalizes robustly. 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 in this thesis. Many of the findings in the ALE have been synthesized into best practices for the DRL community [37], with the hope of setting the standard for research to follow.

Chapter 5

Conclusion and Future Work

In this thesis, we advocated for deterministic implementations as a partial solution to the reproducibility crisis in DRL. We identified the sources of nondeterminism present in DRL and described a deterministic implementation of the deep Q-learning algorithm. We then described how deterministic implementations can be used with paired tests to improve algorithm comparisons. We then performed an OFAT sensitivity analysis of different sources of nondeterminism and measured the variance caused by each source on the resulting performance of the agent. We found that individual sources of nondeterminism can substantially impact the performance of a DRL agent, reaffirming the benefit of deterministic implementations. We then studied the effect of nondeterminism on the reproducibility of policies in DRL, and found that in some scenarios, agents trained nondeterministically have differing policies. However, in other scenarios we found that agents had similar policies. Furthermore, the impact of nondeterminism on policy similarity did not appear to be directly related to the impact of nondeterminism on performance. Lastly, we empirically measured the Type I and Type II error when performing algorithm comparisons in deep Q-learning. We found that the Type I error was near our target significance level α, and we quantified the effect size for which standard DRL algorithm

comparison tests would not achieve our desired Type II error $\beta = 0.2$. Ultimately, given the large impact of nondeterminism on results, the benefit of deterministic implementations becomes apparent. 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.

There are several potential avenues for future work beyond this thesis:

- The conditions for which paired difference tests improve statistical power are well known. However, the extent to which we can benefit from paired difference tests has yet to be empirically tested in DRL.

- In this thesis, we investigated the single domain of Breakout. However, in the future, we hope to extend our sensitivity analysis to more domains, since the impact of nondeterminism may be domain-dependent.

- In this thesis, we investigated deep Q-learning. However, there have been several algorithms developed that aim to reduce variance during learning. It will be interesting to study the impact of nondeterminism on these more robust algorithms.

- In this thesis, we briefly touched upon some experimental conditions that can impact replicability. However, it would be interesting to see a comprehensive study of experimental conditions that can cause results to vary.

