Bachelor Thesis

Designing Experience Replay Algorithms for Off-Policy Reinforcement Learning by Studying their Sampling Distributions

Josep Borrell Tatché

July 2022

Bachelor's Degree in Data Science and Engineering
Bachelor's Degree in Engineering Physics

Advisor	Prof. Pulkit Agrawal
Co-Advisor	Zhang-Wei Hong
Tutora (UPC)	Prof. Carme Torras i Genís

Massachusetts
Institute of
Technology

MIT CSAIL

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos
BCN

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Acknowledgements

I would first like to express my sincere gratitude to my supervisors and advisors at MIT and UPC: Prof. Pulkit Agrawal, Zhang-Wei Hong, and Carme Torras i Genís for their invaluable guidance, expertise and advice. Their insightful conversations have taught me to navigate the daily life of research, pursue new ideas and develop a much deeper knowledge on a lot of different topics.

Additionally, I would like to extend my thanks to all other members of the Improbable AI lab for welcoming me into the lab. Discussions and lab meetings with them were enjoyable and enriching. They helped me learn a lot.

I am extremely grateful to the *Centre de Formació Interdisciplinària Superior* (CFIS) for the opportunity to conduct the research for my thesis at MIT. Especially, I want to thank Toni Pascual for his persistence and continued support in the process of making this exchange possible.

Vull agrair a tota la meva família el suport que m'heu donat sempre, i sobretot durant aquests mesos a Boston. Tinc ganes de veure-us aviat!

Finally, I would like to say thank you to all the amazing people I've met in Cambridge. Especially to Alex, Berta, Cai, Guillem, Héctor, Jakob, Louisa and Marina, thank you for being a great part of this adventure.

Abstract

Off-policy reinforcement learning algorithms make use of experience replay mechanisms to learn from experience gathered by earlier policies. Several algorithms have been proposed to sample transitions from the replay buffer to make training more sample efficient. However, a general framework to compare them and explain their performance differences is missing. In this thesis we propose the transition and state sampling distributions as tools to study these algorithms, allowing to draw comparisons across sampling strategies. An analysis from the distribution point of view reveals that replay buffers are imbalanced, with parts of the state space being underrepresented, while other sections are massively overrepresented. These findings happen across several environments, especially in sparse reward settings. We finish by proposing two algorithms that address the imbalance problem and show that they lead to better performance in sparse reward tasks while matching our baselines in low-imbalance situations.

Keywords

machine learning, reinforcement learning, deep reinforcement learning, off-policy learning, experience replay

Resum

Els algorismes *off-policy* d'aprenentatge per reforç fan ús dels mecanismes de repetició de la memòria per a aprendre de l'experiència viscuda prèviament per altres agents. Existeixen diversos algorismes per a obtenir mostres de la memòria, i tots tenen la intenció de fer el procés d'aprenentatge més ràpid i eficient en el nombre de mostres visualitzades. Tot i això, no existeix cap marc comú que permeti comparar-los i explicar les diferències dels seus rendiments. A aquesta tesi presentarem una eina per a estudiar aquests algorismes: les distribucions de mostreig de transicions i estats. Aquestes eines fan possible establir comparacions entre diferents algorismes de repetició de memòria. Una anàlisi des del punt de vista de les distribucions evidencia que les memòries dels agents no són equilibrades: hi ha parts de l'espai d'estats que estan sobrerepresentades, mentre que d'altres amb prou feines són a la memòria. Aquesta troballa es repeteix en diversos entorns diferents, i de manera marcada a entorns on les recompenses són disperses. Acabarem proposant dos nous algorismes que solucionen el problema del desequilibri i obtenen millor rendiment a tasques amb recompenses disperses, mentre que funcionen igual de bé que els algorismes de referència en situacions més equilibrades.

Paraules clau

aprenentatge automàtic, aprenentatge per reforç, aprenentatge per reforç profund, aprenentatge *off-policy*, repetició de la memòria

Resumen

Los algoritmos *off-policy* de aprendizaje por refuerzo usan los mecanismos de repetición de la memoria para aprender de la experiencia previamente obtenida por otros agentes. Existen varios algoritmos que permiten obtener muestras de la memoria, con la intención de acelerar el proceso de aprendizaje y hacerlo más eficiente en el número de muestras visualizadas. Sin embargo, no existe un marco común que permita compararlos y explicar diferencias en su rendimiento. En esta tesis presentamos una herramienta para estudiar estos algoritmos: las distribuciones de muestreo de transiciones y de estados. Estas herramientas permiten comparar diferentes algoritmos de repetición de la memoria. Un análisis desde el punto de vista de las distribuciones muestra que las memorias de los agentes no están equilibradas: hay partes del espacio de estados que están sobrerrepresentadas, y hay otras que prácticamente no están. Este descubrimiento se repite en distintos entornos, y de manera destacada en entornos con recompensas dispersas. Finalizamos proponiendo dos nuevos algoritmos que solucionan el problema del desequilibrio y consiguen mejor rendimiento en situaciones con recompensas dispersas, mientras que funcionan tan bien como los algoritmos de referencia en situaciones donde el desequilibrio no es un problema.

Palabras clave

aprendizaje automático, aprendizaje por refuerzo, aprendizaje por refuerzo profundo, aprendizaje *off-policy*, repetición de la memoria

Contents

Chapter 1

Introduction

Reinforcement Learning (RL) is an intriguing paradigm. Its main goal is to learn to make decisions by trial and error. This way of learning *by doing* resonates with us as a species, because it is the way in which children and animals learn. We recognize this when we see our younger siblings learn to walk, when we remember how we rode a bicycle for the first time, or when we teach a dog to sit. Therefore, it is natural that human researchers on the mission to create artificial intelligence (AI) thought about teaching their decision-making systems in this way.

A lot has happened in this adventure to artificially create intelligence that is able to learn by interacting with its environment. Impressive results in recent years, ranging from superhuman chess and Go AI players (Silver et al., 2016) to robots that learn to run (Margolis et al., 2022) and grasp (Gu et al., 2017), show that learning by trial and error is more than a shared experience by all humankind. It is a valid framework by which a very different kind of intelligence can evolve.

The evolution of the field and the headline-making results have gone hand in hand with the intensive improvement of the technology used to build these systems. In particular, the introduction of Deep Learning into the toolbox of the 21st century computer scientist has enabled what was before viewed as impossible. The field of deep reinforcement learning has evolved with algorithms that become faster and more efficient every year.

This work will focus on the algorithms for RL. Particularly, we will work with *off-policy RL* algorithms. These training procedures have the goal of training a policy using experience that was gathered in the past and stored in a dataset. They do not care if the data comes from another policy, a human expert, or the same policy. Every transition used for training is treated equally in off-policy RL.

The main feature that enables off-policy learning is *experience replay*. The concept of experience replay is easy to grasp because we all experience it in our daily lives. Humans learn by performing a few trial and error operations, and then reflecting on the outcomes of these trials. In particular, human brains have been shown to replay past experiences during periods of rest to reflect on them, and learn from them. Research on this topic focuses on how to take advantage of memory (also called *replay buffer*) to learn as much and as fast as possible from the experience seen previously.

In this thesis, we aim to study several experience replay algorithms that are ubiquitous in the RL literature. The motivations behind their designs are different, and rarely account for the performance differences found in practice. There is a lack of a common analysis framework to explain when and why a method is more successful than others.

In practice, replay buffers are often found to show an imbalance problem. Because data have been gathered by earlier, suboptimal policies, their biases towards certain regions of the state space and bad exploration properties enter the replay buffer. Usually, methods adapted to work with imbalanced datasets are applied in a situation like this. In RL, instead, algorithms designed for balanced datasets are used.

We propose a set of tools that enable the study of all existing experience replay methods, based on the transition and state sampling distributions. In Chapter 4 we motivate why these tools are useful, based on the replay buffer imbalance problem and the tight relationship of replay algorithms and sampling distributions. By clearly defining the few useful equations that determine the sampling behavior, we can accurately study and diagnose how these methods behave while visualizing the sampling distribution plots.

Once we know more about the existing algorithms for experience replay, and after having established which of these are interesting for us, we would like to design new algorithms that take these insights into account. In Chapter 5 we take the results of the analysis and define two different replay methods that address the imbalance problem and try to mitigate it. We show that they are supported by theoretical findings in the literature. Then, we measure their empirical performance against some benchmarks in several environments.

This thesis is a step towards a better understanding of replay methods and determining what is the main characteristic that yields the largest leaps in performance. I hope it will pave the way for future development of better-understood methods, taking the ideas presented here to more general RL algorithms.

Chapter 2

Problem statement

In this chapter we will define the problem that we want to tackle in this thesis. The goal is to introduce the problem in an intuitive way that motivates the research exposed in the next sections. Chapter 3 will deal with introducing all foundations and existing methods needed to understand the technical concepts that will be investigated. However, before that, the scope of this project will be established so that every chapter takes us closer to better understanding the challenges and answering the right questions.

2.1 Motivation

Reinforcement learning algorithms address the problem of learning by interacting with the real world. There are several strategies for this[1]. We are going to focus on a specific family of these algorithms which takes inspiration from the concept of *experience replay* in neuroscience (Foster, 2017) to design the learning mechanisms.

Neuroscientists have discovered that mammal brains recall episodes of earlier-seen memories in a very rapid way. It has been observed to happen during their rest phase (both in sleep and wakeful resting). This phenomenon has been identified both in rats (Skaggs and McNaughton, 1996) and humans (Kurth-Nelson et al., 2016). The process, usually referred as replay, has been shown to be of utmost importance during learning. For instance, it was discovered that repeated disruption of experience replay in rats resulted in a significant decrease in their learning ability (Ego-Stengel and Wilson, 2010).

Computer scientists trying to embed better learning abilities into their algorithms were inspired by experience replay and tried to implement a similar procedure. Many scientific breakthroughs towards artificial intelligence have happened by trying to imitate the brain, from artificial neural networks to new perception algorithms. This is no exception.

The basic design principles for experience replay are simple: all episodes experienced by the learner are stored in a memory. This memory is usually called *replay buffer*. During learning,

[1] Do not worry, they will be duly described in Chapter 3

the agent is presented with episodic information in the form of states seen and actions taken. The information can be sourced from either interacting with the real world or from the replay buffer. As a result, former experiences are regularly replayed in the agent's "brain", similarly to the discoveries made in rats.

This new tool enables algorithms to be more sample-efficient, since they can get the most of all seen experiences by revisiting them several times. It also helps make RL more applicable in fields where data generation is costly (e.g. robotics or self-driving), because it allows to decrease the number of needed interactions with the real world.

These ideas suggest a possible implementation of experience replay onto reinforcement learning algorithms. However, there are several details that need to be cleared to make them work. They can be divided into two categories:

1. How to take experiences from the replay buffer.
2. How to learn from the information that comes from the memory.

These two concerns raise a number of questions about what can work and which is the best-performing alternative. Existing literature proposes several design choices that result in different implementations of experience replay, with varying levels of performance. All of them work to some extent, but the RL community does not agree on what the best method is. In this thesis we aim to analyze existing approaches to implement replay buffers and experience replay in RL algorithms, draw conclusions from their performance, and propose new alternatives based on the new evidence. We hope that the conclusions we reach may take us some steps closer to a better understanding of experience replay that can make our algorithms more efficient.

2.2 Brief introduction to the mathematical notation

To define the problem, we will greatly profit from defining some notation shortcuts that make the exposition less cumbersome. Let $\mathcal{R}$ be the replay buffer. The information we gather from the environment comes in form of *transitions*.

Every transition t contains a state s, the action taken a, and the next state s'. We represent them using a tuple:

$$t = (s, a, s')$$

The *state* s represents all the information that the learner has on the situation of the world. For instance, for a robot that has a single camera, the image coming from that camera is all the robot knows about the world. That is its current state.

The learner needs to interact with the world. We represent those interactions using *actions* a. Using the robot example, a possible action is picking up a ball from a table.

When an agent executes an action on its surroundings, the state of the world changes. That is what we mean by the *next state* s. In the robot case, the next state would be a new image where the ball is no longer on the table.

The replay buffer $\mathcal{R}$ is the agent's memory. It contains all transitions it has seen during all its lifetime.

$$t_i \in \mathcal{R}, \quad \forall i \in [0, T]$$

T is the total lifetime of the agent. This means that the agent has seen a number of T transitions from its inception up until this very moment.

2.3 Replay buffer operations

Now we have the basic tools we need to talk about how experience replay works. It is interesting to delve deeper into the algorithmic choices that define how agents use replay buffers to learn. In this way we can understand what the different methods focus on.

There are three operations that govern how the interplay between the learning algorithm and the replay buffer works:

1. Storing transitions
2. Sampling transitions
3. Learning from the sampled experience

Let's analyze each of these operations separately.

2.3.1 Storing transitions

Storing transitions in the buffer is pretty straightforward. However, the choice of data structure to represent the memory is important, and it can have a big impact on performance.

Most algorithms choose to have a list-like replay buffer of limited size and append all transitions there. This is the simplest choice, and it has been shown to work well in practice (Mnih et al., 2015; Schaul et al., 2016).

Even though the list option is the most common one, there are other structures to represent the data in the memory. We are particularly interested in graphs as a tool to store the transitions. Graphs provide some interesting properties, such as the ability to more closely represent the structure of episodes and the relationship between them. Imagine two episodes that visit the same state. By using a graph, we can use a single state representation for states visited in both episodes.

Additionally, the implicit information stored in relational structures allow for the application of several path-finding and search algorithms. These algorithms can be used to replay states in novel ways.

In further chapters, research on existing and new methods of leveraging graph-like structures for memory storage and experience replay will be discussed. These can be interesting and will bring additional insights that older list-based methods do not provide.

2.3.2 Sampling transitions

During the training loop, experience replay routines sample from the replay buffer and train the agent as if the transition seen was taken from the real world. Naturally, different sampling strategies can lead to different outcomes in terms of agent performance.

Most experience replay implementations are based on DQN (Mnih et al., 2015), which samples transitions uniformly at random from the buffer. This strategy, while useful and simple, has no theoretical properties that motivate its use. Actually, a line of research on *prioritized experience replay* (Hong et al., 2021; Schaul et al., 2016) deals with alternative sampling techniques that make training more efficient.

We are interested in studying the performance and the properties of alternative replay strategies. Different sampling schemes involve changing the sampling distribution of both the transitions and the states in the memory. This can have a tremendous impact on the learning performance of the agent. The way in which transitions are sampled from the replay buffer and the distributions they induce will be a driving factor of our research.

2.3.3 Learning from the sampled transitions

The way in which the information in the memory is used to make the agent learn is called the *backup process*. It has been given this name because it involves backing up the existing knowledge and the newly found memory information to update the knowledge on what states are better. It is a hot research topic, and one of the oldest in reinforcement learning (H. v. Hasselt et al., 2021; Precup, Richard S. Sutton, and Singh, 2000; Rummery and Niranjan, 1994; Richard S. Sutton, 1988). A nice summary of most existing backup techniques can be found in Richard S. Sutton and Barto, 2018.

The most commonly used backup strategy for value-based methods[2], by far, is 1-step temporal difference (TD) learning. It is inspired by dynamic programming and was already used in Bellman, 1957b. However, by exploiting additional information in the memory, such as the graph structure or episodic rewards, the variance and/or bias of the training signal can be reduced, inducing a more efficient learning.

There are additional factors that may influence learning convergence, which can be partially considered backup strategies. One example is the ordering of memory samples. Another example is the weighting of different samples at training time, when some examples are given more importance than others. It has been shown that weighting strategies are equivalent to changing the sampling distribution as discussed in section 2.3.2 (Fujimoto, Meger, and Precup, 2020).

Novel backup strategies that exploit graphical information have been introduced lately (Jiang

[2] Value-based methods is the family of methods that we are interested in. They will be described in Chapter 3.

et al., 2022). They provide a new point of view on the way in which memory information can be exploited.

2.4 Dataset distribution correction

The process of data curation (Miller, 2014) has become prominent in recent years, with the advent of powerful computing and convenient algorithms that enable data scientists to crunch through large amounts of data. Relevant data repositories have been formed in several scientific domains with the goal of maintaining up-to-date, pristine-clean data sources for the fast advancement of science, with the potential of being reused thousands of times. Some examples of these are the Protein Data Bank ("Crystallography" 1971), containing the largest repository of protein structures worldwide, the Sloan Digital Sky Survey (Blanton et al., 2017), a large scale multi-spectral survey of the universe, and datasets such as ImageNet (Deng et al., 2009), which has contributed to large advances in image recognition technology (Krizhevsky, Sutskever, and Geoffrey E Hinton, 2012).

A relevant task when curating datasets consists on making sure data is correctly labeled and there are no duplicates. When using these datasets for supervised learning, there are other, related issues such as dataset imbalance and class underrepresentation. An imbalanced dataset can result in a drop in performance in models trained using said data. Therefore, machine learning has a history of dealing with issues of data imbalance for supervised training of classifiers (Chawla et al., 2002; Kubat and Matwin, 1997; Pazzani et al., 1994). So, naturally, class rebalancing and example deduplication are important steps in the data science loop to train better systems.

In reinforcement learning, however, not much attention is paid to the replay buffer distribution, which is essentially the training data for the algorithm. Data gathered by previous actors is stored in the replay buffer, so it contains the same transitions multiple times, as well as an imbalanced representation of states in the environment. Transitions are sampled randomly from it, with different methods involving different sampling probabilities. Therefore, a closer look at which samples are used during learning can improve the performance of RL models.

2.5 Research questions

This thesis will focus on the first two replay buffer operations introduced in sections 2.3.1 and 2.3.2. We are interested in knowing how different replay strategies shape the memory sampling distributions. By assessing their performance and relating it to the distribution used, we hope to determine what makes a sampling distribution good for learning. Using this newfound lens, we will also analyze replay buffers based on graphs.

Having said this, it is useful to determine a question that drives the research. In this thesis, the main question is: How do replay buffer sampling distributions affect performance?

Other unanswered queries stem from this one. It would be useful to use the research to reach conclusions on which is the optimal sampling distribution, how can an algorithm be designed so that it achieves it, or how do graphical replay algorithms shape probabilities. The experiments and conclusions found in the next sections will be aimed at reaching explanations to these issues.

Chapter 3

Background and literature review

Reinforcement learning is a rich topic. Its formal framework is based on the fields of probability and algorithms, and it touches upon a diverse set of disciplines: statistical decision-making theory, optimal control, analysis of dynamical systems, etc. To accurately understand the evolution of the field, a broad background is needed on several topics.

In addition, I have invested a large part of this project in getting a deeper knowledge of the many subtopics, advances, and challenges that the RL researchers focus on. Therefore, this section will feature an extensive overview of the information needed to correctly understand and frame the problems that will be tackled in the following chapters. The background section may be redundant to those readers more knowledgeable about the topic, but I hope it helps other readers better understand the problems we are trying to solve and how we address them.

3.1 Machine learning

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data (Mitchell, 1997). It evolved from computer science and some engineering branches. It is considered a subset of artificial intelligence (AI) (Legg and Hutter, 2007), because AI focuses on more objectives other than just learning, such as planning or reasoning.

In general, ML algorithms can be recognized because they perform decisions based on data they have used to *learn*. Some examples of ML algorithms include linear regression and other statistical learning techniques, kernel-based methods, such as support vector machines, and clustering algorithms.

The first recorded evidence of the word machine learning dates back to 1959 (Samuel, 1959). The origins of the field, however, are older. The initial birthplace of machine learning can be set in the 18th century (Fienberg, 2006), with the publication of Reverend Thomas Bayes' now-famous theorem (Bayes and Price, 1763) laying the bedrock for inferential statistics.

Pierre-Louis de Laplace further advanced the knowledge on inference by publishing a paper attempting to estimate the parameter of a binomial distribution (Laplace, 1774). Least-squares regression, arguably one of the most commonplace methods in ML, was already used by Legendre and Gauss in 1805 and 1809 respectively (Stigler, 1986).

3.1.1 Taxonomy of ML algorithms

ML algorithms can be categorized into 3 main groups: supervised learning, unsupervised learning, and reinforcement learning. While some experts claim that there are more subfields inside of machine learning, there is a certain consensus that these are the most relevant ones. We will briefly present supervised and unsupervised learning, while RL will be discussed at large in Section 3.2

Supervised learning

Supervised learning is the subset of machine learning techniques that are trained with labeled data. A typical example of this is an animal classification task: we have a dataset of pictures of cats and dogs, and an algorithm is trained to classify every picture in the dataset to say if there is a cat or a dog. Here, the dataset contains a label for every image that specifies what animal is in the picture.

There are many examples like this, from the classification of plant species from the sizes of their features to the prediction of a student's math marks given the physics and English marks. The common feature in all of them is that a labeled dataset is available or can be obtained to train the algorithms.

Usually, supervised learning tasks are either classification tasks (assign a label to the input data) or regression tasks (predict a numerical value given the input). These are two of the most well-studied settings for machine learning, and algorithms that perform well and achieve incredible results are available off-the-shelf.

Unsupervised learning

Unsupervised learning is the group of machine learning tasks that learn without an explicit supervision signal, hence the name. This umbrella term gathers several concepts.

One of the most typical examples is clustering: a series of data points are embedded in a metric space and the algorithm aims to find groups of similar points. Clustering is a good example of the main goal of unsupervised learning. The idea is that in many real-life situations there are no labels to the data we process, however, we can process it to find structure inside it.

In recent years, a concept known as *self-supervised learning* has shown impressive advances. Although experts do not yet fully agree on a clear definition of what the phrase means (David

Pfau [@pfau], 2022), it is generally used when a task is carefully designed, so a supervised learning algorithm can be used to learn from the structure of the data that was not originally labeled. An example is research on causal (Brown et al., 2020) and masked (Devlin et al., 2019) language models, where algorithms are trained to predict a hidden word given other words in the sentence. Other examples include learning from unlabeled videos or audio signals. The ability to learn from large amounts of unstructured data unlocks the ability to scale the pretraining of machine learning models to previously untapped levels. This is currently the subfield where unsupervised learning is evolving the most.

3.2 Reinforcement learning

Reinforcement learning (RL) is a subset of problems and algorithms in the field of ML. Just like all other branches of ML, the goal is to design adaptive algorithms that *learn* by looking at the data. However, reinforcement learning is slightly different from other ideas in ML: its main purpose is to learn to make decisions and to interact with an environment via trial-and-error learning. Most ML is instead focused on dealing with prediction tasks with independent and identically distributed (iid) data (that is, data that doesn't depend on other data and is drawn from the same probability distribution).

To make the reader aware of how different the problems that RL and other branches of ML try to solve, let us propose a few examples. ML tackles tasks such as predicting how busy the streets will be depending on the weather variables and the time of the day, classifying images of animals saying whether a cat is in the picture, or analyzing if a text reads happy or sad. Instead, reinforcement learning is a good tool for solving problems such as learning to play chess, learning to balance a robot on its feet, or learning to reach a specified point in a maze. As noted in Richard S. Sutton and Barto, 2018: "These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing characteristics of reinforcement learning".

For those readers interested in RL who would like to read a more detailed introduction to it, I refer them to the introduction and first chapters of Richard S. Sutton and Barto, 2018. The overview given here is loosely based on the book.

Rephrasing the above paragraphs, we can arrive at a better definition: the goal of reinforcement learning is to train an algorithm that determines which action to take given some environmental conditions to maximize a numerical reward signal. The word reward is chosen as an analogy for classical conditioning in psychology, where a reward is given to e.g. an animal to reinforce positive behaviors or to punish bad ones.

Another aspect worth mentioning is that the nature of RL makes it a good framework for the embodiment of systems. The problems that RL tackles are problems that usually involve interacting with the outside environment. By setting the environment to be the real world (or a simulation of the world) and giving our algorithm a body such as a humanoid robot or a self-driving car, the framework is trivially extended to train embodied systems in the real world. When a system interacts with the world performing actions and shaping it, we call it

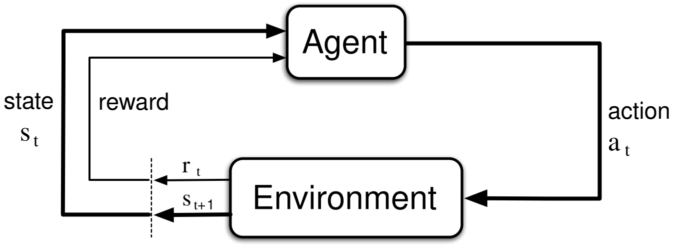


Figure 3.1 / The reinforcement learning / Markov decision process framework.

Adapted from: Richard S. Sutton and Barto, 2018.

an agent or actor.

3.2.1 Markov decision processes (MDP)

To formalize the problems in RL, a framework is established that borrows ideas from the field of dynamical systems. In particular, from the theory of optimization of Markov decision processes (MDP) (Bellman, 1957a). While some simpler tools can be used to work with some instances of problems in RL, such as the multi-armed bandit formalism, MDPs provide a general and powerful framework that is not overly complicated to work with. The dissection of the MDP formalism here is based on the explanations in Richard S. Sutton and Barto, 2018, with a few changes in the notation.

Essentially, MDPs are a formalization of sequential decision-making problems. By taking actions, agents change the environment and receive rewards. And the process can be carried out *ad infinitum*.

More precisely, an MDP consists of two elements: an agent that learns and makes decisions and the environment that contains everything outside the agent and with which the agent interacts. The formal way to specify the framework is as follows: The agent interacts with the environment during a discrete set of sequential time steps $t = 0, 1, 2, \ldots$. The agent sees the *state* of the environment $s_t \in \mathcal{S}$ and takes an *action* $a_t \in \mathcal{A}$. The environment receives the action a_t and returns the *reward* for the action taken $r_t \in \mathcal{R} \subseteq \mathbb{R}$ and the next state $s_{t+1} \in \mathcal{S}$. Each of these steps is called a *transition*. A schematic of this process can be found in figure 3.1.