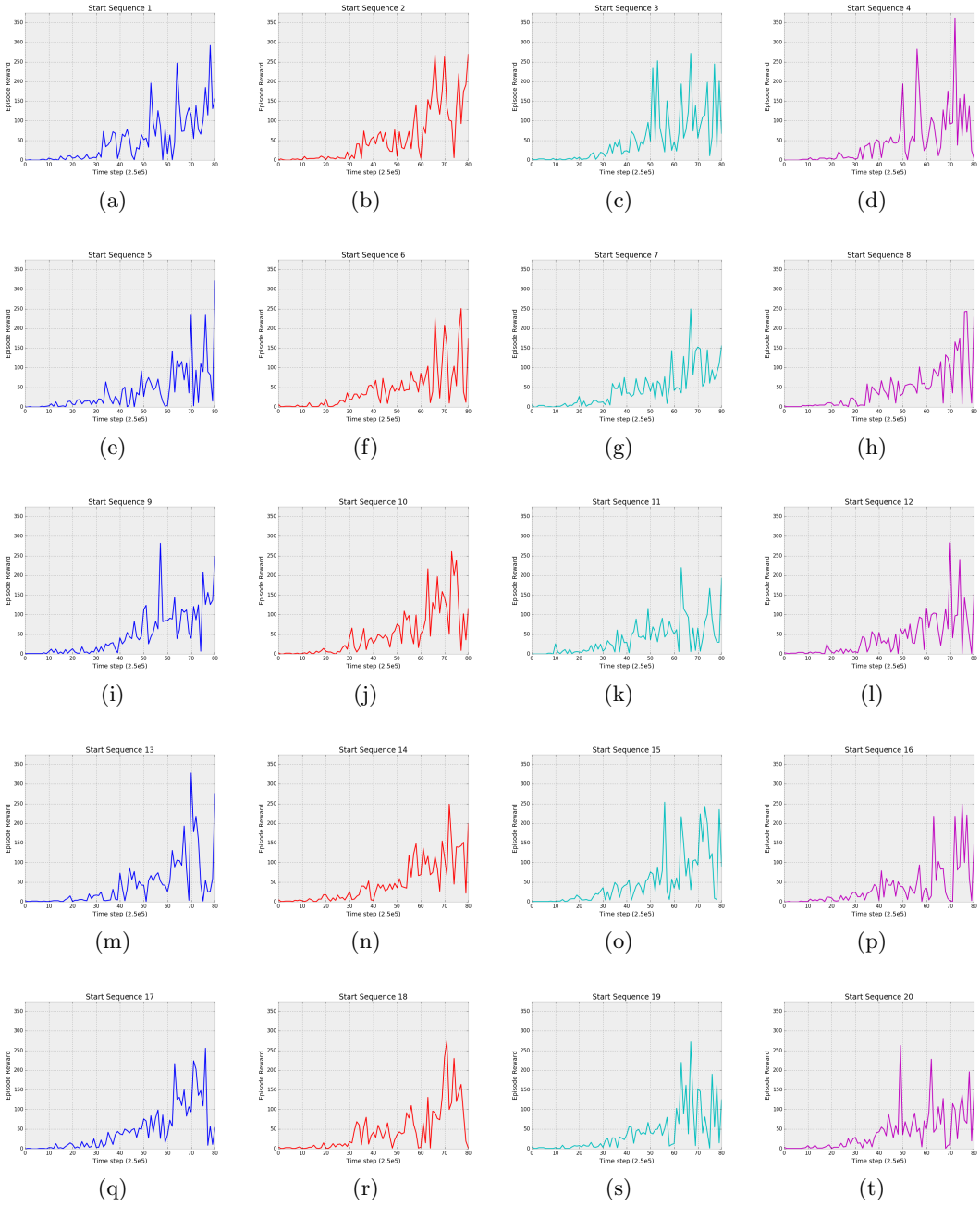
- In this thesis we performed an OFAT sensitivity analysis, studying each source of nondeterminism in isolation. However, it would be interesting to see how different sources of nondeterminism interact, and whether it increases the variance in performance.

Looking forward, we hope that the DRL community sees the benefit of utilizing deterministic implementations. Further, we hope that researchers are now informed of the different sources of nondeterminism at play in DRL, and are cognizant of them when producing research.

Appendix A

Supplemental Material

In Figure A.1 we include the learning curves of a deterministic implementation of deep Q-learning. Each graph represents the agent's score after performing a specific action sequence to start the episode. The key feature consistently observable across all of these curves is the volatility of the agent's performance. We see that as the agent learns, its performance on individual start states fluctuates heavily. The general trend of increasing performance we see for a DQN is based off of an average of these 100 curves. However, even if we observe a general improvement in average score, the agent becomes drastically worse on specific start sequences. For further discussion, see Section 3.3.4 earlier in this thesis.



(a) (b) (c) (d)