This sequential process of taking actions, receiving states and rewards, and starting over results in a *trajectory* (also called episode) of the form $\mathsf{e} = (s_0, a_0, r_0, \ldots, s_t, a_t, r_t, \ldots)$. Usually, there are states in the environment called *terminal states* that signal the end of an episode when reached.

It is easier to work in the *finite* MDP setting, where $\mathcal{S}, \mathcal{A}, \mathcal{R}$ are finite sets[1]. In this setup, it is straightforward to define *transition probability*:

[1] The formalism can be extended to work with continuous action and state spaces, but we will not cover it here. H. v. Hasselt, 2012 is a good resource to learn about RL in continuous state and action spaces.

$$\begin{aligned} p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} &\to [0,1] \\ & p(s_{t+1} | s_t, a_t) \end{aligned}$$

Let's also define the *reward probability* p_r and *reward function* r:

$$\begin{aligned} p_r : \mathcal{R} \times \mathcal{S} \times \mathcal{A} &\to [0,1] \\ & p_r(r_t | s_t, a_t) \\ r : \mathcal{S} \times \mathcal{A} &\to \mathcal{R} \subseteq \mathbb{R} \\ r(s,a) = \mathbb{E}\left[r \mid s, a \right] &= \sum_{r \in \mathcal{R}} r p_r(r | s, a) \end{aligned}$$

The function p governs the dynamics of the environment. At first, it is assumed that the transition probabilities are known for all the state-action space ($\mathcal{S} \times \mathcal{A}$) but many interesting problems arise when p is not known or can only be partially approximated.

Reinforcement learning is focused on optimizing the total amount of reward an agent gets during a trajectory, the cumulative reward. To have a clearer metric of the cumulative reward, we use the *return*: the cumulative sum of future rewards:

$$g_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_T$$

where T is the end of the episode. This is what the agent aims to maximize at every time step. Once we have defined the return, we need to be able to tweak the return to make the agent more or less myopic: make it look for short or long-term rewards. There is also the possibility that $T = \infty$, then $g_t = \infty$, resulting in an intractable problem. To enable these cases we use the *discounted return*:

$$g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{\tau=0}^{\infty} \gamma^{\tau} r_{t+\tau}$$

Where $\gamma \in [0,1]$ is the *discount rate*. When $\gamma = 1$ we recover the original case of return and when $\gamma = 0$ we are in the "myopic" case where the agent only cares about the immediate next reward.

Policies

We have now defined most functions and variables we need to completely define the environment. However, the environment represents only half of the MDP. In most cases, the environment is something given and passive, and we particularly care about the agent part of the MDP. Let's define some functions that characterize the agent.

The function that maps states to actions is called the *policy* π:

$$\pi : \mathcal{S} \to \mathcal{A}$$
$$\pi(a|s)$$

The goal in RL is to find an optimal policy π_* that maximizes the discounted returns:

$$\pi_* = \arg\max_{\pi} \mathbb{E}_{\pi}[g_t \mid s_t] \;\; \forall s_t \in \mathcal{S}$$

Note that the policy used influences the returns obtained by the agent. To make this point clear we explicitly write that we take the expectation using policy π of the return. This *search in the policy space* for the optimal policy is the main challenge in RL and the one for which all algorithms are designed.

Value functions

Once we have a model for the environment $p(s_{t+1}|s_t, a_t), p_r(r_t|s_t, a_t)$ and a policy π, we can compute a family of interesting metrics called *value functions*. These represent the value that each state has when acting under a particular policy.

The *(state-)value function* of a state s under a policy π is equal to the expected return when starting from s and acting with π. It is denoted by $V^\pi(s)$ and defined by:

$$V^\pi(s) = \mathbb{E}_\pi[g_t \mid s_t], \quad \forall s \in \mathcal{S}$$

The *action-value function* $q^\pi(s, a)$ is the expected return of taking the action a in state s and then following the policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[g_t \mid s_t, a_t], \quad \forall (s, a) \in \mathcal{S} \times \mathcal{A}$$

We will now introduce a particular property of value functions that will be useful in the future. It is called the *self-consistency* property, more popularly known as the *Bellman equation* for the state-value function:

$$\begin{aligned} V^\pi(s) =& \mathbb{E}_\pi[g_t \mid s_t] = \cdots \\ =& \sum_a \pi(a|s) \sum_{s',r} p(s'|s, a) p_r(r|s, a) \left[r + \gamma V^\pi(s')\right] \\ & \forall s \in \mathcal{S} \end{aligned} \tag{3.1}$$

Optimal policies and optimal value functions

As mentioned earlier, the goal of RL is to search for the optimal policy π_*. There is at least one optimal policy, and there can be more than one, but all share the same *optimal state-value* *function* V_*. In particular, we define the optimal value function V_* as:

$$V_* = \max_{\pi} V^{\pi}(s), \quad \forall s \in \mathcal{S}$$

The optimal policies also have the same *optimal action-value function* q_*:

$$Q_* = \max_{\pi} Q^{\pi}(s,a), \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A}$$

Using the concept of the value functions it is straightforward to deal with the fact that there are several optimal policies. The important idea to take into account is that, even though they may be different, they are equally optimal because they provide the same expected value for every state and action. Therefore, none of them is better or worse than the other on average.

Similarly to the Bellman equation for the value function [3.1], we can define the *Bellman optimality equation* for the optimal state-value function. Essentially, it says that the value of a state for the optimal policy is equal to the expected return of the best action:

$$\begin{aligned} V_*(s) &= \max_a Q_*(s,a) = \max_{a_t} \mathbb{E}\left[r_t + \gamma V_*(s_{t+1}) \mid s_t, a_t\right] \\ &= \max_a \sum_{s',r} p(s'|s,a) p_r(r|s,a)[r + \gamma V_*(s')] \end{aligned} \tag{3.2}$$

In the same spirit, we can define a similar Bellman optimality equation for the action-value function:

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}\left[r_t + \gamma \max_{a'} Q_*(s_{t+1}, a') \,\Big|\, s_t, a_t\right] \\ &= \sum_{s',r} p(s'|s,a) p_r(r|s,a) \left[r + \gamma \max_{a'} Q_*(s', a')\right] \end{aligned} \tag{3.3}$$

Note that the optimality equations [3.2], [3.3] do not depend on the policy. Since we are performing max operations on the actions, we do not need the policy to tell the agent how to act. This property will be useful in the design of off-policy reinforcement learning algorithms.

An important remark is that, given an optimal value function, we can obtain an optimal policy $\pi_* : \mathcal{S} \to \mathcal{A}$. Using the action-value function as an example:

$$\pi_*(s) = \arg\max_a Q_*(s,a) \tag{3.4}$$

The same can be done for the state-value function by taking the expectation over the environment transitions:

$$\begin{aligned} \pi_*(a|s) &= \arg\max_a \mathbb{E}_{p,p_r}[r + \gamma V_*(s_{t+1}) \mid s_t, a] \\ &= \arg\max_a \sum_{s',r} p(s'|s,a) p_r(r|s,a) \left[r + \gamma V_*(s')\right] \end{aligned} \tag{3.5}$$

3.2.2 Dynamic programming to solve MDPs

There are several approaches based on dynamic programming (DP) (Bellman, 1957b) that can be used to compute the optimal value functions (and thus policies) given an MDP. These form the basis of many RL algorithms that are used nowadays. We will give a brief overview of them because they are central to understanding the methods explained later. These methods are based on the iterative approximation of value functions.

Policy evaluation

To compute the state-value function of the policy $v^{\pi}(s)$, one option is to iteratively apply the Bellman equation [3.1] as an update rule. This is the algorithm known as *iterative policy evaluation*. The update rule is as follows.

$$V^{\pi}_{k+1}(s) \leftarrow \sum_{a} \pi(a|s) \sum_{s',r} p(s'|s,a)p_r(r|s,a)\left[r + \gamma V^{\pi}_{k}(s')\right] \quad \forall s \in \mathcal{S} \qquad [3.6]$$

This algorithm is guaranteed to converge to v^{π} (Bertsekas, 2015), where the convergence is defined as $\left\{\sum_s \|v_k(s) - v^{\pi}(s)\|\right\}_k \rightarrow 0$.

Policy improvement

This algorithm is concerned with improving the policy given a value function. The key result that enables this is *policy improvement theorem*. This theorem states that, given two deterministic policies π, π', such that

$$\forall s \in \mathcal{S} : Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \qquad [3.7]$$

Then the policy π' is at least as good as π. In other words:

$$V^{\pi'}(s) \geq V^{\pi}(s),\ \forall s \in \mathcal{S} \qquad [3.8]$$

The proof of this result can be found in Richard S. Sutton and Barto, 2018. What the theorem suggests is that when we find a pair of policies π, π', we can update $\pi \leftarrow \pi'$ that verify property 3.7, we can use them to improve the values of all states.

In practice, this can be used as an opportunity to improve current policy by taking max over all possible actions. This can be written as follows.

$$\begin{aligned} \pi'(s) &= \arg\max_{a} Q^{\pi}(s,a) \\ &= \arg\max_{a} \sum_{s',r} p(s'|s,a)p_r(r|s,a)\left[r + \gamma V^{\pi}_{k}(s')\right] \end{aligned} \qquad [3.9]$$

Policy iteration

Now we know how to compute the value function of a policy (policy evaluation) and how to improve a policy given a value function (policy improvement). By combining these previous two algorithms we can obtain an optimal policy. This is called *policy iteration* and consists of iteratively performing policy evaluation and policy improvement steps. Every update is guaranteed to be a strict improvement on the previous policy or value function. Because the MDPs we are working with are finite, the succession of strict improvements is guaranteed to find the optimal policy (Bertsekas, 2015). The process schematic is as follows.

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \cdots \rightarrow \pi_*$$

Policy iteration can be slow, because it involves policy evaluation until convergence at each step. The algorithm guarantees convergence, but does not provide any measure of speed to the solution. Value iteration attempts to solve this issue.

Value iteration

To avoid waiting for the full convergence of the policy iteration, we can obtain the *approximate* value functions by stopping before the policy evaluation algorithm finishes. This does not cost us the loss of the convergence guarantees (Richard S. Sutton and Barto, 2018). In the particular case of *value iteration*, we only do one iteration of the policy evaluation loop, then update the policy and start again.

$$V^{\pi}_{k+1}(s) = \max_a \sum_{s',r} p(s'|s,a)p_r(r|s,a)\left[r + \gamma V^{\pi}_k(s')\right] \qquad [3.10]$$

This can be viewed as using the Bellman optimality equation [3.2] as an update rule. Other variations of value iteration are often used in practice, the only difference being that the policy improvement steps are done after a different number of policy evaluation steps. These methods are called *generalized policy iteration*, and they are also guaranteed to converge to the optimal policy π_*.

3.2.3 Not all RL is the same

Reinforcement learning algorithms and applications have evolved in different directions and *flavors*. To better understand the field, a taxonomy that allows for the categorization of problems and methods is useful. There are several axes by which we can divide the field: model-free vs. model-based, online vs. offline, or on-policy vs. off-policy.

Model-free and model-based RL

Up until now, we have been assuming access to a perfect model of the environment. For every pair of state-actions (s, a), we have a model $s \sim p(s_{t+1}|s_t, a_t)$ that perfectly describes how the environment dynamics are going to work. This is true for some cases, e.g. board games: in chess, the exact outcome that a move will have is known. More complicated games can use dice to introduce randomness into the game, but we can still build a simple model that takes the probabilities of dice into account.

In cases where there is no finite set of well-defined rules, we can build an approximate model of the environment that is good enough to run an RL algorithm and succeed. This is a common approach in the literature (Hafner et al., 2019; Kaiser et al., 2020; Nagabandi et al., 2020).

In practice, however, approximate models suffer from a common drawback: *compounding error*. As RL involves sequential decision-making, model predictions involve applying a model on top of a long chain of earlier predictions. If the model has a small error every time a prediction is made, all errors will add up during a trajectory, thus resulting in a large error after a few time steps and rendering the approximation useless. To bypass the fact that there are no available models and approximate models are not useful, a family of RL algorithms called *model-free* drops the model assumption and acts by discovering everything about the environment from the transitions they experience.

The model-free approach has been successful in recent years, with the advent of neural networks as a powerful learning algorithm and the ubiquity of intensive computation resources in research labs. Model-free deep RL algorithms have been applied to Atari games (Mnih et al., 2015), harder video games such as StarCraft (Vinyals et al., 2019), as well as several real-world robotics applications like manipulation (Gu et al., 2017) and locomotion (Hwangbo et al., 2019; Margolis et al., 2022).

On-policy and off-policy RL

This distinction is relevant to the main problem and the research intentions of this thesis, as specified in Chapter 2. It is a difference that affects the algorithms used for learning while keeping the other aspects of the framework (the RL problem, interactions with the environment) pretty much the same. It emerges more clearly with deep RL model-free algorithms, where the dynamic programming through the environment dynamics approach is less effective or computationally tractable.

The most naive way to think about an RL algorithm is to imagine an agent taking action in the world, getting some reward signal back, and using the signal to train the policy. This approach is called *on-policy learning*, and it is fine in cases where very little is known about the environment[2]. However, as explained in Chapter 2, the idea of replay in neuroscience has been shown to be relevant in the development and learning of the mammal brain. As

[2] In particular, they are applied in settings where the Markov property does not hold.

a result, some algorithms implement a replay buffer where they store all experiences and recall them regularly in a process similar to memory replay.

But experience replay poses an additional constraint on the algorithms used. They cannot assume that the action decisions were made by the current policy, because this is not true. The experience the agent sees during training comes from the memory gathered using an array of policies at different learning stages. As a result, algorithms used to learn by experience replay have to be *off-policy*: they have to be able to learn from experience collected using other policies.

The Bellman optimality equation is useful in this case because it does not depend on the policy, as stated in section 3.2.1. As a result, most of the off-policy RL algorithms are based on the Q-learning algorithm. This allows them to take all policy dependence out of the update rules thanks to the Markov assumption and max operations. Therefore, all transitions stored in the replay buffer help to take learning a step further towards the optimal action-value function.

Online and offline RL

There is yet another way in which RL algorithms can be divided. The difference between online and offline learning relates to how an agent interacts with the environment during learning.

An agent learns *online* if it interacts with the environment during learning. This means that no matter how it is trained (directly from experience, using a replay buffer, etc.), the agent is able to test its guesses in the real domain. This enables properties that make it easier to solve several problems, such as online exploration or counterfactual action testing.

In contrast to this, *offline learning* means that the agent cannot interact with the environment at any step of the training process. Usually, learners are trained online to get a fixed dataset from which they have to learn. This poses many difficulties in the learning process, mostly related to exploration. Levine et al., 2020 is an excellent and complete survey on the main characteristics and challenges found in offline RL.

The main challenge in offline learning is the *overestimation* of some actions. If a given state-action pair is either not found or underrepresented in the dataset, there is a risk that the agent assigns it a good value and has no way of testing whether this is true. If this situation arises in the online setting, the agent would quickly try to execute the action and realize that it was actually not good. This stems from the two most common issues that a fixed dataset brings: the inability to explore transitions that are not in the dataset and the distribution shift between the training data and the setting in which the agent will be tested.

To solve this issue, several regularization methods have appeared in recent years. The most relevant among the proposals are CQL (Kumar, Zhou, et al., 2020) and IQL (Kostrikov, Nair, and Levine, 2021). The main idea behind them is that agents should be skeptical about transitions that are not in the dataset, and derive regularization measures to avoid thinking that unseen transitions are good. The term *pessimism in the face of uncertainty* has been coined for these algorithms, in opposition to the concept of optimism under uncertainty found in traditional online RL literature.

3.3 Deep learning

Deep learning (DL) is a subfield of machine learning that studies *artificial neural networks*. Arguably, this field started with the invention of the perceptron (Rosenblatt, 1958), an adaptive algorithm that is loosely based on a mathematical model of how the brain works.

From the point of view of the perceptron and seriously simplifying the topic, brains are a network of neurons that are highly connected to each other by axons. Neurons can fire as a response to a sensory impulse. When a neuron fires, the axons transmit the signal information to all other neurons to which the original is connected. In this way, a neuron receives input information from many other neurons connected to it. When this input information exceeds a certain threshold, it causes the neuron to fire. So, in the end, the brain is a network of traveling spiking signals that can process information when it is sufficiently large in scale.

Because of their biological-like foundation, neural networks have been studied as a path to achieving artificial intelligence. In some controversial claims that were highly debated among the ML community, some of the most prominent researchers in the field have dared to suggest that they may even be "slightly conscious" (Ilya Sutskever [@ilyasut], 2022). Even though this is very interesting, we are going to constrain ourselves to merely analyze how they work in their most basic form from a machine learning or statistical inference perspective.

Artificial neural networks are made up of artificial *neurons*. A single neuron can be viewed as a function f that takes a vector input $\mathbf{x}$ and returns a scalar output $f(\mathbf{x})$. The neuron also contains an adaptive weight vector and bias ($\mathbf{w}, b$ respectively) that can be trained. f is composed of two simple operations: (1) multiply the inputs by the weight vector and add the bias: $\mathbf{w}^T\mathbf{x} + b$, and (2) apply a nonlinear function g to the result of this linear operation. In conclusion, the operations on a neuron can be written as:

$$\begin{gathered} \text{neuron} = (f, \mathbf{w}, b, g) \\ f : \mathbb{R}^n \to \mathbb{R} \\ \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R} \\ f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b) \end{gathered} \tag{3.11}$$

These neurons are in turn combined in *layers*. We can treat them as a function $\ell : \mathbb{R}^n \to \mathbb{R}^m$. These are essentially a concatenation of artificial neurons: $\ell(\mathbf{x}) = \{f_i(\mathbf{x})\}_m$. We can represent the effect of a layer on an input vector $\mathbf{x}$ using linear algebra:

$$\begin{aligned} \mathbf{W} \in \mathbb{R}^{m\times n},\ \mathbf{b} \in \mathbb{R}^m \\ \ell(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \end{aligned} \tag{3.12}$$

Where g is applied element-wise on the vector.

A concatenation of these layers is called multi-layer perceptron (MLP), which is the most basic instance of a neural network. By changing the way the inner operations of neurons take place a bit, we can build new types of neural network, such as convolutional neural networks.

Details about neural networks are not very relevant to this work. While they are a very useful tool that will be used extensively onward, we will treat them as black-box function approximators. Therefore, it is not worth it to go into much more detail about how they are trained (Rumelhart, Geoffrey E. Hinton, and Williams, 1986) and the different architectures used (Hochreiter and Schmidhuber, 1996; Lecun et al., 1998; Vaswani et al., 2017).

The concept of *function approximator* is a useful one. Suppose that we would like to have an oracle mathematical function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ to which we do not have access. However, some samples of inputs $\mathbf{x}$ and outputs $f(\mathbf{x})$ of the function f are available. Therefore, we can fit a function approximator $\hat{f}$ so that it produces the expected behavior in the samples: $\hat{f}(\mathbf{x}) \simeq f(\mathbf{x})$ and hope that it will behave like f when given inputs that are not available in the training samples.

In reinforcement learning the goal is often to approximate value functions or value functions to which access is not available. Function approximators are useful for training a function that gives the expected values for every state and action. This idea is the main concept around neural networks that will be used going forward.

3.4 Deep RL algorithms

Reinforcement learning has used *function approximation* as a tool for many years. The objective of the function approximation paradigm is to train statistical models to simulate the expected input-output behavior of the functions in the RL framework (environment dynamics, value functions, policies, etc.). A RL system requires these approximation components when the problem becomes too complex, either because the state-action combinatorial space becomes too large [3] to account for, or because a continuous state space calls for a different type of solution. In both of these cases, the application of state-action tables in the MDP framework as seen in Section 3.2.1 is unfeasible.

The first instances of function approximation in RL are in linear value function approximation (Richard S. Sutton, 1988). Later methods use decision trees (Adele and Howe, 1998), kernel methods (Ormoneit and Sen, 2002), or other more complex regression methods, such as the linear quadratic regulator (Kwakernaak and Sivan, 1972). Although these methods of function approximation achieved impressive results (Abbeel et al., 2006), the most influential results in RL have historically used neural networks as the main function approximator.

The reinforcement learning subfield that studies neural network-based RL algorithms is

[3] Using board games as an example, Backgammon has 10^{20} possible states, and Go has about 10^{171} distinct board configurations.

called *Deep RL*. Some early successes of neural networks as function approximators are TD-Gammon (Tesauro, 1995), or robot soccer (Riedmiller et al., 2009). Although artificial neural networks have been used in RL since the early 2000s (Riedmiller, 2005), deep reinforcement learning has evolved at a fast pace alongside the field of deep learning since 2012.

Deep model-free RL algorithms are divided into two main classes: Deep Q-Learning and Policy Gradient methods. Both their origins date back to the pre-deep RL literature, but, as they both represent good ways of getting gradients for learning, they have been applied to deep learning with great success. Although sometimes their differences are blurred, they are based on two different principles, which we will elaborate briefly.

3.4.1 Q-Learning

Q-Learning (Watkins and Dayan, 1992) is an algorithm based on a dynamic programming approach similar to value iteration. It works by iteratively applying the Bellman optimality equation as an update rule for the Q-function (the state-value function). Recall that the Bellman optimality equation claims that a state-action pair's value is equal to the expectation of the reward obtained plus the best q-value at the next state. As we already saw in section 3.2.1, the Bellman optimality equation [3.3] is off-policy, it does not depend on the policy. This property is good for the algorithm, as we can use samples that were obtained using past policies to train our present value function.

To mathematically formulate these thoughts, we will use the Bellman optimality operator $\mathcal{T}$, which serves as a shorthand for the Bellman optimality equation.

$$(\mathcal{T}Q)(s,a) = \mathbb{E}\left[r + \max_{a'} Q(s',a')\right] \qquad [3.13]$$

As a result, we can write the Bellman optimality equation [3.3] in an easier form:

$$Q_* = \mathcal{T}Q_* \qquad [3.14]$$

In the stochastic setting, we cannot take averages if we cannot perfectly model the environment. However, we can perform progressive updates by dynamic programming that asymptotically tend to the average. Therefore, the Bellman operator in the stochastic, more practical case is defined as

$$(\mathcal{T}Q_*)(s,a) = r_t + \max_{a'} Q_*(s_{t+1},a') \qquad [3.15]$$

If we use the operator as an update rule instead of a simple tool to shorten the equations, we arrive at the Q-Learning algorithm:

$$Q^{t+1}(s,a) \leftarrow (1-\alpha)Q^t(s,a) + \alpha(\mathcal{T}Q^t)(s,a) \qquad [3.16]$$

It may seem that we are talking about policy-independent q functions, while we have been writing all value functions as dependent on a policy (V^π, Q^π). There is a slight detail: Q-

Learning implicitly assumes the agent is using the greedy policy that selects actions by taking the maximum action value at every state. This choice is done when *boostrapping* the values of the next state, where the algorithm assumes $V^{\pi}(s) = \max_a Q^{\pi}(s, a)$. It is the iterative application of this update rule that allows the algorithm to be off-policy and therefore get the concept of a policy-independent state-action value function.

The tabular version of this algorithm, which involves the iterative application of the Bellman update rule, can be applied in a straightforward way. However, if we want to use some function approximation method, which usually requires gradient-based optimization, the updates need to be written in terms of loss functions. It is pretty straightforward:

$$\hat{Q} = \arg\min_{Q} \mathbb{E}\left[(Q(s,a) - (\mathcal{T}Q)(s,a))^2 \right] \qquad [3.17]$$

The term within the loss function is called the temporal difference (TD) error, and it is a common measure of error for many value-based RL algorithms.

Formally, it can be shown that the Bellman backup is a γ-contraction in the L-∞ norm and q_* is its fixed point. Using these assumptions, we can show that the tabular Q iteration algorithm converges to q_*. Although this is interesting, we will simply use the result and refer the reader to Richard S. Sutton and Barto, 2018 for more formal arguments.

However, when using function approximation, these convergence guarantees are lost. Even in the case of fitted Q-iteration, which uses a linear Q-function, there are no guarantees for convergence of this algorithm. But, as sometimes happens in computer science, it has been shown to work in practice in many cases.

The idea of Q-Learning is a standard foundation in deep RL algorithms. We will see some examples in the following sections.

3.4.2 Policy gradients

Policy gradients propose a simple idea. Instead of measuring value functions and improving them to get a measure of how good every state is, they tackle a problem that is arguably simpler. These algorithms *optimize the policy directly*. The cumulative reward J perceived by the agent is a function of the policy π_θ, therefore gradients can be obtained and the policy can be trained to minimize them.

The policy gradient approach is especially useful in continuous or high-dimensional action settings, where value-based methods have drawbacks. In continuous action spaces, an algorithm learning the q-value of every state-action pair needs to record an infinite number of values for every action. With high-dimensional actions, the number of actions grows exponentially, making the search over the action space too costly. Policies trained using policy gradients bypass all this by training only a policy function $\pi_\theta(a|s) : \mathcal{S} \rightarrow \mathcal{A}$.

The reward function is defined as: [4]

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s,a) \qquad [3.18]$$

We want to find the policy gradient $\nabla_{\theta} J(\theta)$. It is not easy to do so, as the state distribution $d^{\pi}(s)$ depends on the policy π, which makes it harder to take derivatives. The *policy gradient theorem* (Richard S Sutton et al., 1999) solves this obstacle by rewriting the derivative of the return without requiring to know how the state distribution works.

The proof of the theorem is not the main focus of this thesis, so it will be omitted. It can be found in Richard S. Sutton and Barto, 2018, along with clear explanations of what the result implies. The result of the policy gradient theorem is:

$$\nabla_{\theta} J(\theta) \propto \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \nabla_{\theta} \pi_{\theta}(a|s) \qquad [3.19]$$

Using the well-known property of the derivative of the logarithm that $(\ln x)' = 1/x$, we can rewrite the expression as:

$$\begin{aligned} \nabla_{\theta} J(\theta) &\propto \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \nabla_{\theta} \pi_{\theta}(a|s) \\ &= \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s,a) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} \\ &= \mathbb{E}_{\pi}[Q^{\pi}(s,a) \nabla_{\theta} \ln \pi_{\theta}(a|s)] \end{aligned} \qquad [3.20]$$

Here $\mathbb{E}_{\pi}$ means that the state and the action distributions depend on the current policy (so the gradient is computed using on-policy data). The expression with the expectation [3.20] is the most commonly used in practice, because the gradient can be obtained using some recent samples and taking their average.

This gradient presents an unbiased update rule to optimize the policy, but has a high variance. Several techniques have been proposed to make learning more stable while maintaining a low bias. The most common involve subtracting a baseline b from the Q-value to reduce variance, because $|Q(s,a) - b| \leq Q(s,a)$ if $b < Q(s,a) \forall (s,a)$. The most typical example is called *generalized advantage estimation* (GAE) and consists on using the advantage function $A^{\pi}(s,a) := Q^{\pi}(s,a) - V^{\pi}(s)$. The main benefit relies on A being smaller in magnitude and centered around 0, given that $\max_a Q^{\pi}(s,a) \geq V^{\pi}(s) \geq \min_a Q^{\pi}(s,a)$. This causes the metric to have less variance.

3.4.3 Relevant algorithms

There is a vast literature in RL algorithms which use a large array of techniques to make learning faster and more sample efficient. For the development of this thesis, however, we

[4] The exposition in this section is based on an insightful blog post by Weng, 2018

will not study them too deeply because we do not need most of them. Instead, we will provide a brief introduction to some of the most representative methods in deep RL and then study the only one we need.

VPG, TRPO and PPO

Some modern algorithms consist on applications of the *policy gradient* theorem. The most basic implementation of the idea is called *Vanilla Policy Gradient*, which consists in estimating the advantage function using collected trajectories and their rewards to go, and then performing a gradient ascent on the policy gradient.

When optimizing large, non-convex function approximators such as neural networks, a common issue is that small changes in parameter space can introduce huge variations in the behavior of the policy. To solve that, *Trust Region Policy Optimization* (TRPO) (Schulman, Levine, et al., 2015) establishes an additional condition on the optimization routine. It limits the policy updates by imposing a bound on the Kullback-Leibler divergence[5] between the old and new policies. To implement the bounds on the KL divergence, TRPO uses line search and second-order methods, which are computationally demanding. In an attempt to solve this problem with simpler, faster methods, *Proximal Policy Optimization* (PPO) (Schulman, Wolski, et al., 2017) poses a bound on the quotient of the new and old policies. In the setting where the advantage $A^{\pi}(s,a)$ is positive, the loss function that PPO optimizes is:

$$\mathcal{L}(s,a,\theta_k;\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a) \qquad [3.21]$$

And if $A^{\pi}(s,a) < 0$, the second term changes by $1-\epsilon$. By imposing these bounds, the optimization process ensures that policies change in small steps without using higher-order methods that make neural network optimization difficult.

DQN

The algorithm known as *Deep Q-Network* (DQN) (Mnih et al., 2015) is based on Q-learning instead of the application of the policy gradient theorem. It performs the Q-Learning operations explained above, but using a deep learning model as a Q-function approximator Q_θ. The reason behind this choice is that the state and action space soon become too large to record every possible input in a table. Instead, a neural network is trained in the hope that it summarizes all the knowledge in its weights and that it is able to generalize to new, unseen observations given the data it has seen in the past.

This combination of the goals of machine learning—striving for generalization—and of reinforcement learning—learning the right policy—has proven to be effective on a large number of occasions. DQN was arguably the first algorithm to show that these ideas can

[5] A divergence is a distance-like measure between two probability distributions, albeit with weaker properties than a distance

work on a large scale.

From a more algorithmic standpoint, DQN proposes a few tricks to enable the algorithm to learn. Since neural networks can be slightly brittle during training, some modifications have to be made to Q-Learning to make it work. The main trick is the addition of a target network: Because we optimize Q_θ to minimize $\mathcal{L} = (Q_\theta(s_t, a_t) - (r_t + \max_{a'} Q_\theta(s_{t+1}, a')))^2$, the two terms inside the loss function will change after every gradient update. It essentially means that the function is chasing a moving target. This goes against the assumptions of most machine learning algorithms, where a stationary target is assumed. To avoid this, the authors propose using a slowly updating target network Q_{θ^-}. It is updated far less frequently than the model trained, using polyak averaging of the weights with the trained models' weights:

$$\theta^- \leftarrow (1-\rho)\theta^- + \rho\theta \qquad [3.22]$$

Another trick is the use of a replay buffer, where old experience gets stored and is sampled later to make the network learn. This enables the algorithm to be off-policy and learn from a large amount of data. Online data collection to learn is no longer necessary.

The combination of a convolutional neural network, the Q-Learning algorithm, a replay buffer, and the target network trick is basically what binds the DQN algorithm. We will be using it for all our experiments here, due to its simplicity compared to other RL algorithms and the ubiquity of good implementations.

DDPG and TD3

Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2021), learns both a Q-function and a policy. The Q function is learned using data in a replay buffer, and the Q function is then used to learn the policy. This architecture is called Actor-Critic, and it is a common one in RL algorithms. One of the benefits of DDPG is that it can be used to generalize the Q-Learning idea to continuous action spaces, by learning a policy.

The Q function is learned as with DQN, using the replay buffer and target network trick. Novelty here comes in the policy training. Since in a continuous action space it is unfeasible to search to optimize $\max_a Q_\theta(s, a)$, a policy π_ϕ is trained such that:

$$\phi = \arg\max_{\phi} \mathbb{E}_{s\sim\mathcal{R}}\big[Q_\theta(s, \pi_\phi(s))\big] \qquad [3.23]$$

But DDPG is frequently brittle at tuning time. To overcome these issues, *Twin Delayed DDPG* (TD3) (Fujimoto, Hoof, and Meger, 2019) proposes some improvements. The most common problem in DDPG is that the Q-function can erroneously overestimate values, and the policy optimizes those errors, leading to performance loss.

The main tricks proposed by TD3 are:

1. Clipped double Q-Learning: two Q-functions are learned, and the smallest of the

values obtained is used to compute TD errors.

2. Delayed updates: the policy and target networks are updates less often than the Q-function.
3. Target policy smoothing: the Q-function is smoothed along changes in action by adding noise to target actions.

These changes solve some of the issues in DDPG.

SAC

Soft Actor-Critic (SAC) (Haarnoja et al., 2018) also proposes an actor-critic architecture, with the difference that the policy is stochastic. In this way, stochastic policy optimization theory and actor-critic algorithms are unified in the same method. Additionally, unlike DDPG and TD3, SAC can work with continuous and discrete action spaces.

The largest difference is that SAC has an entropy regularization term in the optimization problem, so that the policy is incentivized to explore more. Other than that, it uses a few tricks more. We will not go into much detail, but SAC uses the double Q-Learning and the target network trick that have already been introduced. There are other new tricks: There is no target policy, as in DDPG or TD3, and the policy is not explicitly smoothed (but it is smoothed implicitly because of its stochasticity).

3.5 Experience replay algorithms

The concept of experience replay in AI is grounded in findings in neuroscience, as discussed in Chapter 2. However, the details of how an algorithm recalls experience from the memory are not clear, and it has evolved from the most simple instances to more complicated ones during the history of RL. The result of this evolution of methods leaves us with a wide array of different algorithmic techniques that are based on different intuitions to make agents learn faster from their earlier experiences.

Chapter 4 will deal with a thorough analysis of how these methods are formally defined and which kind of sampling distributions do they shape. Therefore, this section is focused on introducing the background and intuitions behind some of the most commonly used algorithms.

3.5.1 Uniform experience replay

The most basic and intuitive experience replay algorithm is uniform experience replay (UER). It evolved during the pre-deep RL era, appearing first in Lin, 1992 and reaching widespread attention in its application to the DQN algorithm (Mnih et al., 2015).

The ideas behind UER are clear. Given that no more information is known about the nature of the transitions stored in the replay buffer, *sample experience uniformly at random from the replay buffer*. This is probably the first way computer scientists would go on to implement experience replay from its neuroscience description, and it was certainly the first way in which it was implemented.

3.5.2 Prioritized experience replay

In a true ML-like fashion, after DQN (Mnih et al., 2015) received attention from the ML community and the mainstream press alike, obtaining good results and setting up a benchmark for others to compete in (M. G. Bellemare et al., 2013), a large number of researchers devoted their research to get to increase the performance of these algorithms. One of the main points that were ripe for improvement was experience replay.

Schaul et al., 2016 proposed prioritized experience replay (PER) based on a simple idea. The algorithm *prioritizes sampling of actions which have higher potential for learning*. While the *learning potential* (or learning opportunity) concept is a bit obscure, there was an easily available proxy that could be used straightaway. They selected the temporal difference (TD) error as a measure of the amount of learning a specific transition provides. Using a few more regularization tricks to reduce bias in this sampling procedure, they obtained better results than UER in 41 of 49 games in M. G. Bellemare et al., 2013.

More recent research (Fujimoto, Meger, Precup, et al., 2021) suggests that while the prioritization scheme in PER may be useful in some domains, there are several others where the TD error $Q - \mathcal{T}Q$ is not a valid proxy for $Q - Q^*$. $Q - Q^*$ is the oracle error and should be considered the learning error and the learning potential error in a setting where a perfect oracle is known.

PER can be applied in two ways

There is a detail in PER in particular and in all prioritized sampling strategies in general that is often overlooked. There are two ways in which prioritized sampling can be implemented: one, sampling from the memory in the probabilities that the method defines, another, sampling uniformly from the buffer and weighting the losses using an importance sampling coefficient. This point was originally made in Fujimoto, Meger, and Precup, 2020 but has its roots in the theory of importance sampling, a widely known method in statistics.

Implementing methods using these techniques involves a much cleaner design where only the importance sampling ratio in the loss changes for every method.

3.5.3 Backward updates

Based on the theory of tabular RL, there is intuition that updating backward through a trajectory may be the best way in which rewards can propagate. By starting from a terminal state that has a TD error of 0 (there are no states after it, therefore $V(s) = 0$), values are propagated backward through the trajectory, thus reducing the error due to a bad initial value estimate. This idea is the one behind *episodic backward update* (EBU) (Lee, Sungik, and Chung, 2019).

Even though the intuition behind backward updates makes sense and it is indeed very valuable in tabular settings—where we are sure there are no function approximation errors—its benefits cease to exist in more complex deep RL settings. Here, updates to the Q-function are done in batches of a few dozens of examples, so the sequential update ordering loses its meaning. Additionally, most of the errors will originate from the function approximation instead of a bad value estimate in a state. However, it has the desirable property of reinforcing highly-rewarding trajectories, so that it still can deliver large improvements in longer-horizon settings where rewards are sparse and involve reaching a goal.

3.5.4 Topological replay algorithms

Some methods proposed in recent years attempt the organization of experience into a graph. Graphical representations are information-rich, and a large set of well-known graph algorithms can be directly applied to them. Several existing works propose using these representations for different purposes: planning a high-level trajectory in the environment (Emmons et al., 2020; Eysenbach, Salakhutdinov, and Levine, 2019; Savinov, Dosovitskiy, and Koltun, 2018), or designing better experience replay strategies (Hong et al., 2021; Jiang et al., 2022). We are particularly interested in the research presented in Hong et al., 2021, which proposes a new algorithm called topological experience replay (TER).

TER designs a new sampling strategy similar to EBU in that it aims to replay experience backward from the terminal states towards the starting positions in the episodes. The main difference is that TER stores its memory in a graph that can connect different episodes, resulting in a richer source of information that allows for search algorithms to be applied to it. This is exactly what TER does: it does a breadth-first search starting from the terminal states, propagating low-error value approximations across different trajectories that have been weaved together in the graph.

The results of the paper show that TER can outperform all known methods in sparse reward tasks with long horizons, which are difficult for most existing experience replay algorithms. In these environments, TER replays the experience in the same distribution as an optimal policy would. As a result, in settings where other algorithms are unable to find a good transition to recall (due to the lack of rewards), TER knows exactly what to replay.

However, the performance of TER suffers a lot when an environment does not satisfy the terminal state assumption: rewards are not concentrated in a few states where episodes end but are scattered in different places along an episode's progress. Most RL benchmarks have some sort of dense reward designed into them to make learning easier, making TER less competitive and applicable when considering most areas of RL research.

3.5.5 Analysis of experience replay

Several papers have conducted analyses of experience replay algorithms. Their goal is to find a common result across several algorithms and hyperparameters that can bring intuition into what makes an agent learn faster with regard to the choice of how it replays its memory.

In Fedus et al., 2020, researchers investigate how the size of the replay buffer and the age of the oldest policy that recorded experience in the replay buffer influence performance. They find that both incrementing the size of the replay buffer and making sure experience is collected by policies that are not too old help improve learning. The article also provides insights on the importance of using n-step returns to make learning more reliable, especially in settings where the replay buffer is made very large.

Another article by Fu et al., 2019 studies several aspects of deep Q-learning. One of their findings involves empirically testing out several experience replay sampling distributions to test their performance. Their findings show that strategies with higher state entropy and even coverage of state-action pairs in the replay buffer obtain a better performance. These results can be related to the replay buffer imbalance problem discussed in Chapter 2 and it will influence some of the decisions taken in Chapter 4 and **??**.

Chapter 4

Analyzing replay strategies by looking at sampling distributions

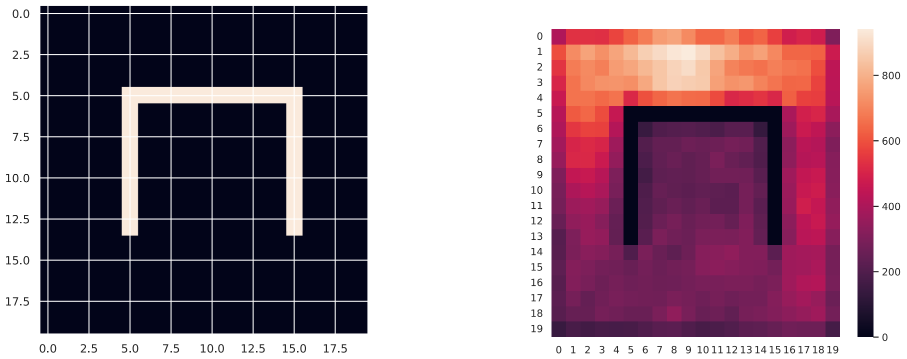
This chapter is focused on introducing the new lens by which we characterize a replay strategy. As suggested by the title and several sections in the introduction, sampling distribution is a useful way of studying the difference between replay buffer sampling algorithms. The following sections will include an overview of the intuition behind measuring state replay probabilities, a formalization of the proposed arguments, followed by a description of the method used to measure the distributions, all leading to a study of replay frequencies in the most commonly found replay strategies.

4.1 Why care about replay frequencies?

At this point, it is perfectly reasonable to ask *why we should bother about the shape of the replay frequencies* in different replay algorithms. In fact, the most widely used memory replay methods are based on other arguments (for example, assume that there is no information on which transitions are better for the learner to see (Lin, 1992; Mnih et al., 2015), or replay those transitions in which the agent has a higher error (Schaul et al., 2016)). As a result, it may seem that transition replay distributions are not as relevant. However, there are a few observations to be made about these claims.

While it is true that the main driving argument behind the proposal of replay algorithms such as uniform experience replay (UER) or prioritized experience replay (PER) (Schaul et al., 2016) is something different, there is something they all have in common: By changing the way in which transitions are sampled, they *unavoidably change the probability distribution* from which they take samples. This is quite evident in the case of UER, where the chosen distribution is uniform in all the replay buffer, so that the probability of sampling a transition is $p(t) = (|\mathcal{R}|)^{-1}$.

In the case of PER, the distribution changes depending on the errors the model makes, and it is therefore not possible to arbitrarily define the distribution in closed form. But even though



a / U-maze toy environment.

b / U-maze state frequency in the buffer.

Figure 4.1 / An example of biased state distribution in the replay buffer .

they acknowledge that the PER replay mechanism can be useful in reweighting sampling distributions for more efficient learning and show it using a simple example of imbalanced supervised learning, they do not analyze which kind of replay priority distributions are induced by the method.

The idea that all replay strategies shape a particular replay distribution alone may justify the need to study their relationships and how they affect performance. But there is another argument that further encourages research on these topics. The contents of the replay buffer consist of *all data collected by previous policies* (or previous versions of the training policy). This means that, even if the training algorithms are off-policy, the performance of previous policies influence current training because their biases are reflected in the data. If a certain past policy was biased towards a certain region of the state space, the region will be overrepresented in the replay buffer. The opposite is also true: regions hardly explored by previous policies will be underrepresented.

Having data by previous policies in the replay buffer is unavoidable. This data will most probably be biased in the state space. As a result, the burden of regularizing the state space distribution relies entirely on the experience replay algorithm[1]. To better understand the issue of bias in the state space we will make use of an example. Consider the U-maze environment, shown in figure 4.1a. In this domain the agents start from the top, and they have to reach a goal in the middle of the U-shaped wall. If the original policies have a hard time exploring all the state space, the state of the replay buffer might be similar to the density in figure 4.1b. By replaying transitions uniformly from the replay buffer, the agent will most frequently see transitions corresponding to the top of the environment, which carry little to no information about the goal.

It is frequently observed that for an agent to start learning in a sparse reward setting, an initial exploration phase is required until the agent reaches some reward by chance. If once the

[1] Regularization can also be applied by adding importance sampling coefficients to the loss, as shown by Fujimoto, Meger, and Precup, 2020

successful transitions are in the replay buffer and they are replayed with the same probability as all the other ones, learning will be harder. Some methods try to solve this issue. TER (Hong et al., 2021) addresses it directly, and PER solves it indirectly by assuming that a newly found high-reward state will lead to a high TD-error.

4.2 Formalization of the arguments

The arguments proposed above may have convinced the reader of the importance of studying the sampling distributions in experience replay. Some other reader may argue that the ideas are hand-wavy and a more structured derivation of the motives behind the analysis is needed. For the latter, and to set a working framework to allow us to work, we will try to lay the concepts in a mathematical way.

4.2.1 Experience replay algorithms can be understood as choosing a sampling distribution

The first idea is that all different experience replay methods change the sampling distribution. In general, this is their only difference and how they are defined. For instance, a *complete* definition of some experience replay algorithms is:

Definition 4.1 (Uniform Experience Replay) *Sample a transition t from the replay buffer $\mathcal{R}$ with equal probability for all elements. If there are $|\mathcal{R}|$ elements, the probability of sampling a transition i is $\Pr_{UER}(i) = (|\mathcal{R}|)^{-1}$.*

Definition 4.2 (Prioritized Experience Replay) *Assign an error value to every transition i in the replay buffer $\mathcal{R}$ according to the last TD error obtained when using the transition δ_i. Define a probability distribution assigning a probability $\Pr_{PER}(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ where $p_i = |\delta_i| + \varepsilon$. Sample a transition from $\mathcal{R}$ using $\Pr_{PER}(i)$.*

Definition 4.3 (Topological Experience Replay) *Define the terminal set $\mathcal{T} \subset \mathcal{R}$ as the set of transitions that have $done(t) = 1$. Sample a transition from $\mathcal{T}$ with probability $\Pr(i) = (|\mathcal{T}|)^{-1}$. Expand using breadth-first search from the selected transition backwards through the graph and add all sampled transitions to the graph.*

As we can see, most distributions can be characterized completely using the sampling distribution. TER is more complex, but once the data structure and algorithm choices are fixed, it is completely defined by the starting sampling distribution.

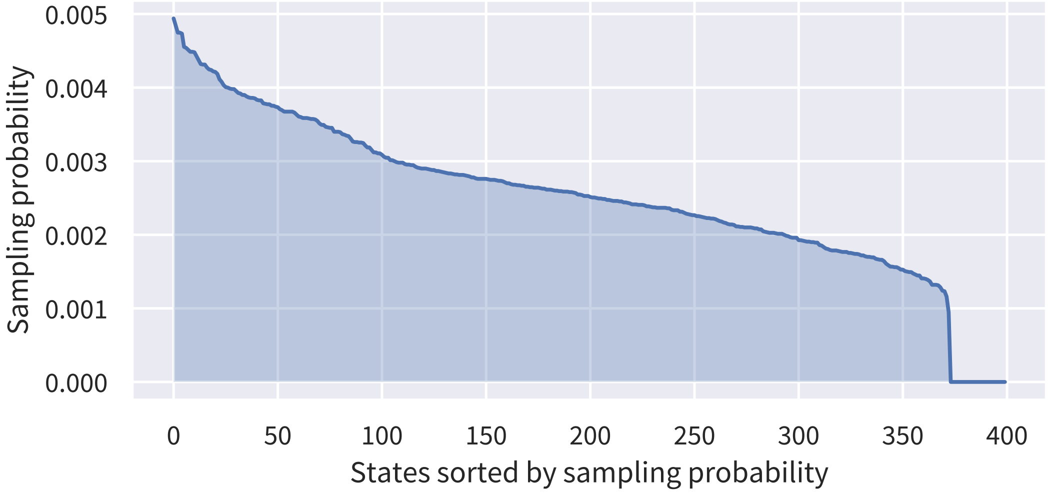


Figure 4.2 / State space sampling distribution when performing random exploration and uniform sampling in the U-maze example (figure 4.1).

4.2.2 Replay buffer contents define a distribution over the state space

The second argument is that the biases of previous policies will influence the contents of the replay buffer. This can be easily seen in the toy example in figure 4.1. Previous policies will determine which parts of the state space are more represented in the replay buffer, and this is unavoidable. By sampling transitions, we will eventually get a state sampling distribution $\Pr(S)$ associated to states in the transitions sampled $s_t = \text{state}(t)$. If exploration is perfectly uniform in all state space $\mathcal{S}$, then $\Pr(S) = (\mathcal{S})^{-1}$.

However, this is hardly ever true. For example, take the U-maze example in Figure 4.1. By performing random exploration and sampling transitions uniformly, the state sampling probability distribution obtained would be similar to the one depicted in figure 4.2. When the environment has a larger state space and more complex dynamics, this effect only increases. The exploration biases of earlier policies are stored in the buffer, and sampling will show skewed distributions similar to the one in the example.

Experience replay algorithms different from UER shape the state sampling distribution in other ways, some more restrictive than others. We are interested in studying how strict alternative replay strategies are in changing the state replay distribution.

4.2.3 Replay techniques induce different state sampling distributions

The last subsections have introduced two different probabilities: $\Pr_{\text{ER}}(T)$ and $\Pr(S)$. We want to investigate more in detail how a replay algorithm defined by $\Pr_{\text{ER}}(T)$ influences $\Pr(S)$.

Trivially, the probability of sampling a state $\Pr(S = s)$ can be dissected into:

$$\Pr(S) = \sum_t \Pr(S, T = t) = \sum_t \Pr(S \mid T = t)\Pr_{\text{ER}}(T = t) \tag{4.24}$$

Where the first term $\Pr(S = s \mid T = t) = \mathbb{1}\left[\text{state}(t) = s\right]$, where $\mathbb{1}$ is the indicator function and the second term $\Pr_{\text{ER}}(T)$ depends on the experience replay method used. Formally, we can study how an experience replay algorithm changes the state sampling policy by studying the choice of $\Pr_{\text{ER}}(T)$ and how it modifies the overall distribution.

An illustrative example is UER. Here, $\Pr_{\text{UER}}(T)$ is uniform. By applying this to equation [4.24] we obtain:

$$\begin{aligned}\Pr(S = s) &= \sum_t \Pr(S \mid T = t)\frac{1}{|\mathcal{R}|} \\ &= \frac{1}{|\mathcal{R}|}\sum_t \mathbb{1}\left[\text{state}(t) = s\right] \\ &= \frac{1}{|\mathcal{R}|}\#\left\{t \mid \text{state}(t) = s\right\}\end{aligned} \tag{4.25}$$

The result is the state distribution in the replay buffer, as can be intuitively understood by looking at the graph in figure 4.2. When the replay method is uniform, all the elements in the definition of the state distribution are constant with respect to t and the sum becomes a counter of transitions that have state s. This means that the graph we see in figure 4.2 is both the state visitation frequency $\#\left\{t \mid \text{state}(t) = s\right\}$ in the replay buffer, and the state sampling distribution obtained by UER in the example.

Let us try to assess the same thing with PER. Arguably, the largest change in the state sampling distribution will occur when the distribution $\Pr(S)$ is uniform, because it would mean that the sampling mechanism has eliminated the biases introduced by previous policies. Therefore, let us find which shape the TD errors should have for different kinds of state visitation frequency in the replay buffer.

A simplified distribution for PER is:

$$\Pr_{\text{PER}}(i) = \frac{p_i}{\sum_k p_k} \propto p_i,\ p_i = |\delta_i| + \varepsilon \simeq |\delta_i| \tag{4.26}$$

Here δ_i is the last seen TD error for transition i.