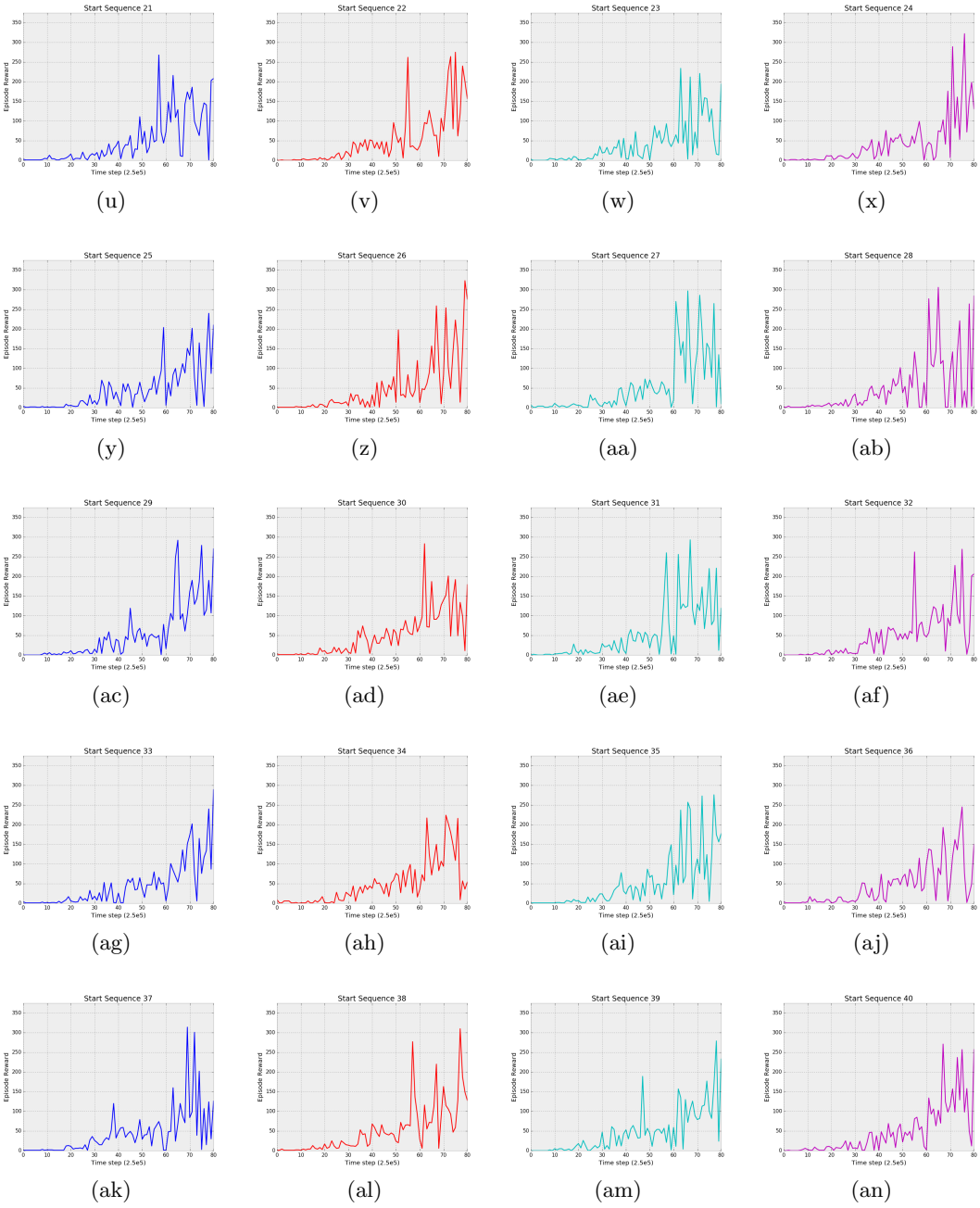
(e) (f) (g) (h)

(i) (j) (k) (l)

(m) (n) (o) (p)

(q) (r) (s) (t)

Figure A.1: Learning curves of the deterministic agent for individual start states/sequences.



(u) (v) (w) (x)