Combining equations [4.24] and [4.26], we get:

$$\begin{aligned}\Pr(S) &\simeq \sum_t \Pr(S \mid T = t)\frac{|\delta_t|}{\sum_k |\delta_k|} \\ &= \sum_t \mathbb{1}\left[\text{state}(t) = s\right]\frac{|\delta_t|}{\sum_k |\delta_k|} \\ &\propto \sum_t \mathbb{1}\left[\text{state}(t) = s\right]|\delta_t|\end{aligned} \tag{4.27}$$

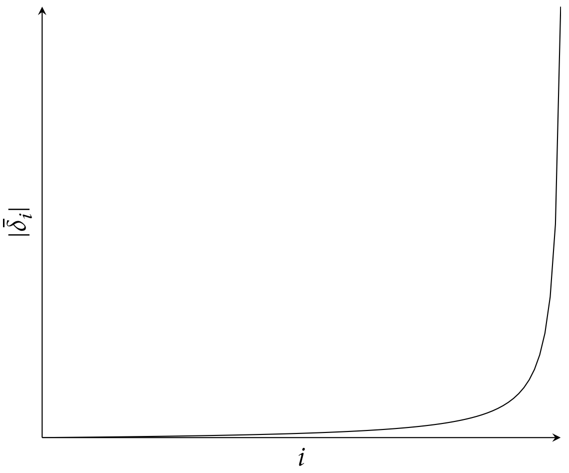


Figure 4.3 / Shape of TD errors as a function of the sorted state indices for the linear state distribution case in PER.

We need to make an assumption to continue on. Let's assume all transitions that have the same state have the same TD error $\bar{\delta}_s$:

$$\delta_t = \bar{\delta}_s, \quad \forall t \in \{t \mid \text{state}(t) = s\} \tag{4.28}$$

The expression then becomes:

$$\begin{aligned} \Pr(S = s) &\propto \sum_t \mathbb{1}\left[\text{state}(t) = s\right] |\bar{\delta}_s| \\ &= |\bar{\delta}_s| \sum_t \mathbb{1}\left[\text{state}(t) = s\right] \\ &= |\bar{\delta}_s| \#\{t \mid \text{state}(t) = s\} \end{aligned} \tag{4.29}$$

If we think about the state frequency in the buffer ($\#\{t \mid \text{state}(t) = s\}$) as a sorted frequency plot like the one in figure 4.2, indexed by s, we can devise different shapes for it and see how $\bar{\delta}_s$ adapts to make the overall distribution uniform.

For instance, if the state representation resulting from exploration is linearly decreasing $\#\{t \mid \text{state}(t) = i\} \propto -i + |\mathcal{S}|$, we obtain:

$$\begin{aligned} (-i + |\mathcal{S}|)|\bar{\delta}_i| &\propto \Pr(S) = \frac{1}{|\mathcal{S}|} \ \forall i \in \mathcal{S} \\ \Rightarrow (-i + |\mathcal{S}|)|\bar{\delta}_i| &\propto 1 \\ \Rightarrow |\bar{\delta}_i| &\propto \frac{1}{-i + |\mathcal{S}|} \end{aligned} \tag{4.30}$$

Figure 4.3 shows a basic plot of this function. The resulting TD error distribution is quite unlikely to happen in real situations, because it shows a very prominent spike for the last

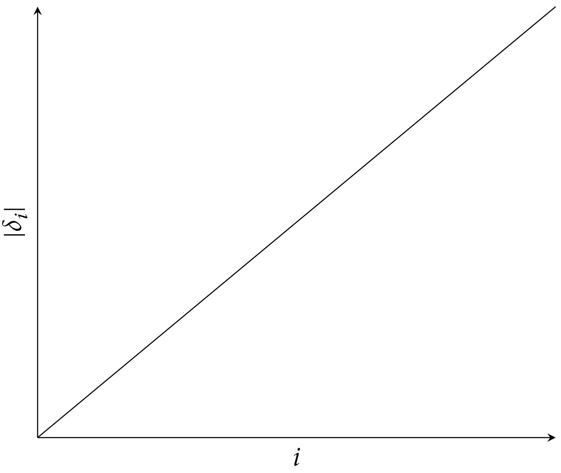


Figure 4.4 / Shape of TD errors as a function of the sorted state indices for the parabolic state distribution case in PER.

few transitions. If this were to happen, there would be a stark contrast between the errors of common states and the errors for lesser-seen ones. With a simple gradient update of the large errors seen in the plot, the distribution would flatten easily. As a result, we can conclude that PER is unable to obtain a uniform state sampling distribution if the state frequency in the replay buffer follows a linear trend.

In practice, we find that state distributions in the buffer are severely skewed. Instead of linearly decreasing, the shape of these distributions is more akin to show a rapidly decaying curve. Example plots of these distributions in real use-cases will be seen in future sections. A function that resembles the states found in practice is, for instance, $\frac{1}{x+1}$. Following the same procedure as above, we get that:

$$|\bar{\delta}_i| \propto i + 1 \qquad [4.31]$$

Figure 4.4 shows the shape of this error distribution. This time, it is more feasible. The errors are well balanced, with no large contrast between more frequent and less frequent states, and this shape is one that we can expect to see in practice. However, as learning advances, the TD error distributions flatten easily, with most variance due to learning noise and the effect of the parameters on the samples. Therefore, while this is a reasonable distribution for $|\bar{\delta}|$, it is unlikely that it is verified once learning advances and errors are no longer due to the frequency with which states are analyzed.

From the two example cases above we can conclude that while some distributions can happen in early learning phases, it is unlikely that they are maintained during all training. So PER is theoretically able to maintain a uniform state sampling distribution (as opposed to UER, that cannot do it), but it is unlikely that this happens in practice. We will see in later empirical analyses in some environments that PER indeed does not have a uniform state replay distribution.

To end this section, let us quickly see how a replay method that aims to always enforce a uniform state replay distribution would be designed. We will call it Uniform State Replay (USR). In this case, the goal is to achieve $\Pr(S) = \frac{1}{|\mathcal{S}|}, \forall s \in \mathcal{S}$. By imposing this on the equation defining the state sampling distribution [4.24]:

$$
\begin{aligned}
\Pr(S = s) = \frac{1}{|\mathcal{S}|} &= \sum_t \mathbb{1}\left[\text{state}(t) = s\right] \Pr_{\text{USR}}(t) \\
&= \Pr_{\text{USR}}(t) \sum_t \mathbb{1}\left[\text{state}(t) = s\right] \\
&= \Pr_{\text{USR}}(t) \# \left\{t \mid \text{state}(t) = s\right\} \\
\Rightarrow \Pr_{\text{USR}}(t) &= \frac{(|\mathcal{S}|)^{-1}}{\# \left\{t \mid \text{state}(t) = s\right\}}
\end{aligned}
\qquad [4.32]
$$

As the reader may have expected, we have to prioritize the transitions with the inverse of the state frequency in the replay buffer. Having obtained this, we can now define a new replay algorithm.

Definition 4.4 (Uniform State Replay (USR)) *Sample a transition t from the replay buffer $\mathcal{R}$, prioritizing them by the probability* $\Pr_{USR}(t) = \frac{(|\mathcal{S}|)^{-1}}{\#\{t' | state(t) = state(t')\}}$.

The analysis done here is a study of how sampling distributions and replay buffer contents shape the state replay, for the two most common experience replay algorithms in the literature. A new sampling algorithm has been proposed, which will be used in the future. While all analyses have been performed using states, the same procedure could be done again using state-action pairs. States have been used for simplicity, but state-action pairs can be more useful in practice because they have much more similar values and errors than mere states.

4.3 Analysis method

This section will introduce the tools we will use to study how the algorithms behave in terms of the sampling distributions introduced above. The goal of these tools is for them to be simple so that we can understand clearly what the results communicate, but also powerful because the conclusions we draw from them will determine how we design algorithms in the future.

As mentioned before, we are interested in analyzing experience replay algorithms from their sampling distributions. In particular, the probability distributions we will be looking at are the transition, state, and state-action sampling distributions. These give insights into the state-space coverage and prioritization of every replay method analyzed. There are other angles by which replay could be studied, namely the order of the samples and batches, and the prioritization of good transitions for faster learning. However, these will be left out from this study, because they are arguably independent from each other and thus standalone analyses can be performed for the different dimensions of the problem.

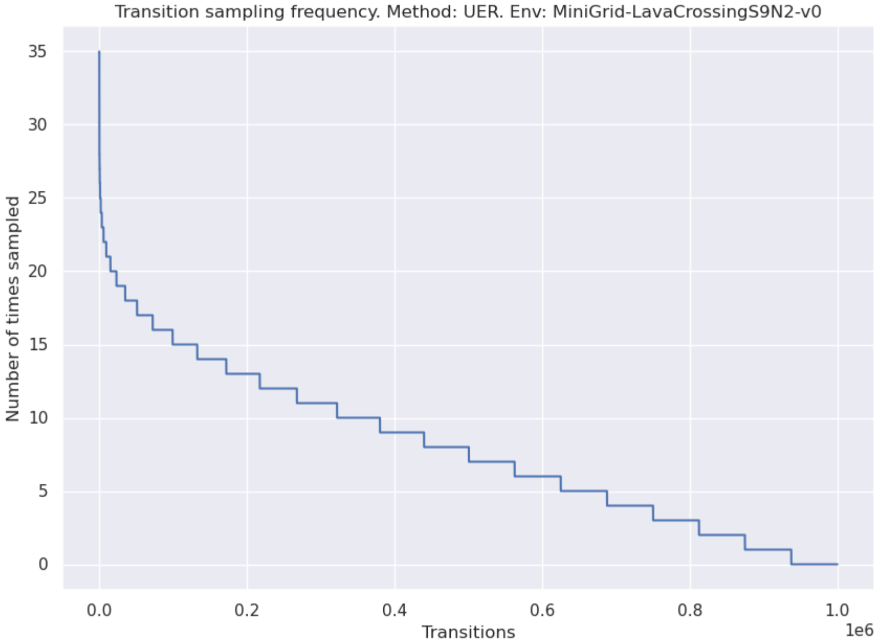


Figure 4.5 / Example of a sampling distribution plot for transitions, using UER.

4.3.1 Sampling distribution plots

The main study of sampling behavior for experience replay algorithms is done by plotting. They are the preferred method because they qualitative nature makes them easy to interpret and to extract conclusions from them. Some of these plots have already been introduced in previous sections of this same chapter. However, there has been no formal introduction yet. Although quite simple, there are a few remarkable details that must be cleared before proceeding.

All these plots show the empirical sampling distribution or, equivalently, the replay frequencies recorded for every case. Let's use transition replay plots as an example. The replay frequencies are obtained by seeing every single transition sampled from the replay buffer, and counting how many times every transition is seen. Now, plotting the frequency against the replay buffer indices, the result would be a noise-looking plot, because every transition is replayed differently. To get a graph that makes sense, the frequencies are sorted from greater to lower and plotted in this order. Now the distribution is seen as a downward-looking curve, as the most frequent transitions are on the left and the less frequent ones are on the right of the picture. An example of the result of this procedure is shown in figure 4.5.

The same procedure can be repeated for states and state-action pairs. However, there is a slightly more complicated procedure for these cases. The algorithms will be tested in discrete state-space environments, but using images as the state representation. It is natural to think that—even though images are a continuous-valued format—when the environment is in the same state in two different situations it will output the same image. To convert images to an index as in the transitions, they are first projected to a lower-dimensional space and then mapped to a position in a map using a hash function. In a discrete state space it is likely that two different states are different enough, therefore we do not need to care about the properties of random projection (Dasgupta and Anupam Gupta, 2003) and hashing collisions. It is in this hash map that the counting procedure is performed, allowing the user to plot the frequencies in the same way as the transitions.

This visualizations are useful to determine whether replay is performed uniformly for all transitions or states in the buffer. It also allows to see the differences between replay algorithms. That is why it is a good method for the analysis task.

4.3.2 Quantitative analysis

It is desirable to have some quantitative way to measure and compare performance. While the graphs provide a good enough way to extract useful insights for this study, we can extract statistics to have a finer-grained view on the distributions.

To start with, we will measure the entropy of the sampling distributions. Entropy stands for the amount of information that a new sample gives us. It will be maximized when the sampling distribution is uniform (flat), because we know less about which sample is going to come next. It is computed as:

$$\mathrm{H}\left[S\right] = -\sum_{i} p(i) \log p(i) \qquad [4.33]$$

Where $p(i)$ is the normalized sampling distribution evaluated at sample i. Ideally, state and state-action sampling entropy should be as high as possible, because that guarantees a good state space coverage.

4.4 Empirical analysis of experience replay algorithms

Having introduced all tools and reasoning behind the study of experience replay algorithms from the point of view of sampling distributions, we can now start performing an analysis of the most common replay strategies. The algorithms selected for this analysis are uniform experience replay (UER) (Lin, 1992; Mnih et al., 2015), prioritized experience replay (PER) (Schaul et al., 2016), topological experience replay (TER) (Hong et al., 2021), and uniform state replay (USR), defined in definition 4.4.

We will be using several RL environments to test the mentioned algorithms. By analyzing results over multiple environments, ad hoc results that depend on a particular setting become less important, facilitating the goal of understanding the bigger picture. In our case, the environments used will be several procedurally-generated scenarios from the MiniGrid suite (Chevalier-Boisvert, Willems, and Pal, 2018) and the full MinAtar suite (Young and Tian,

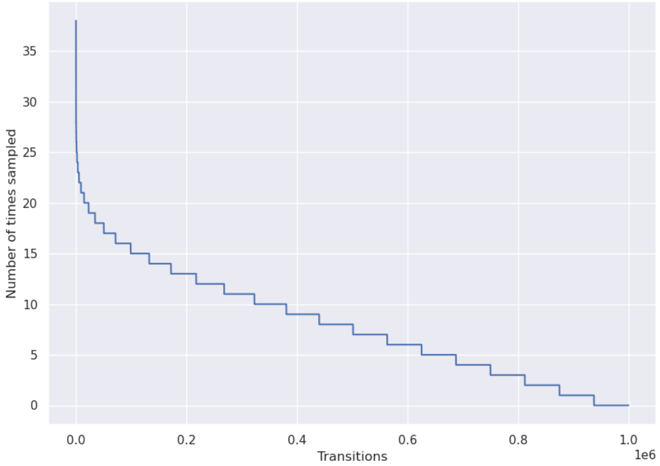


Figure 4.6a / Transition sampling frequencies for UER. Sampling is evenly distributed along all transitions .

2019). This array of setups represents the set of image-based environments with discrete state space and sparse rewards well enough for us to draw conclusions. The environments in the MiniGrid suite are very structured in their state space, requiring an almost-episodic memory, for which TER is a good fit. Most of the MinAtar environments are less structured, with harder dynamics. TER is not that good of a fit for these cases, therefore it is easier for PER to shine.

For the sake of space and simplicity, only results for some environments will be presented here. Complete results for all domains in our testing set can be found in Appendix A.

The selected environments are:

- Lava Crossing (hard), from MiniGrid
- Freeway, from MinAtar

4.4.1 Results on Lava Crossing

Figure 4.6a shows the transition sampling frequency observed when running the different algorithms. And figure 4.7a shows the state sampling frequency, instead. The state-action sampling frequency has not been plotted for the sake of simplicity.

By focusing on the plots a very clear pattern emerges. In the case of transition sampling distributions, UER and PER have a much *heavier tail* in the distribution (Figures 4.6a and 4.6b). TER and USR (Figures 4.6c and 4.6d) show the opposite, very few transitions are sampled a lot, but then the sampling frequency decays very fast. In the case of the state distribution (Figures 4.7a to 4.7d), the effect is quite clearly the opposite. UER and PER are heavily skewed in terms of state sampling, while TER and—especially—USR show much more even coverage of the state space when sampling.

This pattern empirically reveals the facts that have been previously outlined on several

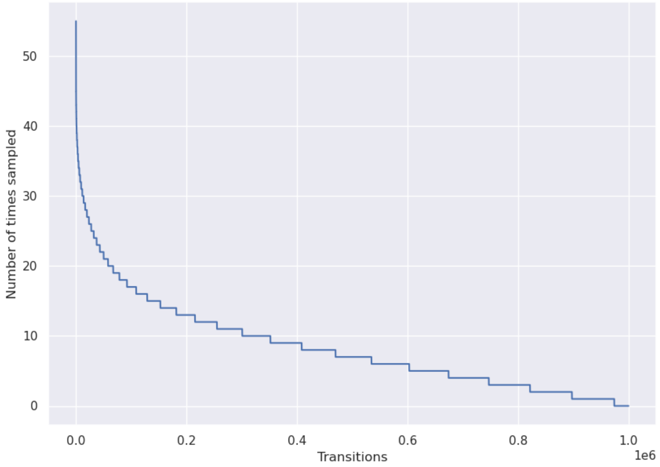


Figure 4.6b / Transition sampling frequencies for PER. Transitions are prioritized but are still somewhat uniformly sampled.

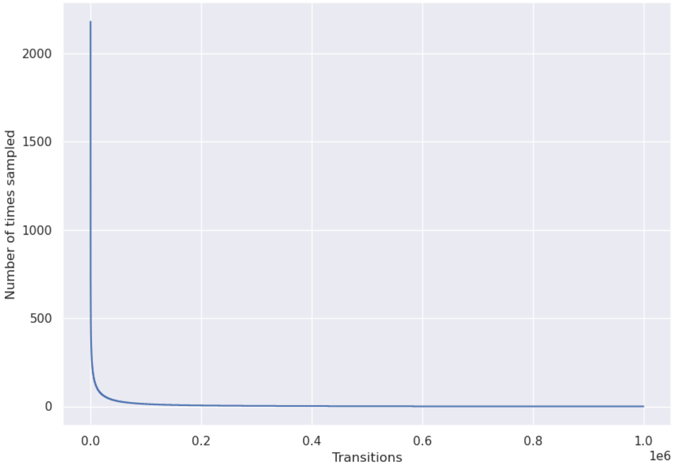


Figure 4.6c / Transition sampling frequencies for TER. Sampling is done in the graph of states. Transitions are very unevenly sampled.

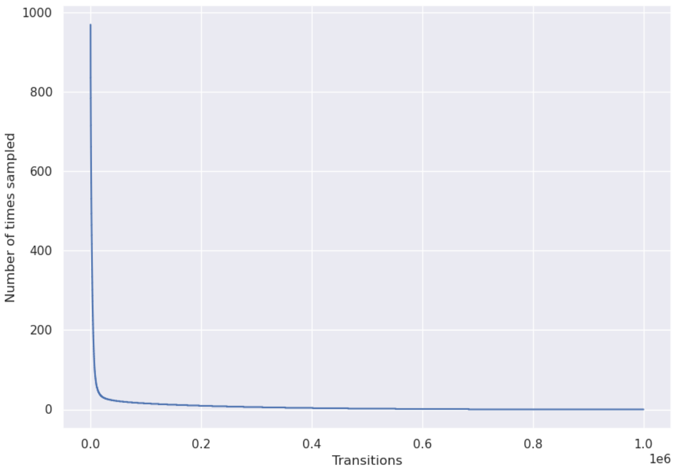


Figure 4.6d / Transition sampling frequencies for USR. First, states are sampled, then transitions belonging to those states are found .

occasions. Contents of the replay buffer are the result of the experience of all previously used policies and their exploration capabilities. It is natural to expect their state-space coverage to be uneven. These plots confirm the expectations: when sampling uniformly from the replay buffer (UER), transitions are sampled uniformly, but states are found to be sampled with a large bias towards only a few of them. This pattern happens because the bulk of transitions are focused on just a few states. As motivated previously, this uneven state coverage may lead to issues during learning due to a bias to train on uninteresting states, and the difficulty to sample the right transitions once they are in the replay buffer.

Although PER has a different sampling prioritization scheme[2], it still relies on the overall *list of transitions* data structure. It would need a severely skewed TD error distribution to sample states more uniformly across the state space, as described in Section 4.2.3. The explanations above are supported by the empirical results seen in Figures 4.6b and 4.7b.

TER and USR, viewed from a very high level, take samples from a different data structure. While the sampling mechanisms are different, both algorithms sample transitions from a state-based data structure. This involves a more even coverage of the state space. Moreover, in the case of USR, it has been explicitly designed to sample uniformly from all states in the replay buffer. In Figures 4.7c and 4.7d we clearly see these effects. In terms of transition sampling we see a clear effect too. Because most transitions are focused in just a few states, we find a much more skewed transition sampling distribution. The bulk of states contain a small amount of transitions, thus they are sampled much more frequently than transitions in states that have a large number of them.

The performance in terms of mean rewards as training progresses is shown in Figure 4.8. We

[2] Actually, PER weights the prioritization so it's less strict than the straightforward application of the probabilities proposed in the paper. The goal of the weighting is to find a sweet spot between PER and UER. As a result, the sampling procedure becomes quite similar to UER. This can be found in the pseudocode of the Schaul et al., 2016 paper.

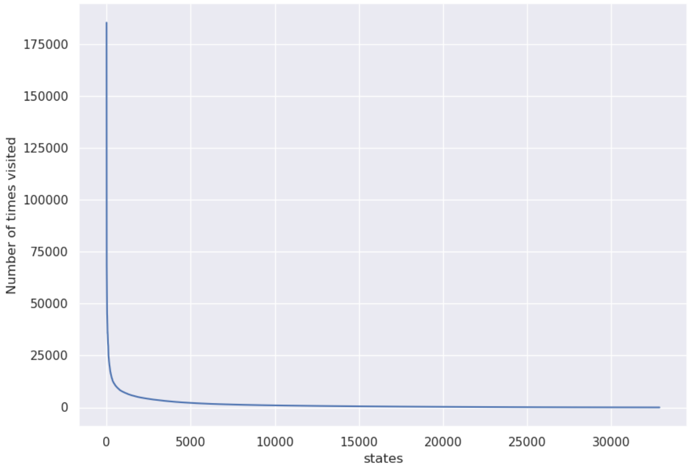


Figure 4.7a / State sampling frequencies for UER.

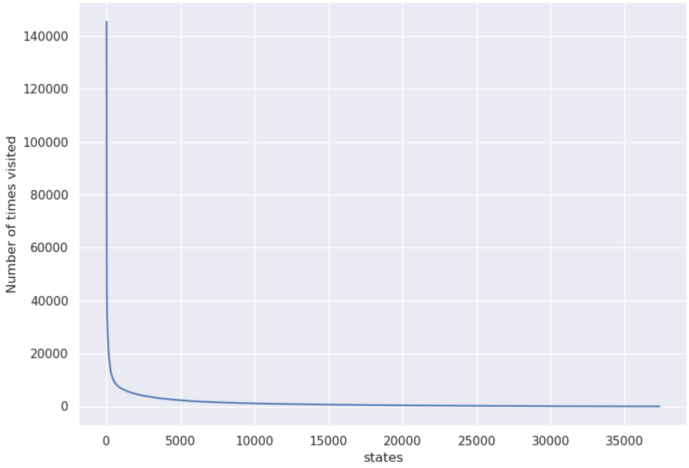


Figure 4.7b / State sampling frequencies for PER.

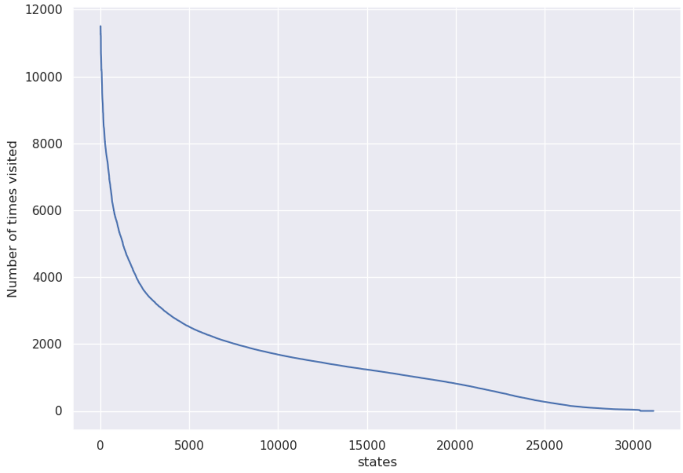


Figure 4.7c / State sampling frequencies for TER.

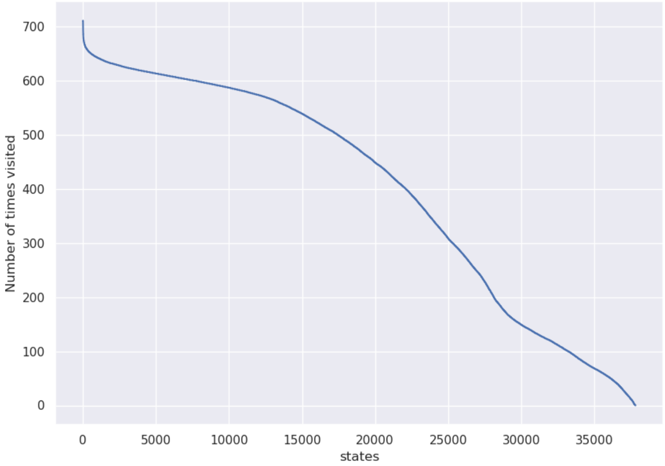


Figure 4.7d / State sampling frequencies for USR.

can see that in this case, USR and TER achieve similar performance, with TER being slightly more sample-efficient. UER and PER, instead, cannot achieve any success in Lava Crossing, obtaining a reward of close to -300 during the progression of the training procedure. We hypothesize that these results may be due to a strong bias towards sampling states close to the initial position of the environment, thus making it hard to propagate rewards backwards through the MDP.

4.4.2 Results on Freeway

Results in terms of transition and state frequencies can be found in the appendices, because they would take up too much space here. However, it is worth mentioning that there is a slight tendency of all the methods to have a more similar transition sampling distribution. This can be seen for USR in Figure 4.9. This suggests that the state coverage in transitions is similar across a large number of states. The most likely reason for this to happen is that the state space is very large and thus all states have a small number of transitions because they are not repeated often.

The performance in this environment also reflects these findings. Due to the nature of the domain—states are rarely repeated—all methods have relatively similar sampling distributions. This results in almost indistinguishable performance by all algorithms, as seen in Figure 4.10. Seeing the outcome of the application of these algorithms to the environment reinforces our starting hypotheses: experience replay algorithms are highly dependent on their sampling distributions. Here, several algorithms are tried, which use different underlying data structures, sample selection, and prioritization schemes. However, when the environment causes the different sampling strategies to degrade to an even distribution in the state space case, the performance of all algorithms is mostly the same.

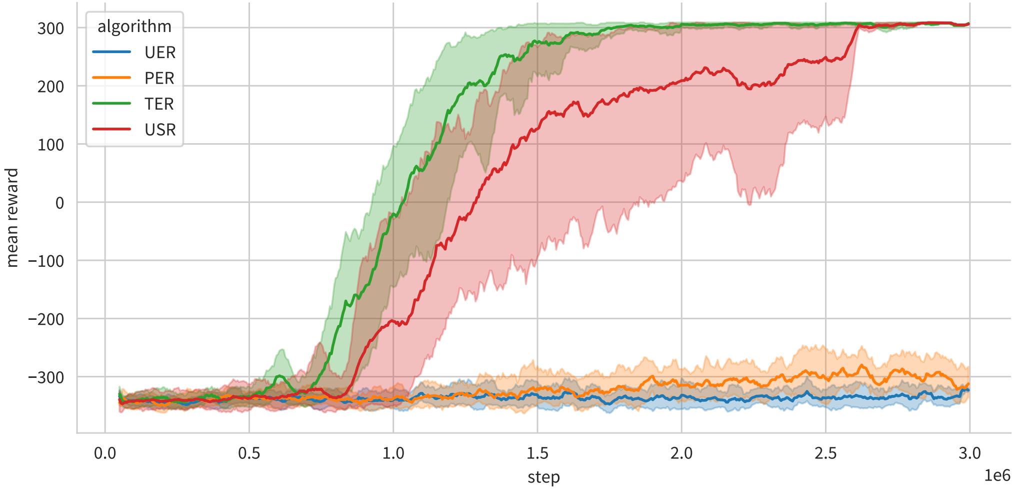


Figure 4.8 / Mean reward obtained by the experience replay algorithms in Lava Crossing (hard). Results are aggregated over 3 random seeds.

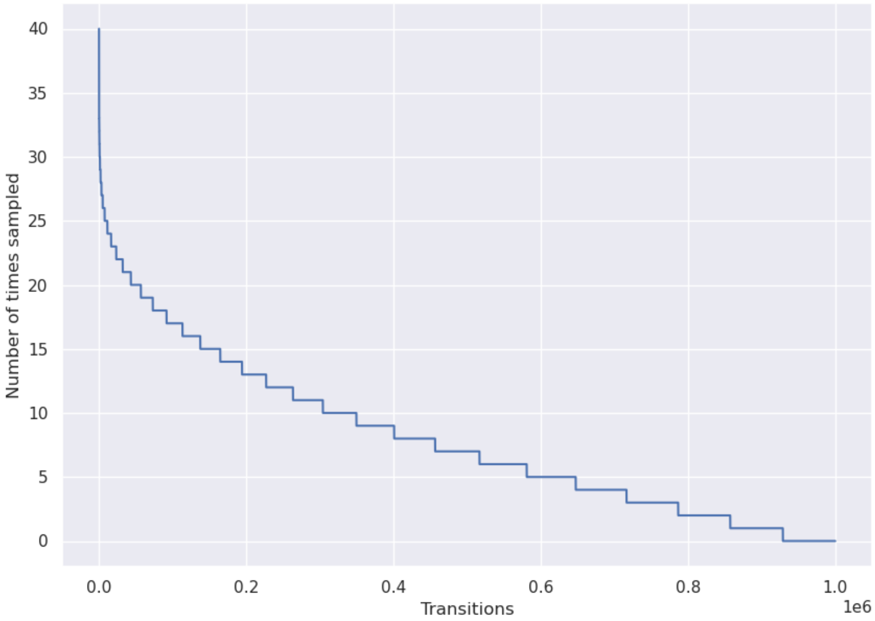


Figure 4.9 / Transition sampling frequencies for USR in Freeway.

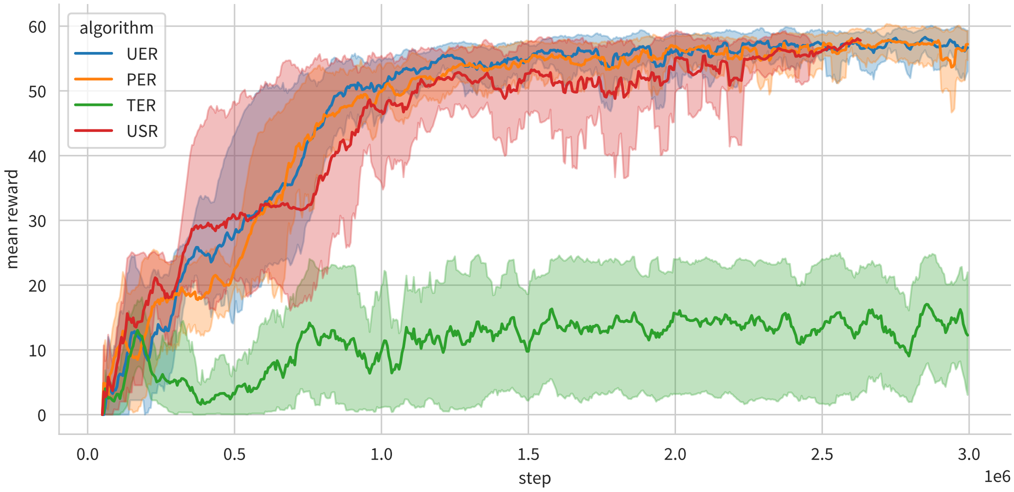


Figure 4.10 / Mean reward obtained by the experience replay algorithms in MinAtar Freeway. Results are aggregated over 3 random seeds.

Chapter 5

New algorithms for experience replay

This chapter is focused on designing new algorithms for experience replay using the insights extracted from both the literature and the analyses in Chapter 4. They will be motivated using ideas about sampling distributions and better graphical representations of the buffer, defined according to these ideas, and their performance will be assessed in several diverse RL environments against state of the art baselines.

The baselines used are the same that were presented in Chapter 4: UER, PER and TER in DQN. The domains where the algorithms will be tested have also been described in previous sections. They are some of the procedurally generated environments in MiniGrid (Chevalier-Boisvert, Willems, and Pal, 2018) and the full MinAtar suite (Young and Tian, 2019).

5.1 Uniform State Replay

Uniform State Replay (USR) has already been described in Chapter 4. There it was motivated by the need to have a uniform coverage of the state space, after it was known that PER was unlikely to achieve it. In this section, we will motivate it more, clarify our reasoning as to why it is useful, and then we will discuss the way in which we have implemented it.

Experience replay takes transitions from the replay buffer and replays them. In reality, what is being replayed is a state, action, next state, and reward. The contents of the replay buffer consist of the experience that previous policies have had, with all the biases and preferences that stem from using a suboptimal policy. As a result, there are certain states and actions that are found much more frequently than others, and this is commonly found in practice (see Figure 4.7a). This fact results in an uneven and skewed state sampling distribution when using the most common replay methods, as shown in Chapter 4.

Several scientific articles, both theoretical (Santos et al., 2021) and empirical (Fu et al., 2019), show that the more uniform a replay strategy is when replaying states, the better the sample efficiency of the method. The realization that replay buffers are heavily biased towards certain states, that commonly found replay algorithms cannot fix these distributions when sampling, adding in the fact that the scientific community has confirmed that uniform state space coverage is desired, motivates the need for a new replay algorithm. One that is able to take the data from the replay buffer, tame its biases so that they do not excessively influence the data seen by the learning algorithm, and makes replay as close as possible to uniform over the state space.

Observing these needs, we propose uniform state replay. As stated in Definition 4.4 the goal is to sample transitions with inverse proportion to the number of transitions that have the same state. Mathematically, this is written as $\Pr_{\text{USR}}(t) = \frac{(|\mathcal{S}|)^{-1}}{\#\{t' | \text{state}(t) = \text{state}(t')\}}$. Naturally, this algorithm aims at replaying uniformly over the *empirical* state space, not the true state space. In Santos et al., 2021, the authors give good guarantees to a policy that is uniform in the true state space. However, we believe that after enough data has been collected the empirical state space will be approximately equal to the true one.

5.1.1 Implementation of USR

While the desired probability distribution is fairly straightforward, there are several decisions that can be made to implement this method. We will discuss them and describe what the final implementation of the method is, along with the decisions made.

To start with, there are two distinct ways one could go implement this sampling procedure. The first that comes to mind is to annotate the probability for every transition given the number of transitions that have the same state, and then sample from those probabilities. But another option is possible: one can sample uniformly from the replay buffer, then weight all updates using the importance sampling coefficient $\frac{\Pr_{\text{USR}}(t)}{\Pr_{\text{UER}}(t)}$. The results are asymptotically the same.

Yet, when we implemented a preliminary version using importance sampling coefficients we obtained unstable results. This was due to extremely large coefficient numbers for states that have only one transition (which implies $\Pr_{\text{USR}}(t) >> \Pr_{\text{UER}}(t)$). Although the results should be the same in expectation, these large numbers do not play well with the low number of samples in mini-batches and SGD. Therefore, we decided to implement the sampling distribution directly on the replay buffer.

Opting for this implementation is not as simple. Recording and updating the probability distribution for all transitions can be costly. By making use of a different data structure we can shape the algorithm so it automatically behaves in the way we desire, without the need to maintain an up-to-date record of all the transition sampling probabilities. The data structure chosen for this approach, based on a simplified version of the algorithms for graphical replay of Hong et al., 2021, is grounded on the principle that we want to sample *states*—not transitions—uniformly. At a high level, it maps a state to a list of all transitions that start from that state. When sampling, a state is selected uniformly at random from all states, and then a transition is sampled from the list that corresponds to that state.

Data structure

Let's give a more thorough description, with special emphasis on the two main replay buffer operations: append and sample. In the experiments required for this thesis, we are working with image-based observations of discrete state-space environments. This means that the intrinsic dimensionality of the observations is low, but the observations are high-dimensional images. To enumerate the seen states we use a combination of random projection and hashing. Random projection reduces the dimensionality of samples while keeping their pairwise distances approximately similar. Hashing allows one to take a randomly-projected state and map it to a position on a table that can record information about that state.

The gist of the data structure has already been implicitly described. Let us formally describe the whole set-up. It consists of:

- A hash table h that maps (randomly projected) states to a transition list
- A state encoder ϕ that does the initial projection
- A global transition FIFO queue q with limited capacity

The *append* operation given a new transition (s, a, s') is described in Algorithm 1. Note that by using a limited capacity queue, whenever the maximum replay buffer capacity is reached, the queue automatically handles the deletion of transitions that are too old.

Algorithm 1 Append a new transition to the USR replay buffer

```
function APPEND($s, a, s'$)
    $t \leftarrow (s, a, s')$
    $q \leftarrow [q, t]$                       ▷ Append transition to queue
    $i \leftarrow$ LENGTH($q$)                  ▷ Get transition id
    $e \leftarrow \phi(s)$                      ▷ Encode state
    $h(e) \leftarrow [h(e), i]$                 ▷ Add transition id to state transition list
```

The *sampling* procedure for the USR replay buffer is described in Algorithm 2. By sampling from the state list (the list of keys to the hash table), we get uniform state sampling without needing to keep a record of all transition sampling probabilities.

Algorithm 2 Sample a transition from USR replay buffer

```
function SAMPLE
    states $\leftarrow$ KEYS($h$)
    $s \leftarrow$ SAMPLE(states, $n = 1$)
    transitions $\leftarrow h(s)$
    $t \leftarrow$ SAMPLE(transitions, $n = 1$)
    return t
```

5.2 Uniform State-Action Replay

Uniform State-Action Replay (USAR) is a straightforward extension of USR. The main point is to cover the state-action space uniformly instead of the state space. This difference allows for a more fine-grained replay. Because most of the environments used in practice are deterministic or weakly stochastic, sampling from the state-action space is—almost—identical to sampling from the (state, action, next state) space. As a result, we get almost-perfect transition deduplication.

Papers that mention uniform coverage of state space as a better sampling procedure (Fu et al., 2019; Santos et al., 2021) implement this version of the algorithm. Similar works show that a more uniform sampling distribution provides better results in offline RL (Zhang et al., 2021) by having a similar method to USAR, plus other improvements to the optimization objective. The combination of our intuition with existing literature suggests USAR is a better alternative to USR. We will be implementing both and then will compare their empirical performances.

In terms of implementation details, there is no big change. There is only a minor detail in state encoding that makes Algorithms 1 and 2 work fine. The state encoding $e \leftarrow \phi(s)$ now becomes state-action encoding $e \leftarrow \phi(s, a)$. By abstracting this step of the process we can leave the algorithm unchanged.

5.3 Why not state, action and next state?

Given that we have proposed USR and USAR, one may think that implementing a new algorithm that replays every (state, action, next state) triple uniformly would be even better and lead to a more fine-grained replay. This is the same argument we used to support using USAR instead of USR. However, there is a key difference that makes this method (that we can call USASR) bad for Q-Learning.

In a triplet (s, a, s'):

- s is a state that the agent receives passively.
- a is an action that the agent chooses to take. So when going from s to a, only the policy π is involved.
- s', instead, is conditioned on the dynamics model of the environment $s' \sim p(s' \mid s, a)$.

Mixing the dynamics model with the sampling strategy can lead to large errors in learning, particularly so in stochastic environments. This is better understood with an example, see Example 5.1.

Example 5.1: USASR does not work

Suppose a state s_0 and an action a_0. The transition probabilities for the next state s' given s_0 and a_0 are:

$$p(s' \mid s_0, a_0) = \begin{cases} \frac{1}{10} & \text{if } s' = s_1 \\ \frac{9}{10} & \text{if } s' = s_2 \end{cases}$$

We also know that, in this situation:

$$\begin{aligned} Q_*(s_1, a) &= 1 \quad \forall a \in \mathcal{A} \\ Q_*(s_2, a) &= 0 \quad \forall a \in \mathcal{A} \end{aligned}$$

Sampling from a replay buffer using USR to calculate $Q_*(s_0, a_0)$, the result would be as follows (assuming all $r = 0$ and $\gamma = 1$).

$$\begin{aligned} Q_*(s_1, a_1) &= \mathbb{E}\left[\max_{a'} Q_*(s', a') \,\middle|\, s, a\right] \\ &\simeq \frac{1}{N} \sum_{i=0}^{N} \left(\max_{a'} Q_*(s'_i, a')\right) \\ &= \frac{1}{N}\left(\frac{N}{10} \max_{a'} Q_*(s_1, a') + \frac{9N}{10} \max_{a'} Q_*(s_2, a')\right) \quad = \frac{1}{10} \times 1 + \frac{9}{10} \times 0 \\ &= \frac{1}{10} \end{aligned}$$

It is easy to see that USAR would obtain the same result, because both s and a are fixed in the action-value function computation. However, if we replayed with the hypothetical method that uses the next state, samples (s_0, a_0, s_1) would be replayed as often as (s_0, a_0, s_2). The calculation of $Q_*(s_0, a_0)$ would then be:

$$\begin{aligned} Q_*(s_1, a_1) &\simeq \frac{1}{N}\left(\frac{N}{2} \max_{a'} Q_*(s_1, a') + \frac{N}{2} \max_{a'} Q_*(s_2, a')\right) \quad = \frac{1}{2} \times 1 + \frac{1}{2} \times 0 \\ &= \frac{1}{2} \end{aligned}$$

Which is wrong.

Recall that in deep Q-Learning, the goal is to learn $Q_\theta(s, a)$ so that the Bellman optimality equation [3.3] is satisfied. This is achieved by sampling mini-batches and learning Q_θ via stochastic gradient descent. If the sampling distribution is similar to the real one obtained by interacting with the environment, the computations done with the sampled batches eventually converge to the expectation in equation [3.3]. This can be done by either learning from online interactions or sampling uniformly from a replay buffer.

In the methods proposed so far, the expectation approximation was not biased because (s, a) are fixed in the equation and s' was sampled according to the distribution in the replay buffer. If we sample uniformly from the (s, a, s')-space, the approximation will no longer asymptotically match the expectation, because the dynamics model's probabilities have been artificially changed. The example shows this phenomenon clearly.

In absolutely non-stochastic environments this sampling strategy would still lead to the same results as USAR, because every time we see (s, a) they will be followed by s'. But this reason makes this method not worth applying, because the replay strategy would yield the same samples.

5.4 Comparison of empirical results

We have now presented the intuition and implementation details of both of our new methods. In Chapter 4 we have seen that USR induces a more uniform distribution over the state space, compared to commonly used methods such as UER or PER. USAR has a similar behavior since it is similar to USR in design. Details on how USAR shapes sampling distributions can be seen in Appendix A.

This section will show what are the empirical results obtained by the two proposed methods and compare them against some known baselines (UER, PER, and TER). The environments to benchmark these algorithms will be SimpleCrossing, LavaCrossing easy and LavaCrossing hard, from the MiniGrid suite (Chevalier-Boisvert, Willems, and Pal, 2018). They are good examples of sparse reward tasks with increasingly long horizons. Experiments are also run in the MinAtar suite (Young and Tian, 2019), a simplification of Atari games in the style of M. G. Bellemare et al., 2013. These are general-purpose games where more RL algorithms can be easily benchmarked, because rewards are not as sparse as in MiniGrid.

Figures 5.1 to 5.8 show the empirical results obtained by all algorithms in the mentioned environments. All lines shown are the averaged results over 3 random seeds. The intervals plotted alongside the lines represent the max-min values obtained across all 3 runs.

5.4.1 MiniGrid environments

A common trend is observed in the MiniGrid environments shown in the figures. As the environments get harder, UER and PER become unable to learn as fast as the other methods. It gets to a point where neither of them is able to learn in Figure 5.3. TER is able to learn the fastest in this setting.

We hypothesize that this happens because in these domains rewards are sparse. Factoring in that horizons are increasingly long as environments get harder, UER and PER are unable to select the right samples to learn from. Most of the samples that the agent will see will be from one of the overrepresented states, as seen in Figures 4.7a and 4.7b. In these domains, UER and PER do not know what to sample and obtain bad performance.

TER is almost tailor-made for these environments: they have clearly defined terminal states with most of the positive reward, and little to no alternative paths that get to the goal other

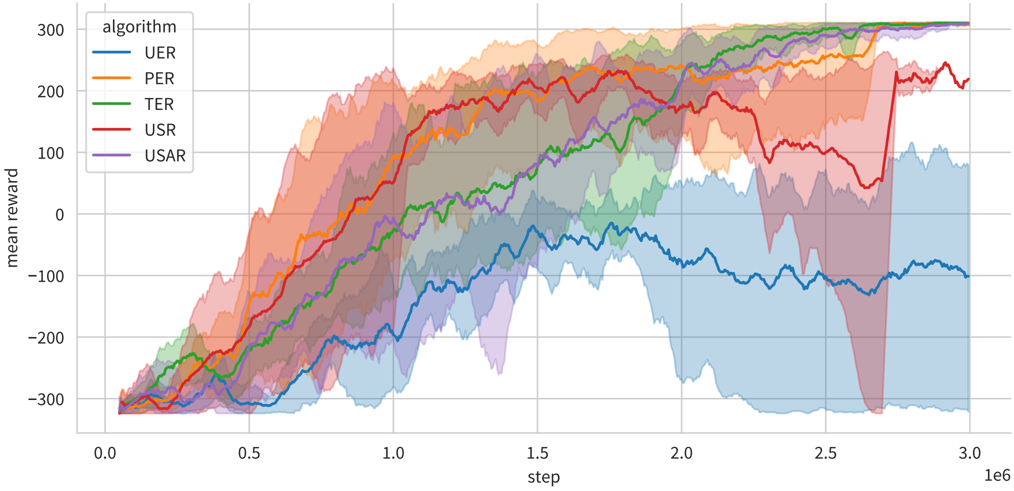


Figure 5.1 / Mean reward obtained by the experience replay algorithms in SimpleCrossing (easy).

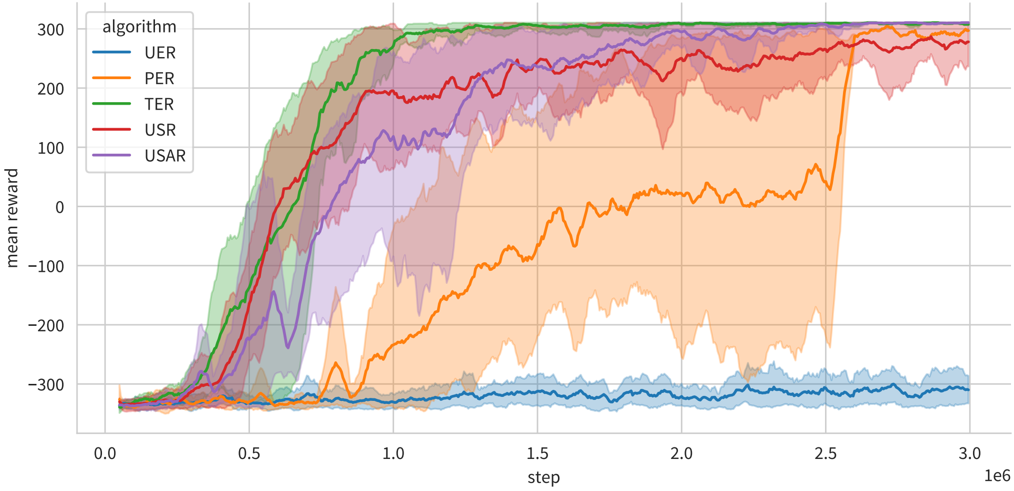


Figure 5.2 / Mean reward obtained in LavaCrossing (easy).

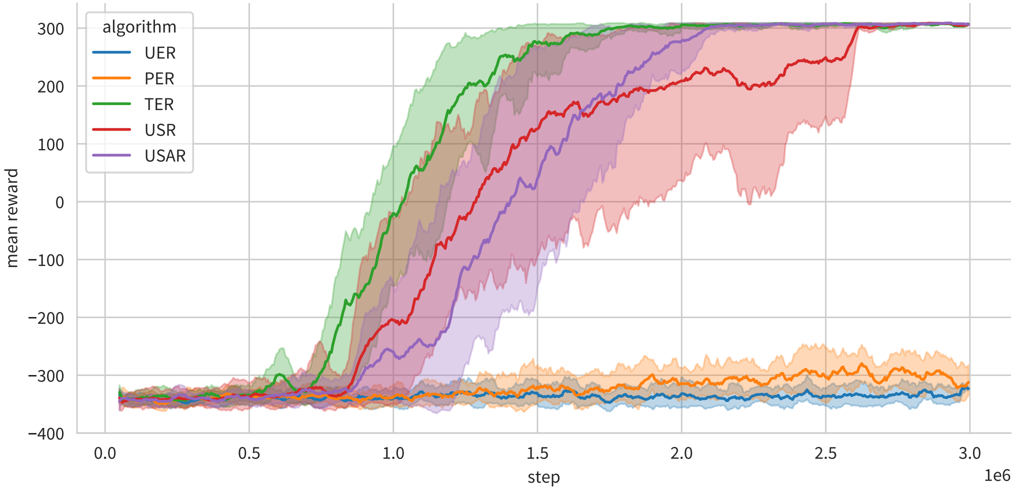


Figure 5.3 / Mean reward obtained in LavaCrossing (hard).

than the optimal one are allowed. TER replicates a sampling distribution similar to the one that the optimal policy would have here, so learning is much faster.

Both USR and USAR perform well in these environments. Remarkably, they are able to reach levels of performance close to TER without the graph and reward information that TER has. By more evenly covering the state space, the imbalance problem is solved and values are better propagated through all the environment. They show that in situations with sparse rewards and a large imbalance in the replay buffer, much better value propagation can be made across the whole state space just by sampling a more diverse set of states.

5.4.2 MinAtar environments

MinAtar environments essentially consist on simplifications of some Atari games. They are quite standard, and none of the games known for their hard exploration (e.g. Montezuma's revenge) are included. Because Atari is the main benchmark used in both Mnih et al., 2015 and Schaul et al., 2016, we can expect UER and PER to perform well in these environments.

Detailed examination of the sampling distribution plots in Appendix A reveals additional insights about the behavior of sampling algorithms in these environments. The main insight is that there is not a large imbalance in the replay balance, and we find the transition distribution for USR to be similar to the one UER has. Factoring everything in, we can say that these environments have much denser rewards, UER and PER are known to be effective here, the replay buffer is not as imbalanced as in the other benchmarks, and there are no long-horizon problems. As a result, these environments are not ideal to show better performance of USR and USAR against the baselines.