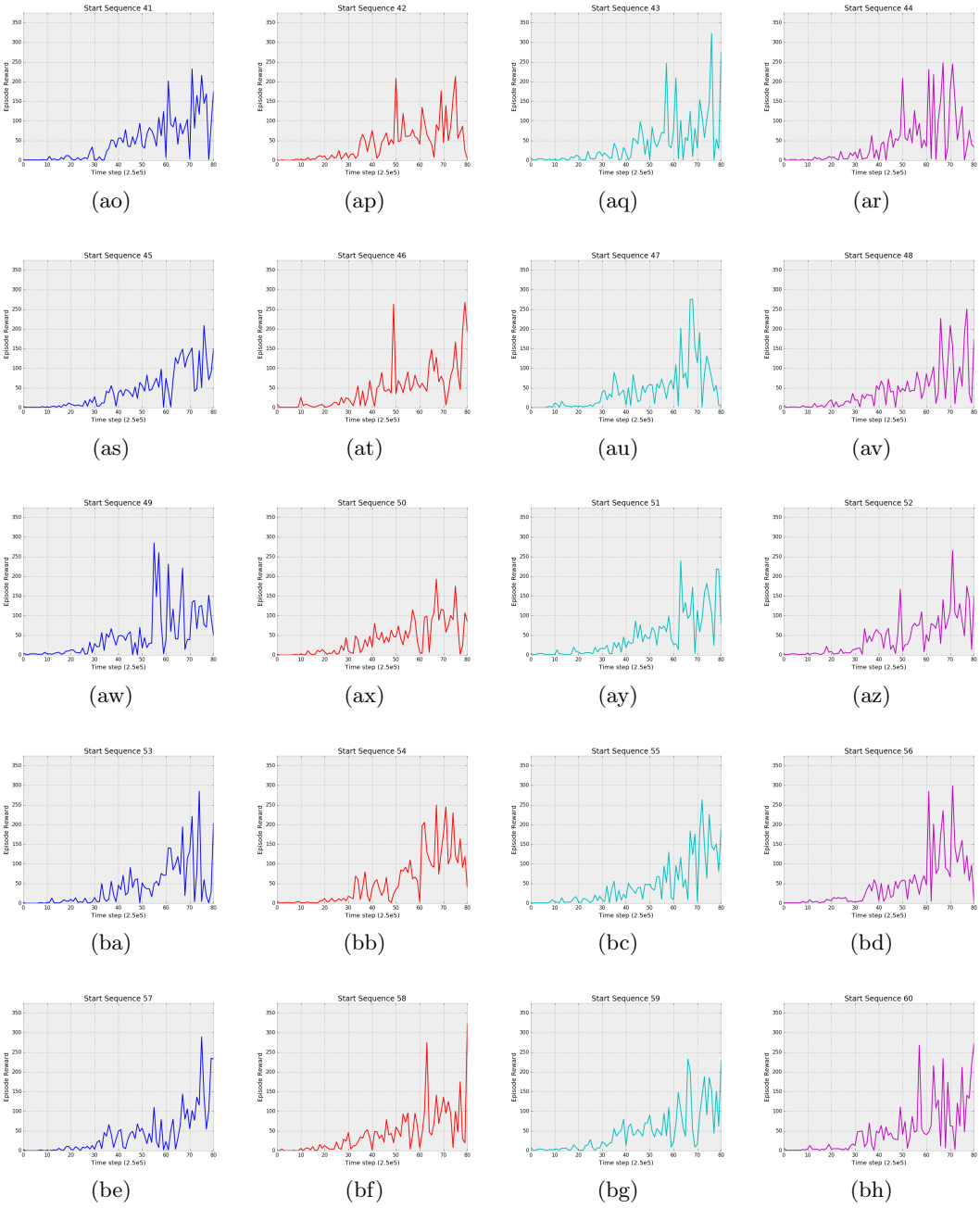
(y) (z) (aa) (ab)

(ac) (ad) (ae) (af)

(ag) (ah) (ai) (aj)

(ak) (al) (am) (an)

Figure A.1: Learning curves of the deterministic agent for individual start states/sequences.



(ao) (ap) (aq) (ar)

(as) (at) (au) (av)

(aw) (ax) (ay) (az)

(ba) (bb) (bc) (bd)

(be) (bf) (bg) (bh)

Figure A.1: Learning curves of the deterministic agent for individual start states/sequences.