This intuition aligns with the mean reward plots in Figures 5.4 to 5.8. UER and PER perform well in all of them. Their mean rewards are closely followed by the ones of USR and USAR,

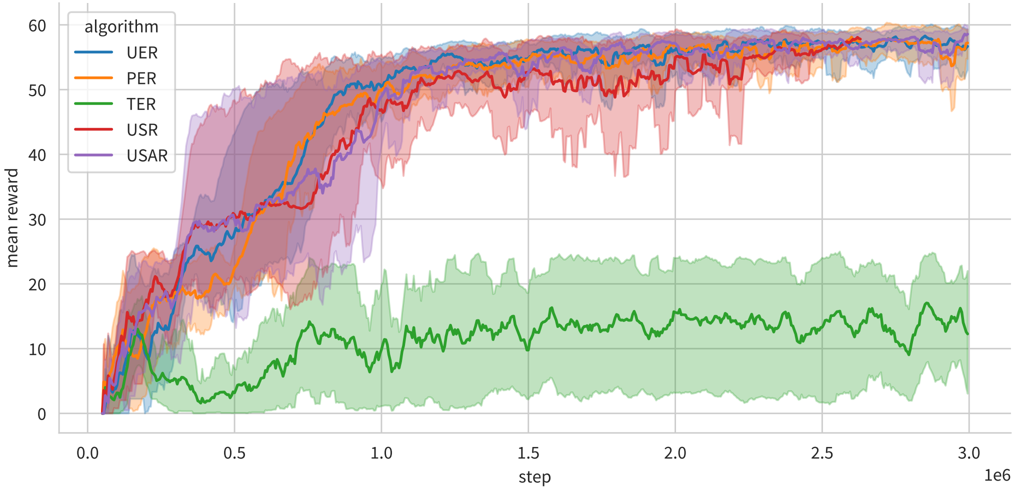


Figure 5.4 / Mean reward obtained in Freeway.

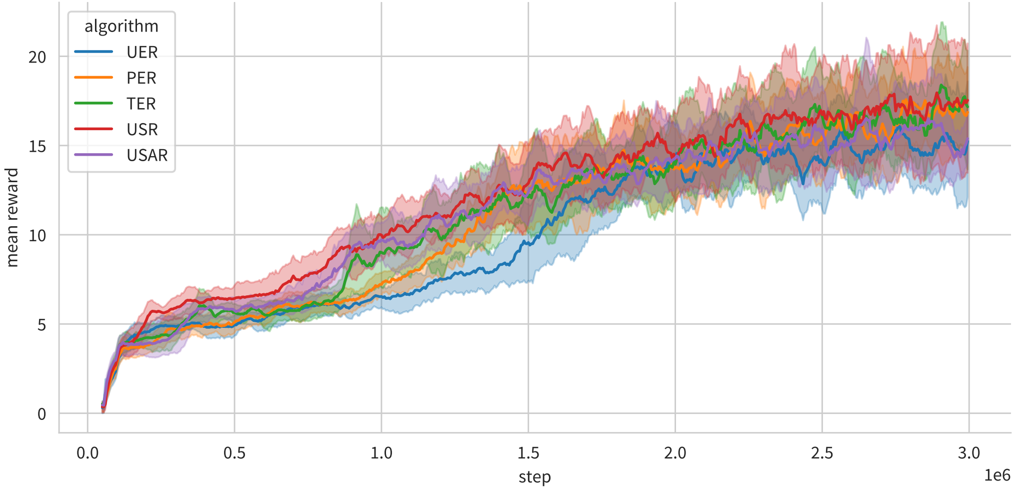


Figure 5.5 / Mean reward obtained in Breakout.

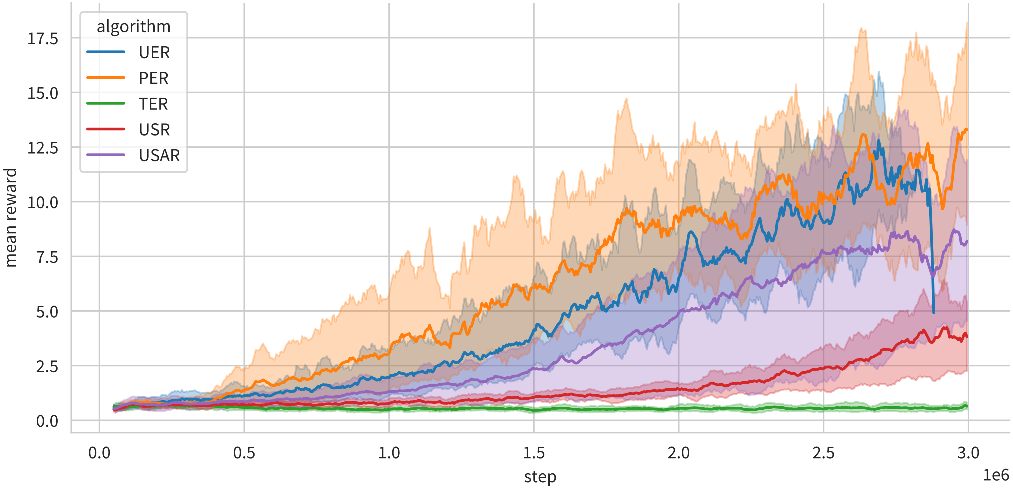


Figure 5.6 / Mean reward obtained in Asterix.

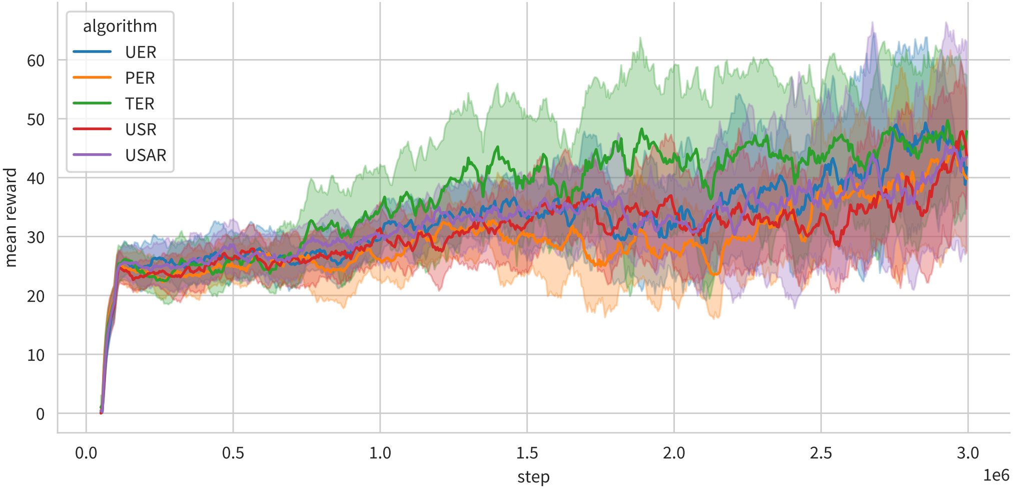


Figure 5.7 / Mean reward obtained by the experience replay algorithms in Space Invaders.

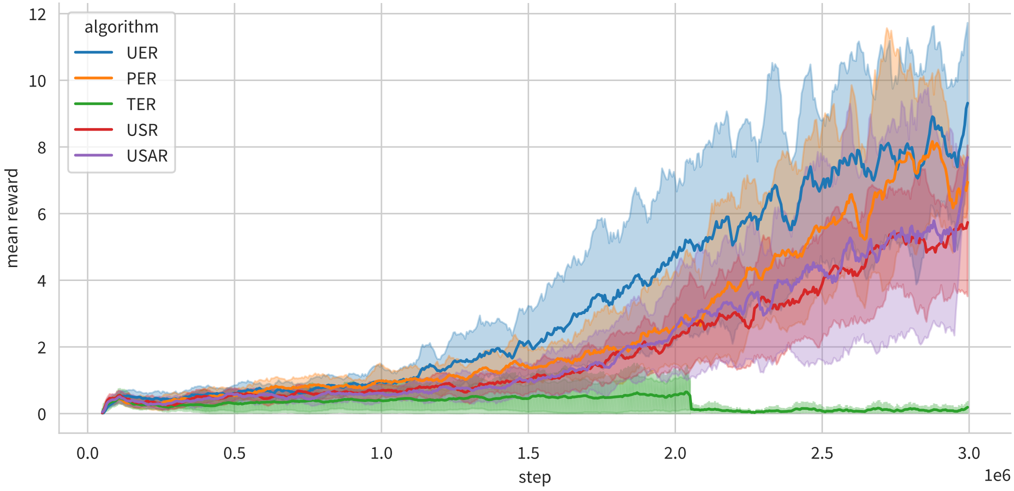


Figure 5.8 / Mean reward obtained by the experience replay algorithms in Seaquest.

and matched in Freeway and Breakout.

TER has bad performance in most of the MinAtar environments. It is often not able to learn during the training process. The reason for this is that what made it good in MiniGrid makes it bad here. The strong assumption of terminal states is not verified in most of these environments, because reward is distributed in different parts of the state space, not in a single goal that the agent has to reach at the end of the episode. In environments that lack these properties, TER does not perform well.

It is also worth noting that, although there is not a large imbalance in the dataset and rewards are not really sparse, USR and USAR perform (almost) as good as the best performing baselines. Seeing these plots it is fair to say that under unfavorable conditions, USR and USAR degrade to be very similar to UER. This is desirable because it means they can only improve on UER, as opposed to TER, which degrades to a bad sampling strategy that makes learning much more difficult.

Chapter 6

Discussion

This section is devoted to briefly discuss the main contributions this thesis provides, underline the relevant insights obtained and sketch the new research that can unfold after it.

6.1 Insights and contributions

We begin by showing that *sampling distribution is* an *important* part of experience replay algorithms. Among other reasons, sampling distributions provide a way to succinctly define how replay happens. As a result, we claim it is a useful way to analyze different experience replay methods using the same set of tools. Along these same lines, we derive how the transition sampling distribution of an experience replay algorithm in turn defines the state sampling distribution when applied to a replay buffer. Although it is clear that examining the distribution frequency of states does not provide the full picture of how a replay mechanism works, it allows to form an intuition on what may result in better performance.

Chapter 4 deals with defining these new tools, describing the ideas behind them, and using them to study several replay algorithms in different environments. Results show that *replay buffer imbalance is a problem*, especially in sparse reward settings. In environments where rewards are rare, values need to be propagated fast from a single point across all state space for the agent to learn. Plots and analysis of the most commonly used algorithms in the literature, UER and PER, show that they are not strong enough to shape the state sampling distribution and solve the imbalance problem. This phenomenon results in bad performance for these *weaker* algorithms.

In contrast, when studying TER we find that a sampling strategy that more boldly biases the state sampling distribution is better suited to mitigate the imbalance problem. TER is designed to thrive in environments where rewards are sparse and replay buffers mostly contain transitions that are bad for value propagation across the state space. This makes it see a larger number of states during learning, overcoming the imbalance in sampling and obtaining improved performance.

We propose *two new experience replay algorithms* in Chapter 5, uniform state replay (USR) and uniform state-action replay (USAR). They are designed to specifically mitigate the issues that imbalance causes while maintaining a simple sampling strategy. Their main goal is to sample uniformly at random from the state space, maximizing the state-entropy and the variety of samples that the agent sees during learning. The study of their sampling distributions in imbalanced replay buffers show that they provide a more even coverage of the state space, prioritizing which transitions to replay to avoid being influenced by the imbalance.

The empirical performance of the two newly proposed algorithms confirms our intuitions. In the SimpleCrossing and LavaCrossing environments of the MiniGrid suite, our algorithms learn slightly slower than TER. UER and PER, instead, are not able to learn at all when these environments get difficult. We manage to *overcome the imbalance* in the replay buffer and propagate the values almost as fast as TER, which is perfectly designed for these environments. When testing them in MinAtar games, performance is similar to the one obtained by UER and PER, showing that in environments where rewards are not sparse and imbalance is not a large issue, USR and USAR become similar to UER.

6.2 Future work

The work done here can be extended in several directions. We have proposed some ideas on how to solve imbalances and learn in sparse reward settings using simple and intuitive replay buffer sampling strategies. These same ideas can be applied in several new domains.

An interesting line of work can evolve from applying these ideas to environments with *continuous or high-dimensional state spaces*. In these domains it is improbable to hit the exact same state twice, and the implementation proposed for the algorithms is no longer valid. New ideas in terms of estimation of state space visitation density have to be applied, and an implementation in terms of importance sampling weights may be more feasible than one similar to the current design.

We have shown that PER replays the states with the larger learning opportunity (even though Bellman error is often not a good proxy for it). Additionally, we have shown that PER cannot overcome imbalance when it happens, and gets degraded to a slightly less biased version of UER. Given that we show how to overcome imbalance and assuming a weakly stochastic transition matrix, *state replay can be prioritized* according to the average TD error of a state's transitions. This may fuse the benefits of both PER and our research.

Another direction in which research can be conducted is in verifying the intuitions behind this work. Perhaps *uniform state space coverage is not the ideal strategy* to look for. Maybe there are some regions of the space that are not worth the agent's time. It can also be possible to infuse the sampling strategy with reward information to determine what a better sampling strategy could be for the particular case.

Bibliography

Abbeel, Pieter, Adam Coates, Morgan Quigley, and Andrew Ng, (2006). "An Application of Reinforcement Learning to Aerobatic Helicopter Flight." In: *Advances in Neural Information Processing Systems*. Vol. 19. MIT Press (cit. on p. 21).

Adele, Larry Pyeatt and Adele E. Howe, (1998). *Decision Tree Function Approximation in Reinforcement Learning*. In Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models (cit. on p. 21).

Bayes, Thomas and null Price, (Jan. 1763). "LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S." In: *Philosophical Transactions of the Royal Society of London* 53. Publisher: Royal Society, pp. 370–418 (cit. on p. 9).

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling, (June 2013). "The Arcade Learning Environment: An Evaluation Platform for General Agents." In: *Journal of Artificial Intelligence Research* 47, pp. 253–279 (cit. on pp. 28, 53).

Bellman, Richard, (1957a). "A Markovian Decision Process." In: *Indiana University Mathematics Journal* 6.4, pp. 679–684 (cit. on p. 12).

– (1957b). *Dynamic Programming*. Google-Books-ID: rZW4ugAACAAJ. Princeton University Press. 339 pp. (cit. on pp. 6, 16).

Bertsekas, Dimitri P., (Feb. 1, 2015). *Parallel and Distributed Computation: Numerical Methods*. 1st edition. Nashua, NH: Athena Scientific. 735 pp. (cit. on pp. 16, 17).

Blanton, Michael R. et al., (July 1, 2017). "Sloan Digital Sky Survey IV: Mapping the Milky Way, Nearby Galaxies, and the Distant Universe." In: *The Astronomical Journal* 154. ADS Bibcode: 2017AJ....154...28B, p. 28 (cit. on p. 7).

Brown, Tom et al., (2020). "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 1877–1901 (cit. on p. 11).

Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, (June 1, 2002). "SMOTE: Synthetic Minority Over-sampling Technique." In: *Journal of Artificial Intelligence Research* 16, pp. 321–357 (cit. on p. 7).

Chevalier-Boisvert, Maxime, Lucas Willems, and Suman Pal, (2018). *Minimalistic Gridworld Environment for OpenAI Gym*. Publication Title: GitHub repository (cit. on pp. 40, 48, 53).

"Crystallography" (Oct. 1, 1971). "Crystallography: Protein Data Bank." In: *Nature New Biology* 233.42. Number: 42 Publisher: Nature Publishing Group, pp. 223–223 (cit. on p. 7).

Dasgupta, Sanjoy and Anupam Gupta, (2003). "An elementary proof of a theorem of Johnson and Lindenstrauss." In: *Random Structures & Algorithms* 22.1. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.10073, pp. 60–65 (cit. on p. 40).

David Pfau [@pfau], (June 8, 2022). *@raphaelmilliere I don't think a single one of these models you cited is self-supervised.* Twitter. URL: https://twitter.com/pfau/status/1534737333127356418 (visited on 06/28/2022) (cit. on p. 10).

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, (2009). "Imagenet: A large-scale hierarchical image database." In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255 (cit. on p. 7).

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. NAACL-HLT 2019. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186 (cit. on p. 11).

Ego-Stengel, Valérie and Matthew A. Wilson, (2010). "Disruption of ripple-associated hippocampal activity during rest impairs spatial learning in the rat." In: *Hippocampus* 20.1. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/hipo.20707, pp. 1–10 (cit. on p. 3).

Emmons, Scott, Ajay Jain, Misha Laskin, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak, (2020). "Sparse Graphical Memory for Robust Planning." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 5251–5262 (cit. on p. 29).

Eysenbach, Ben, Russ R Salakhutdinov, and Sergey Levine, (2019). "Search on the Replay Buffer: Bridging Planning and Reinforcement Learning." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc. (cit. on p. 29).

Fedus, William, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney, (July 13, 2020). "Revisiting fundamentals of experience replay." In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, pp. 3061–3071 (cit. on p. 30).

Fienberg, Stephen E., (Mar. 2006). "When did Bayesian inference become "Bayesian"?" In: *Bayesian Analysis* 1.1. Publisher: International Society for Bayesian Analysis, pp. 1–40 (cit. on p. 9).

Foster, David J., (2017). "Replay Comes of Age." In: *Annual Review of Neuroscience* 40.1. _eprint: https://doi.org/10.1146/annurev-neuro-072116-031538, pp. 581–602 (cit. on p. 3).

Fu, Justin, Aviral Kumar, Matthew Soh, and Sergey Levine, (Feb. 26, 2019). *Diagnosing Bottlenecks in Deep Q-learning Algorithms*. arXiv:1902.10250. type: article. arXiv. arXiv: `1902.10250[cs,stat]` (cit. on pp. 30, 48, 51).

Fujimoto, Scott, Herke van Hoof, and David Meger, (July 16, 2019). "Addressing Function Approximation Error in Actor-Critic Methods." In: ICML (cit. on p. 26).

Fujimoto, Scott, David Meger, and Doina Precup, (2020). "An Equivalence between Loss Functions and Non-Uniform Sampling in Experience Replay." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 14219–14230 (cit. on pp. 6, 28, 32).

Fujimoto, Scott, David Meger, Doina Precup, Ofir Nachum, and Shixiang Shane Gu, (Nov. 21, 2021). "Why Should I Trust You, Bellman? Evaluating the Bellman Objective with Off-Policy Data." In: (cit. on p. 28).

Gu, Shixiang, Ethan Holly, Timothy Lillicrap, and Sergey Levine, (May 2017). "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates." In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 3389–3396 (cit. on pp. 1, 18).

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine, (July 3, 2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, pp. 1861–1870 (cit. on p. 27).

Hafner, Danijar, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson, (2019). "Learning Latent Dynamics for Planning from Pixels." In: *International Conference on Machine Learning*, pp. 2555–2565 (cit. on p. 18).

Hasselt, Hado van, (2012). "Reinforcement Learning in Continuous State and Action Spaces." In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Adaptation, Learning, and Optimization. Berlin, Heidelberg: Springer, pp. 207–251 (cit. on p. 12).

Hasselt, Hado van, Sephora Madjiheurem, Matteo Hessel, David Silver, André Barreto, and Diana Borsa, (May 18, 2021). "Expected Eligibility Traces." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.11. Number: 11, pp. 9997–10005 (cit. on p. 6).

Hochreiter, Sepp and Jürgen Schmidhuber, (Dec. 3, 1996). "LSTM can solve hard long time lag problems." In: *Proceedings of the 9th International Conference on Neural Information Processing Systems*. NIPS'96. Cambridge, MA, USA: MIT Press, pp. 473–479 (cit. on p. 21).

Hong, Zhang-Wei, Tao Chen, Yen-Chen Lin, Joni Pajarinen, and Pulkit Agrawal, (Sept. 29, 2021). "Topological Experience Replay." In: International Conference on Learning Representations (cit. on pp. 6, 29, 33, 40, 49).

Hwangbo, Jemin, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter, (Jan. 16, 2019). "Learning agile and dynamic motor skills for legged robots." In: *Science Robotics* 4.26. Publisher: American Association for the Advancement of Science, eaau5872 (cit. on p. 18).

Ilya Sutskever [@ilyasut], (Feb. 9, 2022). *it may be that today's large neural networks are slightly conscious*. Twitter. URL: `https://twitter.com/ilyasut/status/1491554478243258368` (visited on 06/08/2022) (cit. on p. 20).

Jiang, Zhengyao, Tianjun Zhang, Robert Kirk, Tim Rocktäschel, and Edward Grefenstette, (May 31, 2022). *Graph Backup: Data Efficient Backup Exploiting Markovian Transitions*. arXiv:2205.15824. type: article. arXiv. arXiv: `2205.15824[cs]` (cit. on pp. 6, 29).

Kaiser, Łukasz et al., (Mar. 11, 2020). "Model Based Reinforcement Learning for Atari." In: International Conference on Learning Representations (cit. on p. 18).

Kostrikov, Ilya, Ashvin Nair, and Sergey Levine, (Sept. 29, 2021). "Offline Reinforcement Learning with Implicit Q-Learning." In: International Conference on Learning Representations (cit. on p. 19).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton, (2012). "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc. (cit. on p. 7).

Kubat, Miroslav and Stan Matwin, (1997). "Addressing the Curse of Imbalanced Training Sets: One-Sided Selection." In: *Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997*. Ed. by Douglas H. Fisher. Morgan Kaufmann, pp. 179–186 (cit. on p. 7).

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine, (2020). "Conservative Q-Learning for Offline Reinforcement Learning." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 1179–1191 (cit. on p. 19).

Kurth-Nelson, Zeb, Marcos Economides, Raymond J. Dolan, and Peter Dayan, (July 6, 2016). "Fast Sequences of Non-spatial State Representations in Humans." In: *Neuron* 91.1, pp. 194–204 (cit. on p. 3).

Kwakernaak, Huibert and Raphel Sivan, (Oct. 1, 1972). *Linear Optimal Control Systems*. 1st edition. New York: Wiley-Interscience. 608 pp. (cit. on p. 21).

Laplace, Pierre-Simon, (1774). "Mémoire sur la probabilité des causes par les évènements." In: *Œuvres complètes* 8 (cit. on p. 10).

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner, (Nov. 1998). "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11. Conference Name: Proceedings of the IEEE, pp. 2278–2324 (cit. on p. 21).

Lee, Su Young, Choi Sungik, and Sae-Young Chung, (2019). "Sample-Efficient Deep Reinforcement Learning via Episodic Backward Update." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc. (cit. on p. 29).

Legg, Shane and Marcus Hutter, (June 7, 2007). "A Collection of Definitions of Intelligence." In: *Proceedings of the 2007 conference on Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006*. NLD: IOS Press, pp. 17–24 (cit. on p. 9).

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu, (Nov. 1, 2020). *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems*. arXiv:2005.01643. type: article. arXiv. arXiv: `2005.01643[cs,stat]` (cit. on p. 19).

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra, (Jan. 17, 2021). "Continuous control with deep reinforcement learning." In: ICLR (Poster) (cit. on p. 26).

Lin, Long-Ji, (May 1, 1992). "Self-improving reactive agents based on reinforcement learning, planning and teaching." In: *Machine Learning* 8.3, pp. 293–321 (cit. on pp. 27, 31, 40).

Margolis, Gabriel, Ge Yang, Kartik Paigwar, Tao Chen, and Pulkit Agrawal, (2022). "Rapid Locomotion via Reinforcement Learning." In: *Robotics: Science and Systems* (cit. on pp. 1, 18).

Miller, Renée J., (2014). "Big Data Curation." In: *20th International Conference on Management of Data, COMAD 2014, Hyderabad, India, December 17-19, 2014*. Ed. by Srikanta Bedathur, Divesh Srivastava, and Satyanarayana R. Valluri. Computer Society of India, p. 4 (cit. on p. 7).

Mitchell, Tom M, (1997). *Machine Learning*. OCLC: 36417892. McGraw-Hill (cit. on p. 9).

Mnih, Volodymyr et al., (Feb. 2015). "Human-level control through deep reinforcement learning." In: *Nature* 518.7540. Number: 7540 Publisher: Nature Publishing Group, pp. 529–533 (cit. on pp. 5, 6, 18, 25, 27, 28, 31, 40, 55).

Nagabandi, Anusha, Kurt Konolige, Sergey Levine, and Vikash Kumar, (May 12, 2020). "Deep Dynamics Models for Learning Dexterous Manipulation." In: *Proceedings of the Conference on Robot Learning*. Conference on Robot Learning. ISSN: 2640-3498. PMLR, pp. 1101–1112 (cit. on p. 18).

Ormoneit, Dirk and Śaunak Sen, (Nov. 1, 2002). “Kernel-Based Reinforcement Learning.” In: *Machine Learning* 49.2, pp. 161–178 (cit. on p. 21).

Pazzani, Michael J., Christopher J. Merz, Patrick M. Murphy, Kamal M. Ali, Timothy Hume, and Clifford Brunk, (July 10, 1994). “Reducing misclassification costs.” In: *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*. ICML’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 217–225 (cit. on p. 7).

Precup, Doina, Richard S. Sutton, and Satinder P. Singh, (June 29, 2000). “Eligibility Traces for Off-Policy Policy Evaluation.” In: *Proceedings of the Seventeenth International Conference on Machine Learning*. ICML ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 759–766 (cit. on p. 6).

Riedmiller, Martin, (2005). “Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method.” In: *Machine Learning: ECML 2005*. Ed. by João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 317–328 (cit. on p. 22).

Riedmiller, Martin, Thomas Gabel, Roland Hafner, and Sascha Lange, (July 1, 2009). “Reinforcement learning for robot soccer.” In: *Autonomous Robots* 27.1, pp. 55–73 (cit. on p. 22).

Rosenblatt, F., (1958). “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6. Place: US Publisher: American Psychological Association, pp. 386–408 (cit. on p. 20).

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams, (Oct. 1986). “Learning representations by back-propagating errors.” In: *Nature* 323.6088. Number: 6088 Publisher: Nature Publishing Group, pp. 533–536 (cit. on p. 21).