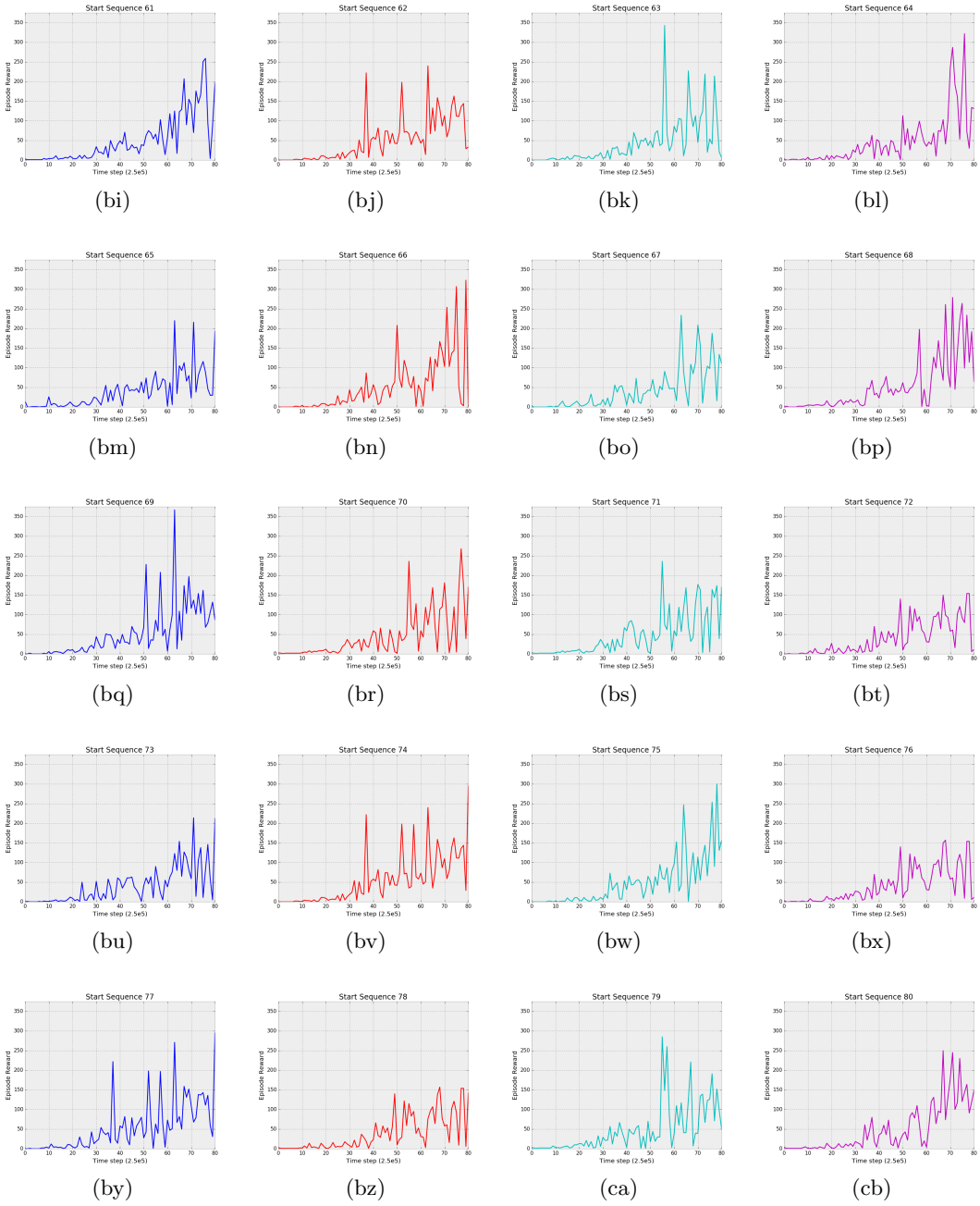
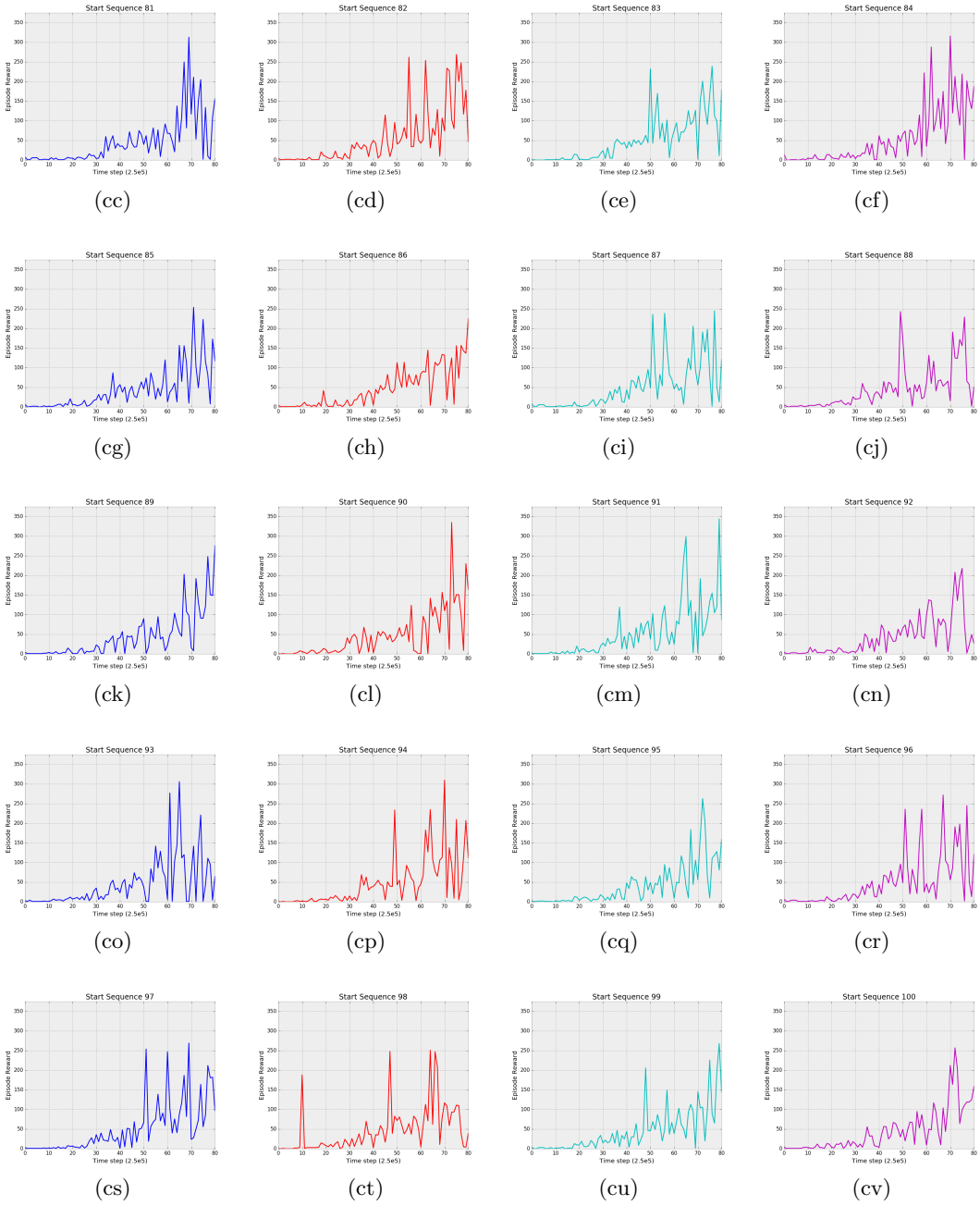


Figure A.1: Learning curves of the deterministic agent for individual start states/sequences.



(cc) (cd) (ce) (cf)

(cg) (ch) (ci) (cj)

(ck) (cl) (cm) (cn)

(co) (cp) (cq) (cr)

(cs) (ct) (cu) (cv)

Figure A.1: Learning curves of the deterministic agent for individual start states/sequences.

Bibliography

[1] Codalab. `http://codalab.org/`. Accessed: 2018-08-07.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] A. A. Ali, M. El-Kalioby, and M. Abouelhoda. The case for docker in multicloud enabled bioinformatics applications. In *Bioinformatics and Biomedical Engineering*, pages 587–601. Springer, 2016.

[4] O. Anschel, N. Baram, and N. Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International Conference on Machine Learning*, pages 176–185, 2017.

[5] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[6] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017.

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

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

[10] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen. Frame skip is a powerful parameter for learning to play atari. *Space*, 1600:1800, 2000.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[12] C. Colas, O. Sigaud, and P.-Y. Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

[13] A. P. Davison, M. Mattioni, D. Samarkanov, and B. Telenczuk. Sumatra: a toolkit for reproducible research. *Implementing reproducible research*, 57, 2014.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[15] B. Djulbegovic and I. Hozo. Effect of initial conditions on reproducibility of scientific research. *Acta Informatica Medica*, 22(3):156, 2014.