Rummery, Gavin Adrian and M. Niranjan, (1994). “On-line Q-learning using Connectionist Systems.” In: *Department of Engineering, University of Cambridge, Cambridge.* (cit. on p. 6).

Samuel, Arthur L., (1959). “Some studies in machine learning using the game of Checkers.” In: *Ibm Journal of Research and Development*, pp. 71–105 (cit. on p. 9).

Santos, Pedro P., Francisco S. Melo, Alberto Sardinha, and Diogo S. Carvalho, (Nov. 23, 2021). *Understanding the Impact of Data Distribution on Q-learning with Function Approximation*. Number: arXiv:2111.11758. arXiv: `2111.11758[cs]` (cit. on pp. 48, 49, 51).

Savinov, Nikolay, Alexey Dosovitskiy, and Vladlen Koltun, (Feb. 27, 2018). “Semi-parametric topological memory for navigation.” In: International Conference on Learning Representations (cit. on p. 29).

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver, (2016). “Prioritized Experience Replay.” In: *International Conference on Learning Representations*. Puerto Rico (cit. on pp. 5, 6, 28, 31, 40, 43, 55).

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz, (June 1, 2015). “Trust Region Policy Optimization.” In: *Proceedings of the 32nd International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 1938-7228. PMLR, pp. 1889–1897 (cit. on p. 25).

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, (Aug. 28, 2017). *Proximal Policy Optimization Algorithms*. arXiv:1707.06347. type: article. arXiv. arXiv: `1707.06347[cs]` (cit. on p. 25).

Silver, David et al., (Jan. 2016). “Mastering the game of Go with deep neural networks and tree search.” In: *Nature* 529.7587. Number: 7587 Publisher: Nature Publishing Group, pp. 484–489 (cit. on p. 1).

Skaggs, William E. and Bruce L. McNaughton, (Mar. 29, 1996). “Replay of Neuronal Firing Sequences in Rat Hippocampus During Sleep Following Spatial Experience.” In: *Science* 271.5257. Publisher: American Association for the Advancement of Science, pp. 1870–1873 (cit. on p. 3).

Stigler, Stephen M., (1986). *The history of statistics : the measurement of uncertainty before 1900.* In collab. with Internet Archive. Cambridge, Mass. : Belknap Press of Harvard University Press. 442 pp. (cit. on p. 10).

Sutton, Richard S, David McAllester, Satinder Singh, and Yishay Mansour, (1999). “Policy Gradient Methods for Reinforcement Learning with Function Approximation.” In: *Advances in Neural Information Processing Systems*. Vol. 12. MIT Press (cit. on p. 24).

Sutton, Richard S., (Aug. 1, 1988). “Learning to predict by the methods of temporal differences.” In: *Machine Learning* 3.1, pp. 9–44 (cit. on pp. 6, 21).

Sutton, Richard S. and Andrew G. Barto, (2018). *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book. 552 pp. (cit. on pp. 6, 11, 12, 16, 17, 23, 24).

Tesauro, Gerald, (Mar. 1, 1995). “Temporal difference learning and TD-Gammon.” In: *Communications of the ACM* 38.3, pp. 58–68 (cit. on p. 22).

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin, (2017). “Attention is All you Need.” In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc. (cit. on p. 21).

Vinyals, Oriol et al., (Nov. 2019). “Grandmaster level in StarCraft II using multi-agent reinforcement learning.” In: *Nature* 575.7782. Number: 7782 Publisher: Nature Publishing Group, pp. 350–354 (cit. on p. 18).

Watkins, Christopher J. C. H. and Peter Dayan, (1992). “Q-learning.” In: *Machine Learning*, pp. 279–292 (cit. on p. 22).

Weng, Lilian, (2018). “Policy Gradient Algorithms.” In: *lilianweng.github.io* (cit. on p. 24).

Young, Kenny and Tian Tian, (June 6, 2019). *MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments*. Number: arXiv:1903.03176. arXiv: `1903.03176[cs]` (cit. on pp. 40, 48, 53).

Zhang, Hongchang, Jianzhun Shao, Yuhang Jiang, Shuncheng He, and Xiangyang Ji, (Feb. 26, 2021). *Reducing Conservativeness Oriented Offline Reinforcement Learning*. Number: arXiv:2103.00098. arXiv: `2103.00098[cs]` (cit. on p. 51).

Further Reading

The following references were used in this work but not cited in the text body; they are provided here as-is.

Daley, Brett, Cameron Hickert, and Christopher Amato, (Feb. 22, 2021). *Stratified Experience Replay: Correcting Multiplicity Bias in Off-Policy Reinforcement Learning*. Number: arXiv:2102.11319. arXiv: `2102.11319[cs]`.

Dehmamy, Nima, Robin Walters, Yanchen Liu, Dashun Wang, and Rose Yu, (May 21, 2021). "Automatic Symmetry Discovery with Lie Algebra Convolutional Network." In: Advances in Neural Information Processing Systems.

Florensa, Carlos, David Held, Xinyang Geng, and Pieter Abbeel, (July 3, 2018). "Automatic Goal Generation for Reinforcement Learning Agents." In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, pp. 1515–1528.

Kakade, S. and J. Langford, (2002). "Approximately Optimal Approximate Reinforcement Learning." In: *undefined*.

Kumar, Aviral, Abhishek Gupta, and Sergey Levine, (2020). "DisCor: Corrective Feedback in Reinforcement Learning via Distribution Correction." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 18560–18572.

Pinto, Lerrel, James Davidson, Rahul Sukthankar, and Abhinav Gupta, (Mar. 7, 2017). "Robust Adversarial Reinforcement Learning." In: *arXiv:1703.02702 [cs]*. arXiv: `1703.02702`.

Sukhbaatar, Sainbayar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus, (Apr. 27, 2018). "Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play." In: *arXiv:1703.05407 [cs]*. arXiv: `1703.05407`.

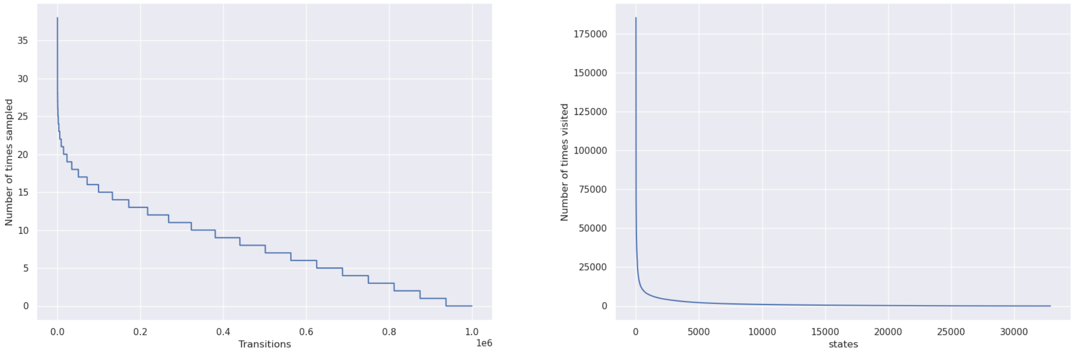
Touati, Ahmed, Adrien Ali Taiga, and Marc G. Bellemare, (Mar. 9, 2020). "Zooming for Efficient Model-Free Reinforcement Learning in Metric Spaces." In: *arXiv:2003.04069 [cs, stat]*. arXiv: `2003.04069`.

Veličković, Petar, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell, (May 31, 2022). *The CLRS Algorithmic Reasoning Benchmark*. arXiv:2205.15659. type: article. arXiv. arXiv: `2205.15659[cs, stat]`.

Appendix A

Sampling distribution plots for all benchmarks

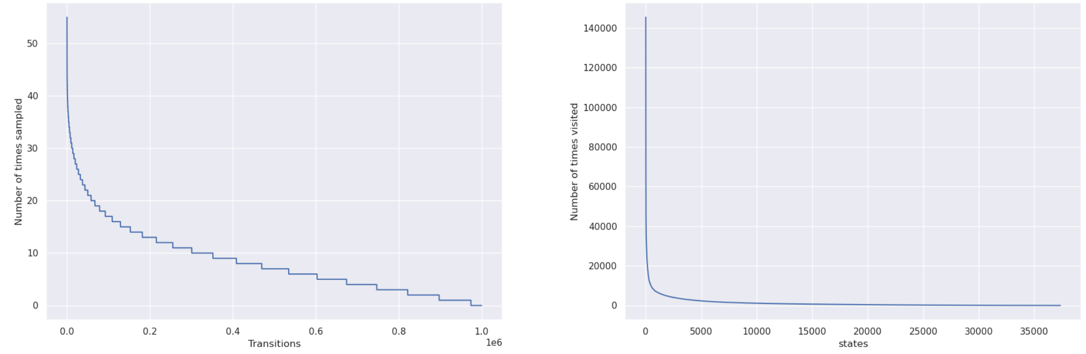
A.1 Plots for Minigrid LavaCrossing (hard)



a / Transition sampling distribution.

b / State sampling distribution.

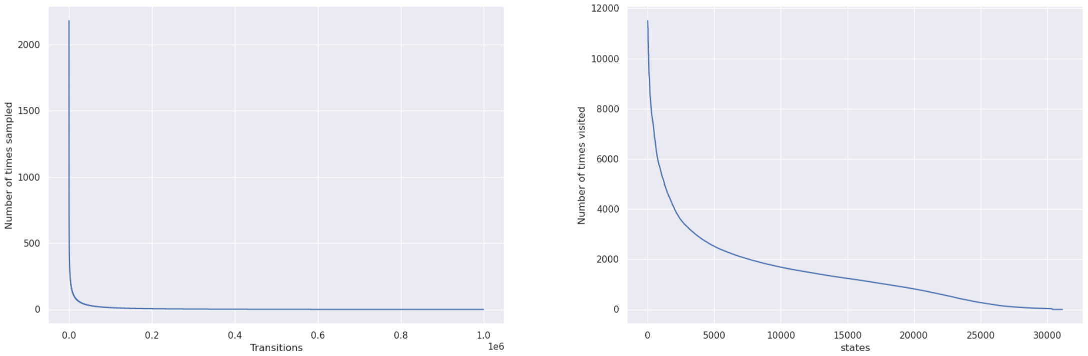
Figure A.1a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

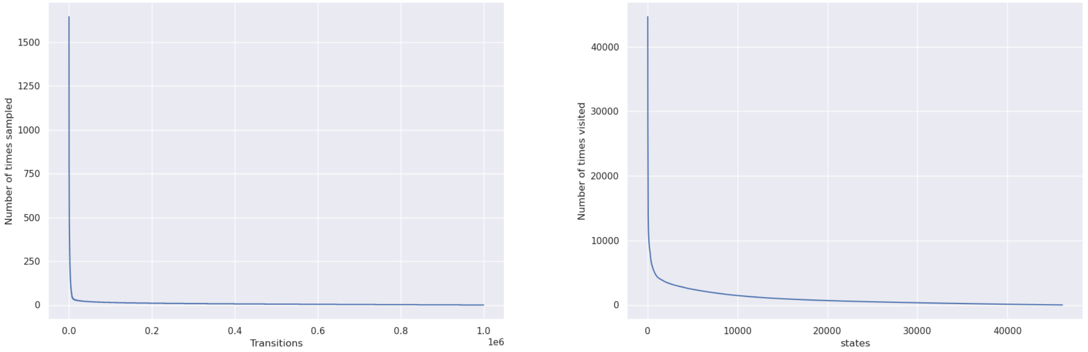
Figure A.1b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

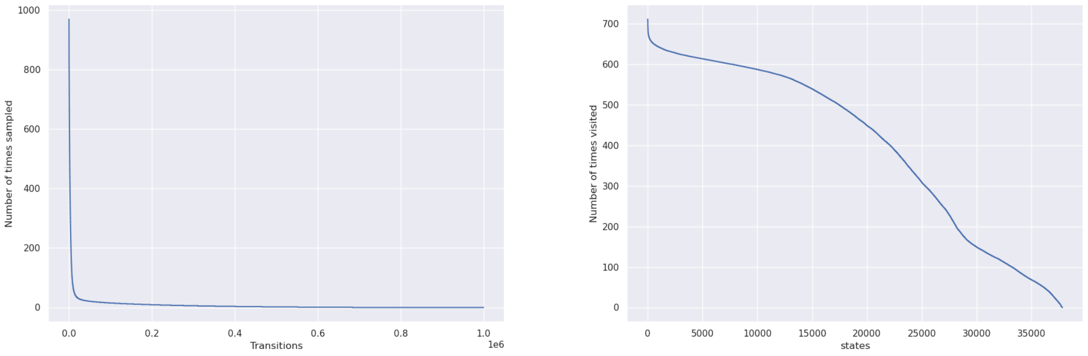
Figure A.1c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

Figure A.1d / Sampling frequencies for TER with batch mixing.

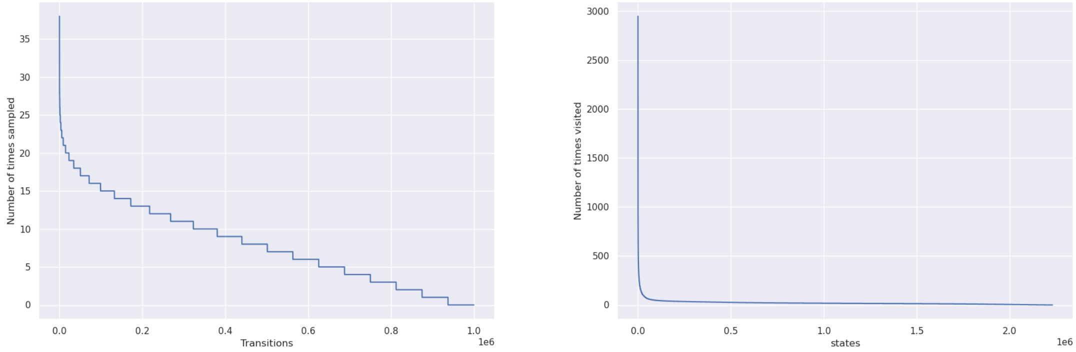


i / Transition sampling distribution.

j / State sampling distribution.

Figure A.1e / Sampling frequencies for USR.

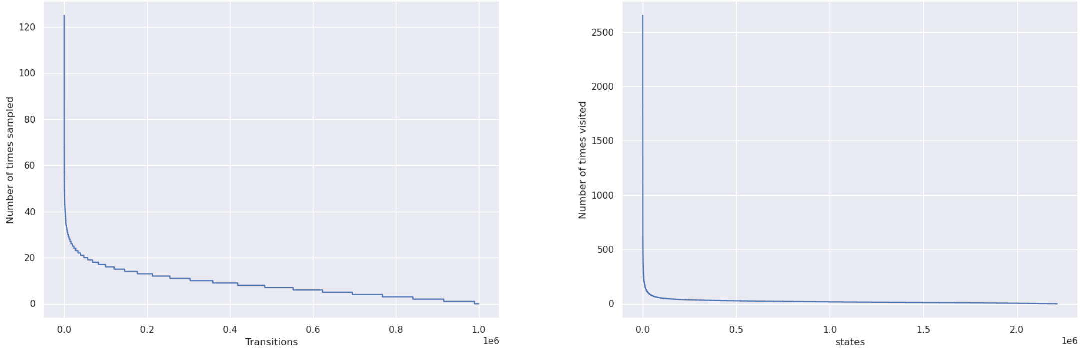
A.2 Plots for MinAtar Freeway



a / Transition sampling distribution.

b / State sampling distribution.

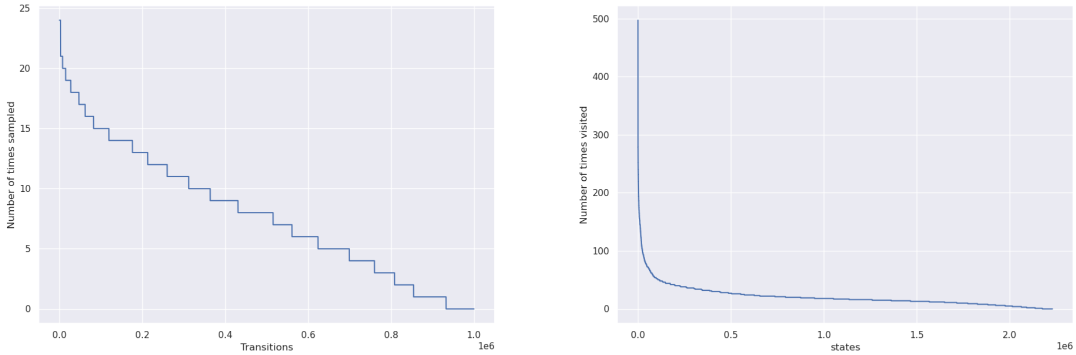
Figure A.2a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

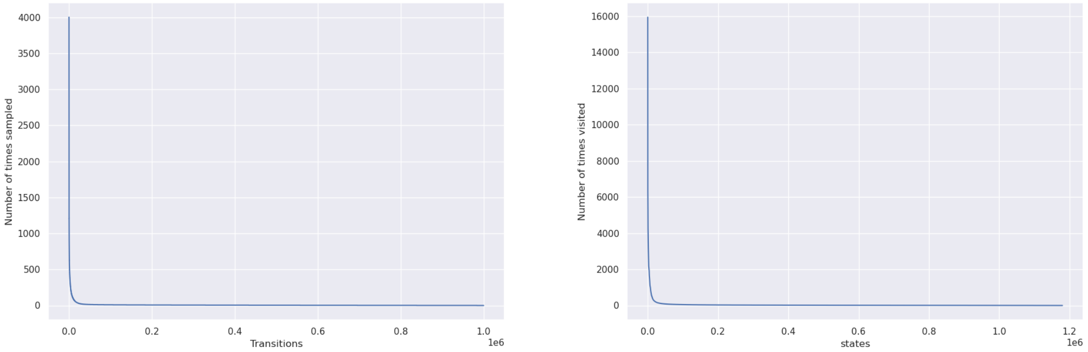
Figure A.2b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

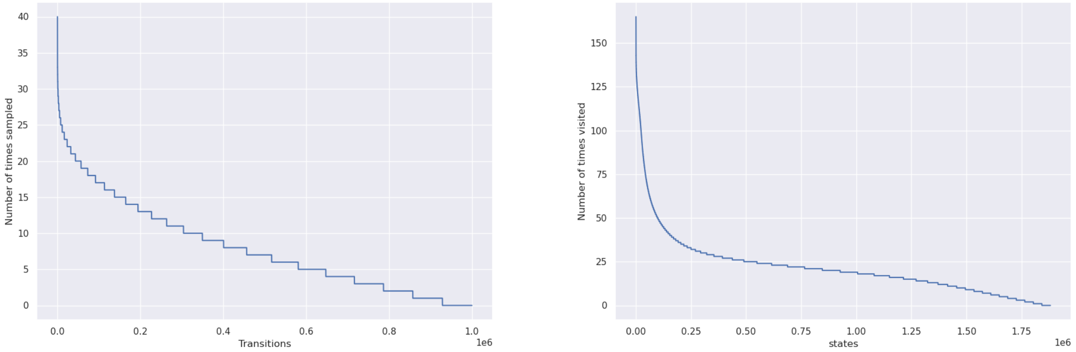
Figure A.2c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

Figure A.2d / Sampling frequencies for TER with batch mixing.

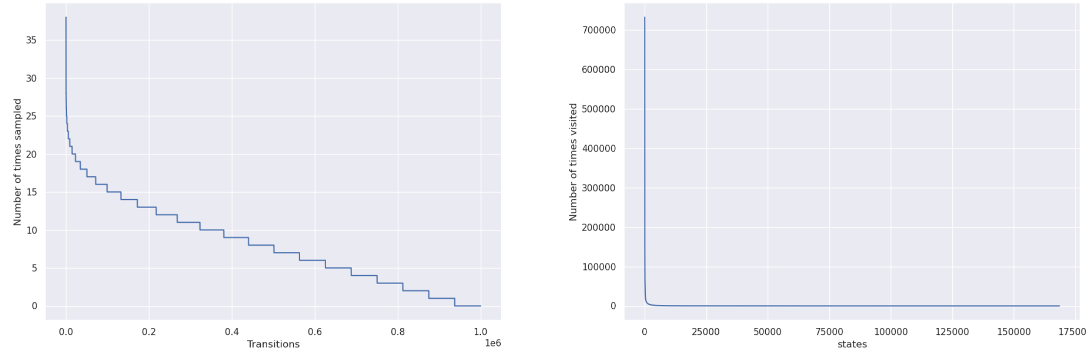


i / Transition sampling distribution.

j / State sampling distribution.

Figure A.2e / Sampling frequencies for USR.

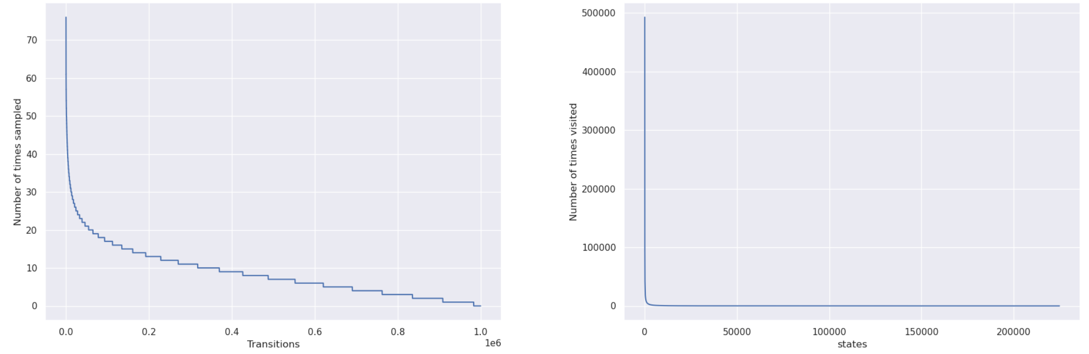
A.3 Plots for MinAtar Breakout



a / Transition sampling distribution.

b / State sampling distribution.

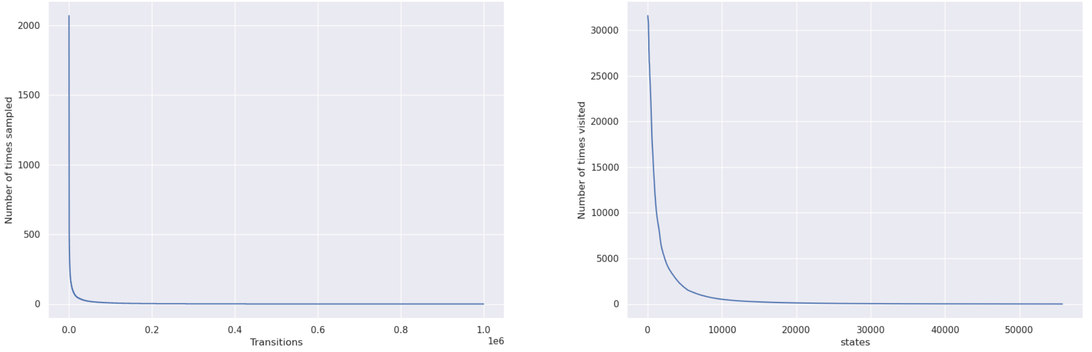
Figure A.3a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

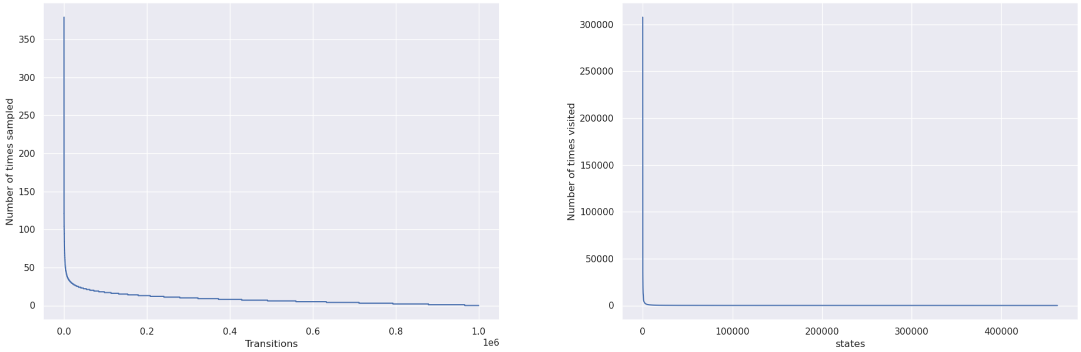
Figure A.3b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

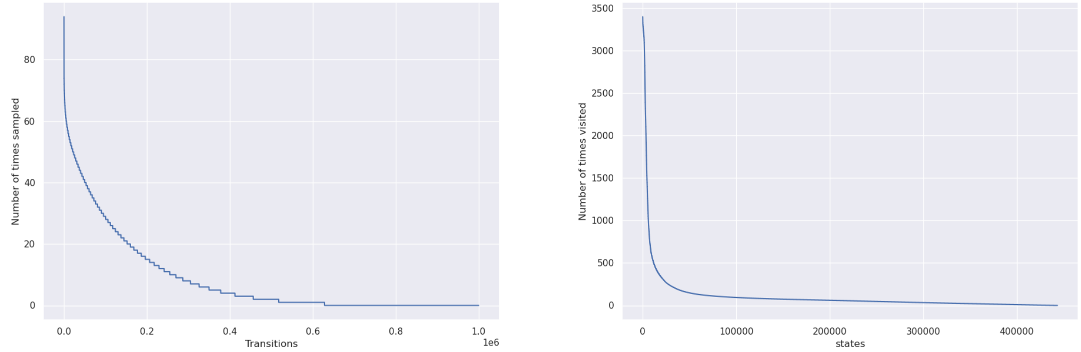
Figure A.3c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

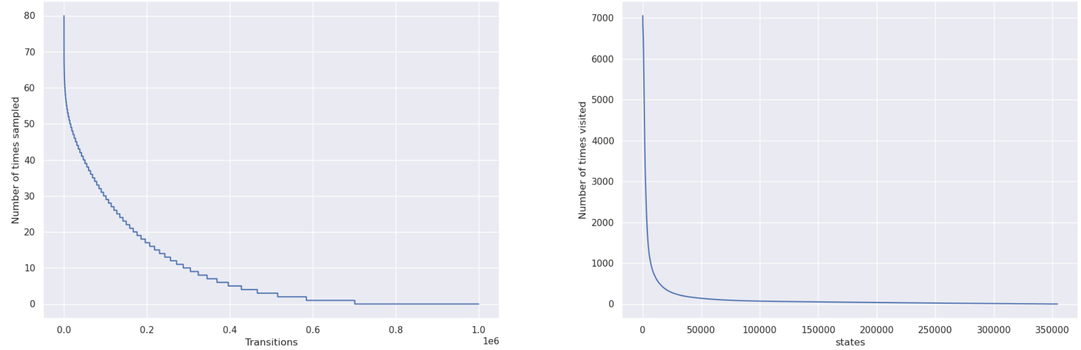
Figure A.3d / Sampling frequencies for TER with batch mixing.



i / Transition sampling distribution.

j / State sampling distribution.