[16] C. Drummond. Replicability is not reproducibility: nor is it good science. 2009.

[17] T. Economist. Problems with scientific research: How science goes wrong. *The Economist*, 2013.

[18] B. Fischl. Freesurfer. *Neuroimage*, 62(2):774–781, 2012.

[19] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[20] E. H. Gronenschild, P. Habets, H. I. Jacobs, R. Mengelers, N. Rozendaal, J. Van Os, and M. Marcelis. The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements. *PloS one*, 7(6):e38234, 2012.

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

[22] O. E. Gundersen and S. Kjensmo. State of the art: Reproducibility in artificial intelligence. 2017.

[23] P. J. Guo. CDE: A tool for creating portable experimental software packages. *Computing in Science and Engineering*, 14(4):32–35, 2012.

[24] M. Hausknecht and P. Stone. The impact of determinism on learning atari 2600 games. In *AAAI Workshop on Learning for General Competency in Video Games*, January 2015.

[25] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

[27] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney,

D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[28] J. P. Ioannidis. Why most published research findings are false. *PLoS medicine*, 2(8):e124, 2005.

[29] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

[30] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[31] A. Krizhevsky, V. Nair, and G. Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[33] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[34] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[35] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[36] L. Ma, Z. Lu, and H. Li. Learning to answer questions from image using convolutional neural network. In *AAAI*, page 16, 2016.

[37] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *arXiv preprint arXiv:1709.06009*, 2017.

[38] F. S. Melo, S. P. Meyn, and M. I. Ribeiro. An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th international conference on Machine learning*, pages 664–671. ACM, 2008.

[39] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[40] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[41] P. Nagarajan, G. Warnell, and P. Stone. The impact of nondeterminism on reproducibility in deep reinforcement learning. In *2nd Reproducibility in Machine Learning Workshop at ICML 2018, Stockholm, Sweden*, July 2018.

[42] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[43] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[44] NVIDIA Corporation. Cuda faq. `https://developer.nvidia.com/cuda-faq#Hardware`, 2018. Accessed: 2018-08-12.

[45] NVIDIA Corporation. cudnn developer guide : Deep learning sdk documentation. `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#reproducibility`, 2018. Accessed: 2018-06-19.

[46] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[47] R. D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[48] D. N. Politis and J. P. Romano. Large sample confidence regions based on subsamples under minimal assumptions. *The Annals of Statistics*, pages 2031–2050, 1994.

[49] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[50] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10):e1003285, 2013.

[51] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[52] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[53] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[54] V. Stodden, P. Guo, and Z. Ma. Toward reproducible computational research: an empirical analysis of data and code policy adoption by journals. *PloS one*, 8(6):e67111, 2013.

[55] V. Stodden and S. Miguez. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. 2013.

[56] V. C. Stodden. Trust your science? open your data and code. *Amstat News*, 409:21–22, 2011.

[57] P. Stone, R. Brooks, E. Brynjolfsson, R. Calo, O. Etzioni, G. Hager, J. Hirschberg, S. Kalyanakrishnan, E. Kamar, S. Kraus, K. Leyton-Brown, D. Parkes, W. Press, A. Saxenian, J. Shah, M. Tambe, and A. Teller. Artificial intelligence and life in 2030. Technical report, One Hundred Year Study on Artificial Intelligence: Report of the 2015-2016 Study Panel, Stanford University, Stanford, CA, September 2016.

[58] R. Sutton. Introduction to reinforcement learning with function approximation, 2015.

[59] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[60] E. K. Tang, P. N. Suganthan, and X. Yao. An analysis of diversity measures. *Machine learning*, 65(1):247–271, 2006.

[61] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[62] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[63] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*, 42(5), 1997.

[64] J. J. Van Bavel, P. Mende-Siedlecki, W. J. Brady, and D. A. Reinero. Contextual sensitivity in scientific reproducibility. *Proceedings of the National Academy of Sciences*, 113(23):6454–6459, 2016.

[65] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[66] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[67] B. L. Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[68] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[69] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.