Figure A.3e / Sampling frequencies for USR.

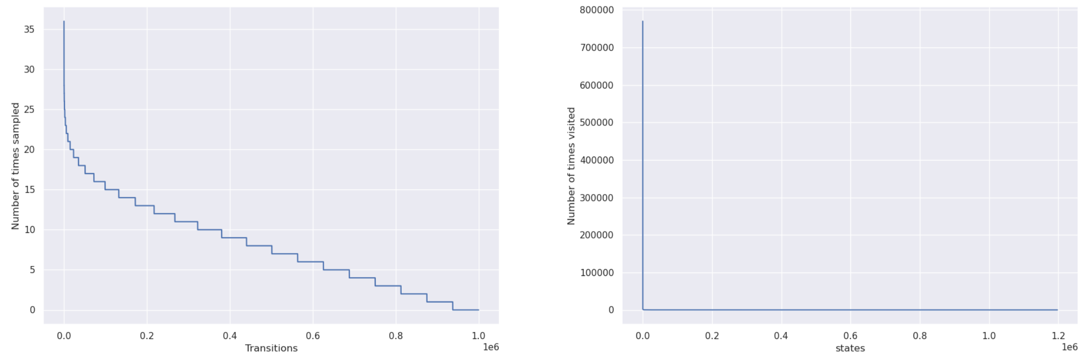


k / Transition sampling distribution.

l / State sampling distribution.

Figure A.3f / Sampling frequencies for USAR.

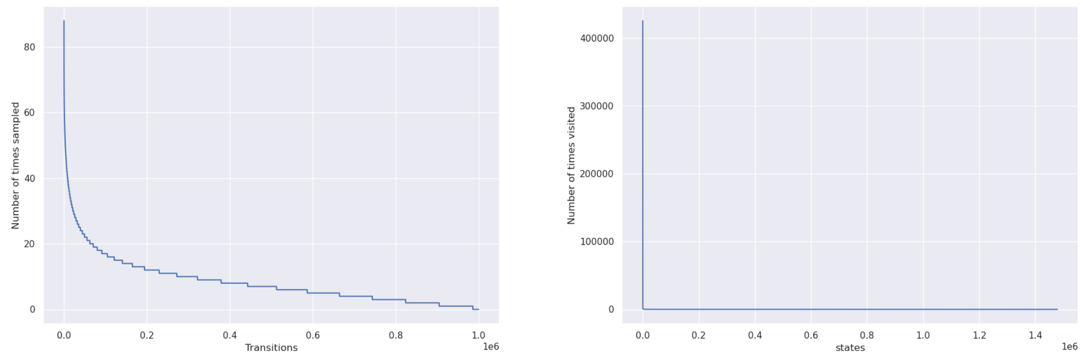
A.4 Plots for MinAtar Asterix



a / Transition sampling distribution.

b / State sampling distribution.

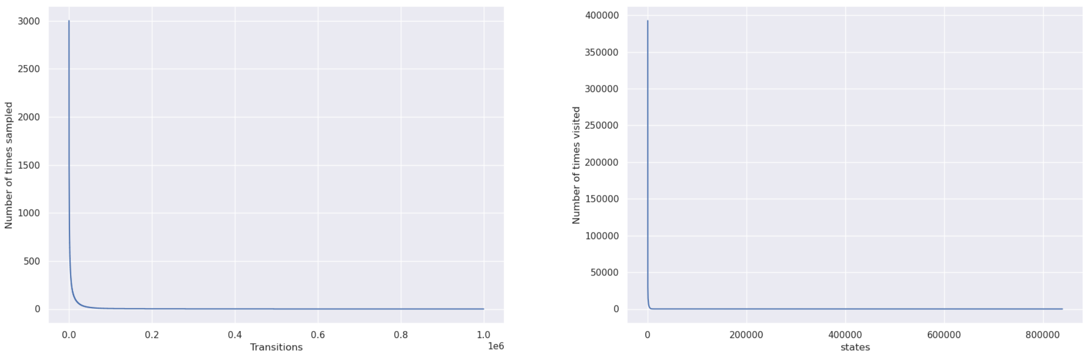
Figure A.4a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

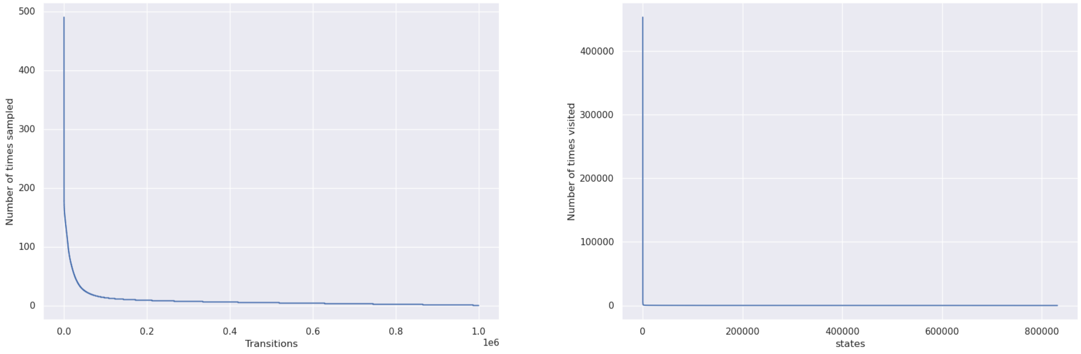
Figure A.4b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

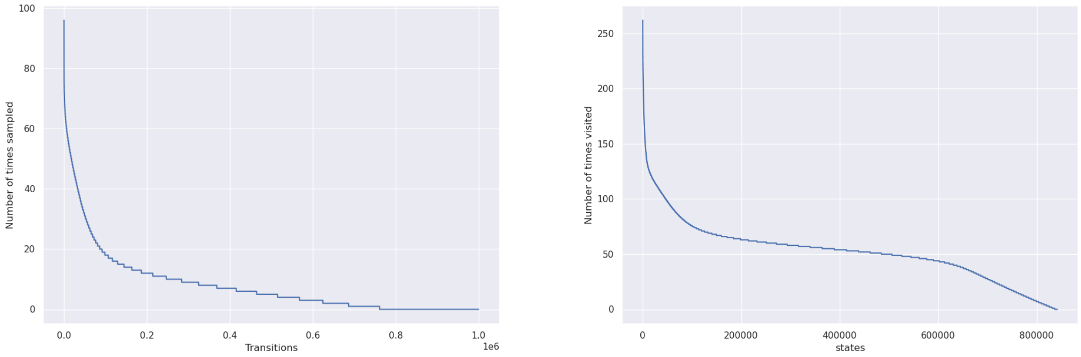
Figure A.4c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

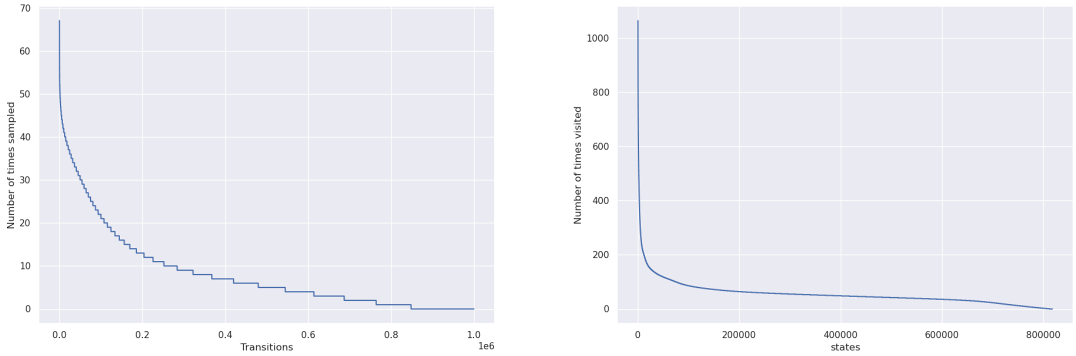
Figure A.4d / Sampling frequencies for TER with batch mixing.



i / Transition sampling distribution.

j / State sampling distribution.

Figure A.4e / Sampling frequencies for USR.

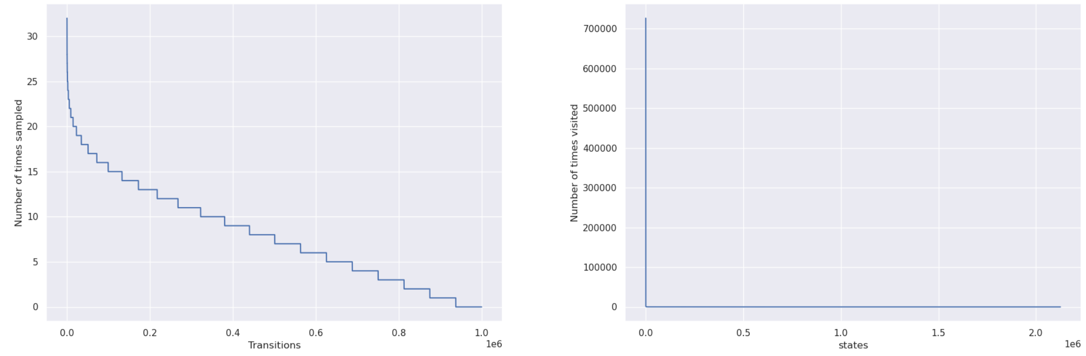


k / Transition sampling distribution.

l / State sampling distribution.

Figure A.4f / Sampling frequencies for USAR.

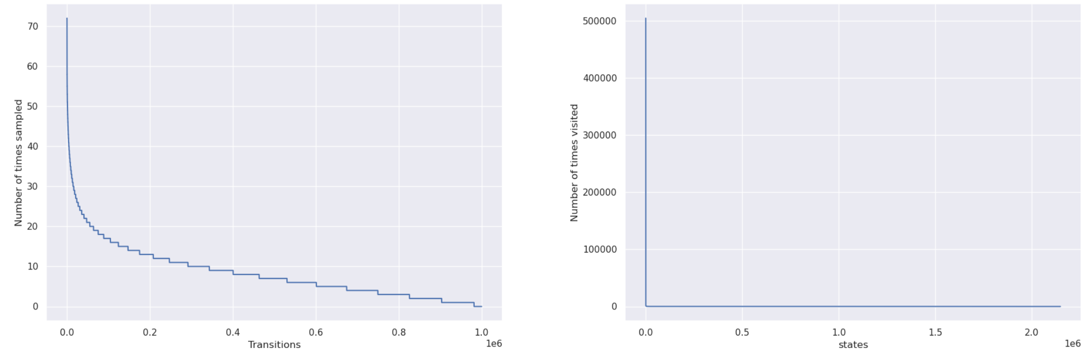
A.5 Plots for MinAtar Space Invaders



a / Transition sampling distribution.

b / State sampling distribution.

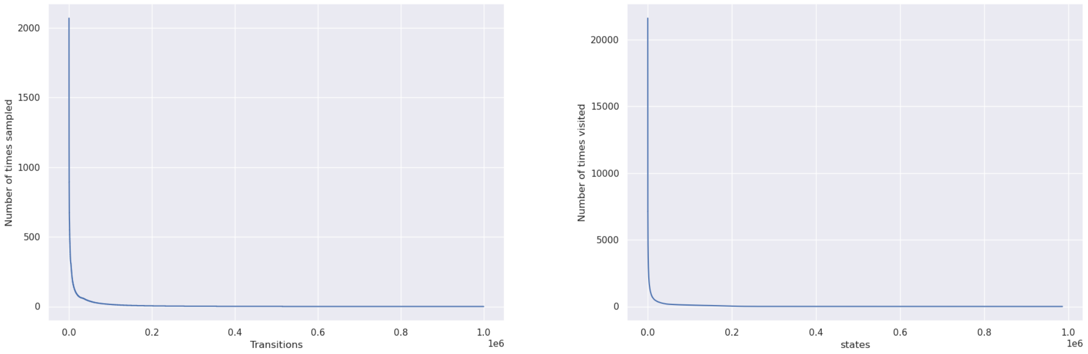
Figure A.5a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

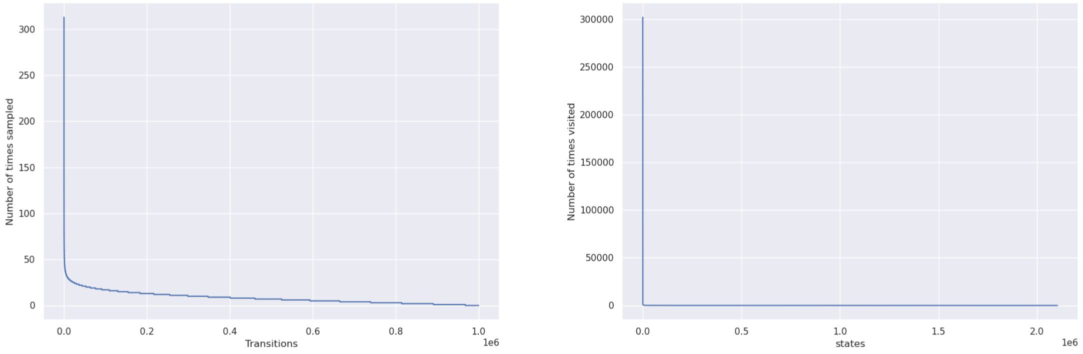
Figure A.5b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

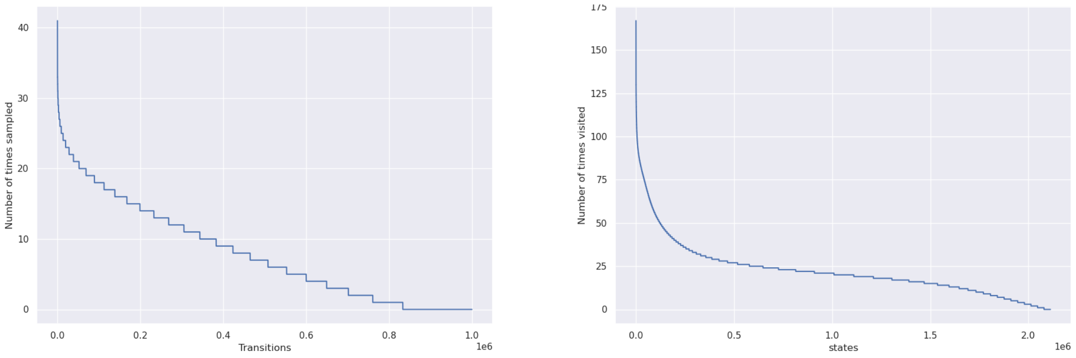
Figure A.5c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

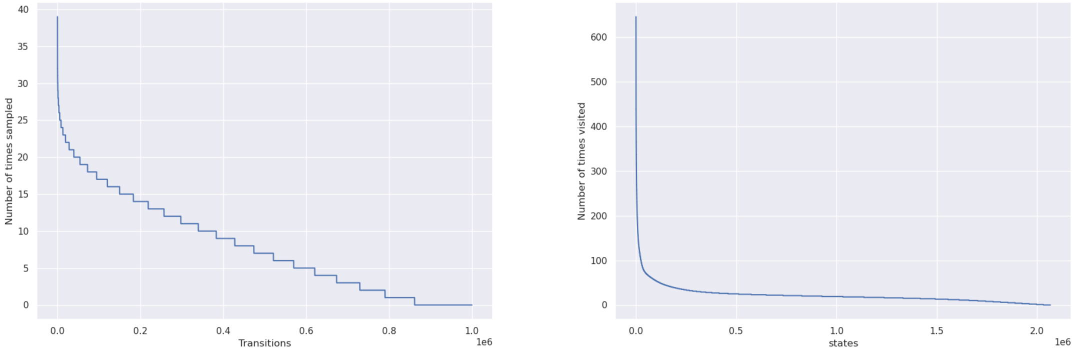
Figure A.5d / Sampling frequencies for TER with batch mixing.



i / Transition sampling distribution.

j / State sampling distribution.

Figure A.5e / Sampling frequencies for USR.

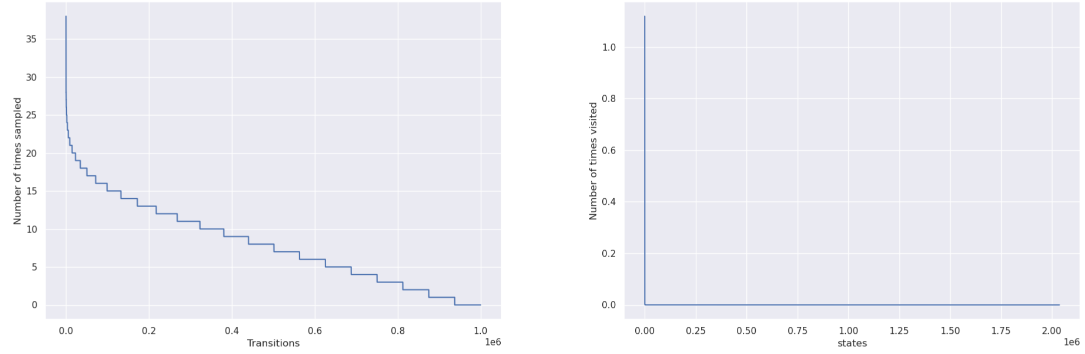


k / Transition sampling distribution.

l / State sampling distribution.

Figure A.5f / Sampling frequencies for USAR.

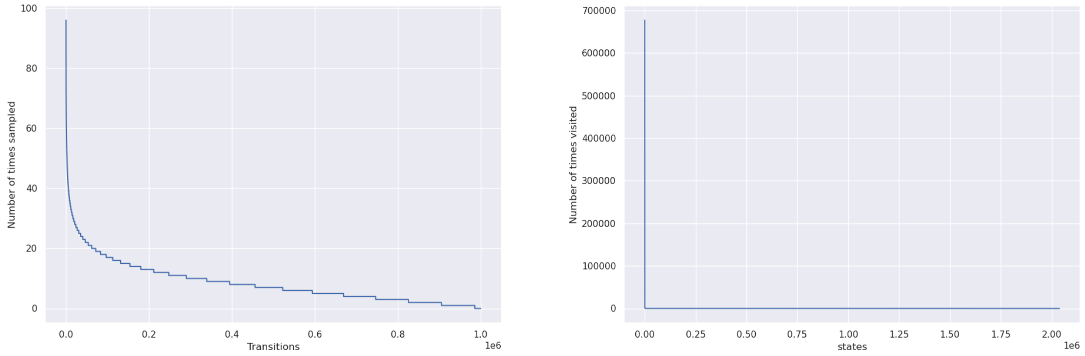
A.6 Plots for MinAtar Seaquest



a / Transition sampling distribution.

b / State sampling distribution.

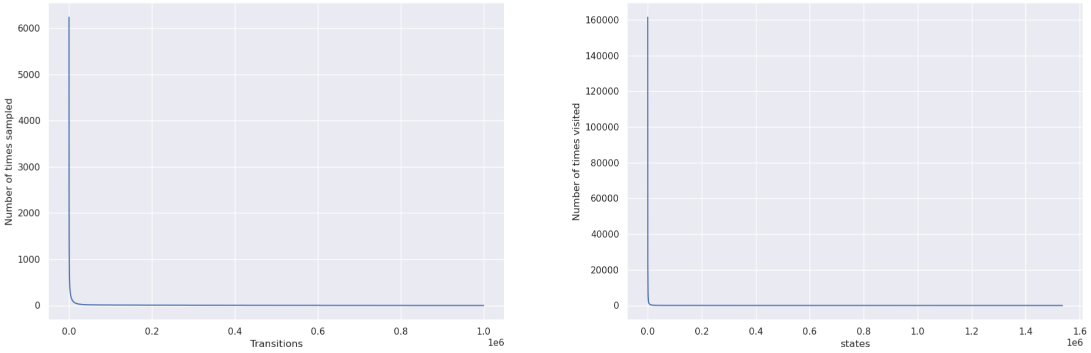
Figure A.6a / Sampling frequencies for UER.



c / Transition sampling distribution.

d / State sampling distribution.

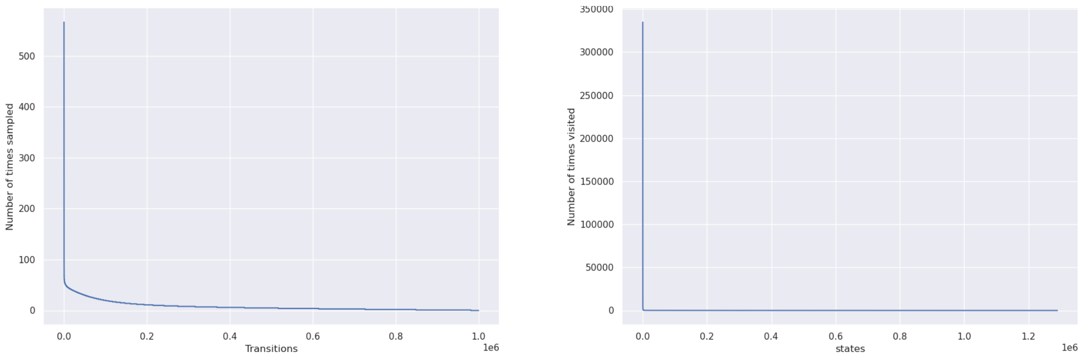
Figure A.6b / Sampling frequencies for PER.



e / Transition sampling distribution.

f / State sampling distribution.

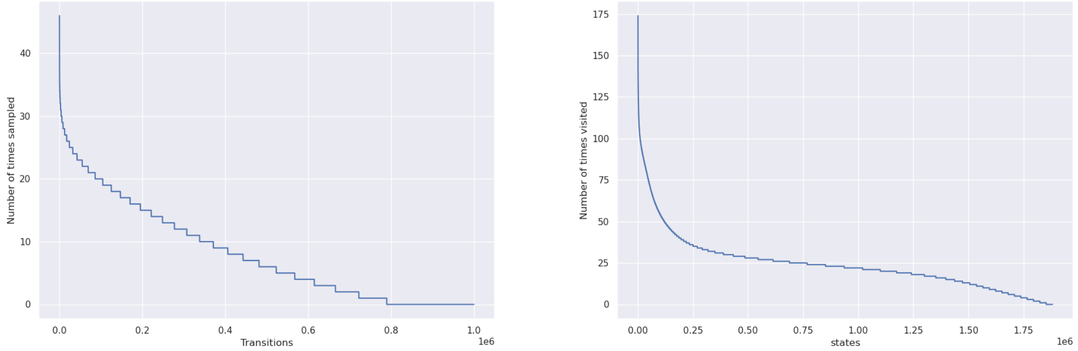
Figure A.6c / Sampling frequencies for TER.



g / Transition sampling distribution.

h / State sampling distribution.

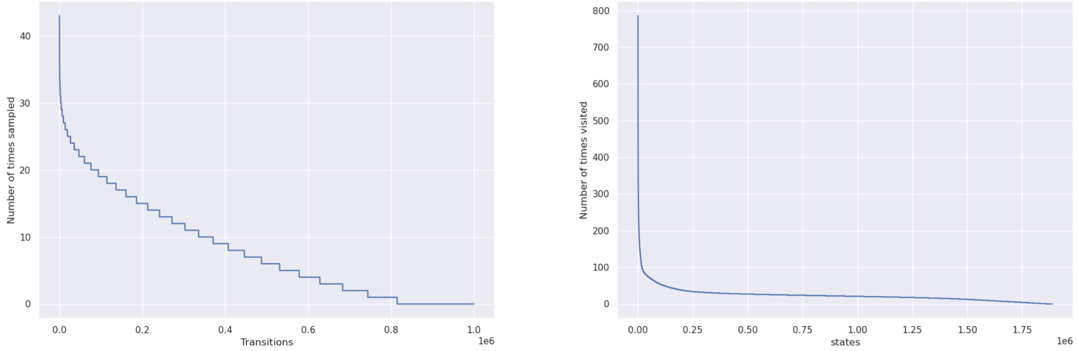
Figure A.6d / Sampling frequencies for TER with batch mixing.



i / Transition sampling distribution.

j / State sampling distribution.

Figure A.6e / Sampling frequencies for USR.



k / Transition sampling distribution.

l / State sampling distribution.

Figure A.6f / Sampling frequencies for USAR.