Information-Theoretic Exploration with Successor Uncertainties

Ioannis Tsetis

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
MPhil in Machine Learning and Machine Intelligence

Wolfson College August 2020

Declaration

I, Ioannis Tsetis of Wolfson College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

The code that was used for this dissertation was build upon the one that was used for the Successor Uncertainties paper [Janz et al., 2019]: https://github.com/DavidJanz/successor_uncertainties_tabular

This dissertation contains 11800 words excluding bibliography, photographs and diagrams but including tables, figure captions, footnotes and appendices.

Ioannis Tsetis
August 2020

Acknowledgements

First of all, I would like to thank my supervisors, José Miguel Hernández-Lobato and Robert Pinsler for their excellent advice and guidance throughout this project. They were always ready to provide me with feedback on my work and give me ideas whenever I found myself stuck.

I would also like to thank David Janz, for his brilliant insights and suggestions on how to approach this subject.

Finally, I would like to dedicate this thesis to my family for their constant and unconditional support.

Abstract

The exploration and exploitation trade-off dilemma has been at the heart of the reinforcement learning research community for a very long time. In recent years, the SU algorithm was created which achieved impressive results on hard tabular benchmarks and on the Atari 2600 domain. In this thesis we extend the original SU model by incorporating a secondary Q-function that accounts for the information gain which we use in a linear combination with the Q-function of the SU model to perform action selection. Based on this information-theoretic approach we propose two algorithms SU-IG1 and SU-IG2 respectively. We conduct experiments on these methods in the binary tree MDP environment where we find out that both algorithms manage to solve the binary tree MDP consistently achieving similar performance to that of the original SU model.

Table of contents

List of figures **xi**

List of tables **xiii**

Nomenclature **xv**

1 Introduction **1**

2 Reinforcement Learning **3**

2.1 Introduction to Reinforcement learning . . . 3

2.2 The reinforcement learning framework . . . 5

2.2.1 Policies, Goals and Value Functions . . . 6

2.3 Dynamic Programming . . . 7

2.4 Temporal Difference Learning . . . 8

3 Exploration vs Exploitation **13**

3.1 Exploration with Multi-Armed Bandits . . . 14

3.1.1 ε-greedy . . . 16

3.1.2 Upper confidence bounds . . . 17

3.1.3 Thompson Sampling . . . 19

3.1.4 Gradient Bandit Algorithms . . . 21

3.2 Deep Exploration . . . 22

3.2.1 Value function approximation . . . 22

3.2.2 Randomized Value Functions . . . 24

3.2.3 Deep Q-Networks . . . 24

3.2.4 Bootstrapped Deep Q-Networks . . . 26

4 Information Theoretic exploration with Successor Uncertainties **29**

4.1 Successor Uncertainties Framework . . . 30

4.1.1 SU Neural network model 32
4.2 Information-theoretic exploration 32
4.2.1 Motivation for Information Gain 35
4.3 Information Theoretic approach to Successor Uncertainties 35
4.3.1 Successor Uncertainties - Information Gain 1st Algorithm 36
4.3.2 Successor Uncertainties - Information Gain 2nd Algorithm 38

5 Experiments and Results **41**
5.1 Environment 41
5.2 Experiments 42

6 Conclusions **45**
6.1 Main contributions 45
6.2 Future work 46

References **47**

Appendix A Computation of the IG for the SU-IG1 algorithm **51**

Appendix B Hyperparameter selection **53**

List of figures

2.1 Many faces of RL 4
2.2 Agent-Environment Interaction 5

3.1 Q-network architecture 25

4.1 Multi-head neural network for the SU model 32
4.2 Multi-head neural network for the SU-IG1 model 38

5.1 Binary Tree MDP 41
5.2 Median number of episodes required for BDQN algorithm to find the optimal policy 42
5.3 Median number of episodes required for UBE algorithm to find the optimal policy 43
5.4 Median number of episodes required for SU algorithm to find the optimal policy 43
5.5 Median number of episodes required for SU-IG 1 algorithm to find the optimal policy 43
5.6 Median number of episodes required for SU-IG 2 algorithm to find the optimal policy 44
5.7 Median number of episodes required for Bootstrapped DQN algorithm to find the optimal policy 44

List of tables

B.1 Hyperparameters for SU and SU-IG experiments 53
B.2 Hyperparameters for Bootstrapped DQN experiments 53

Nomenclature

Acronyms / Abbreviations

BDQN	Bayesian Deep Q-Networks
CDF	Cumulative Distribution Function
DP	Dynamic Programming
DQN	Deep Q-Networks
ETC	Explore-then-commit
IG	Information Gain
KL	Kullback-Leibler
MI	Mutual Information
PSRL	Posterior Sampling for Reinforcement Learning
RL	Reinforcement Learning
RVF	Randomized Value Functions
SGD	Stochastic Gradient Descent
SU	Successor Uncertainties
TD	Temporal Difference
TS	Thompson Sampling
UBE	Uncertainty Bellman Equation
UCB	Upper confidence bounds

Chapter 1

Introduction

One of the main purposes of artificial intelligence is to solve advanced real world challenges. To solve these advanced real world challenges the researchers have turned to Reinforcement Learning (RL) and games. In the past games such as Backgammon [Tesauro, 1994], Chess [Campbell et al., 2002] , have served as an important stepping stones for improving RL and more recently huge advancements have been made in more complex games such as Atari [Mnih et al., 2013], Go (with AlphaGo [Silver et al., 2016] and AlphaGo Zero [Silver et al., 2017]) and Starcraft [Vinyals et al., 2019].

Even though these algorithms have achieved extremely impressive results, the exploration schemes that they used were naive and they needed enormous quantities of data. For example in the case of [Silver et al., 2016], neural networks were trained over hundreds of billions to trillions of simulated games. Thus, to solve real world complex problems in the future, we need to design algorithms for our agents, to enable them to perform efficient decision making. These agents must be able to find a balance between exploration and exploitation. Exploration refers to performing an action with the objective of getting more information from the unknown environment, while exploitation refers to taking advantage of the knowledge that the agent has accumulated in the past with the purpose of selecting the optimal action. The search for a balance between exploration and exploitation has been a very important topic for the RL research community for a long time and it still continues to be ([Kaelbling et al., 1996],[Sutton and Barto, 2018]).

In tabular settings, PSRL (Posterior Sampling for Reinforcement Learning) ([Strens, 2000],[Osband et al., 2013]) (which is also known as Thompson Sampling(TS)), has proven to be a successful approach in balancing the exploration/exploitation trade-off, where it has managed to achieve impressive results and close to optimal regret. Inspired by PSRL, the Successor

Uncertainties (SU) [Janz et al., 2019] algorithm was created.

SU is a scalable, model-free and easy to implement Randomised Value Function (RVF) algorithm, that has proven to be highly effective on hard tabular exploration problems and on Atari 2600 where it managed to outperform its competitors such as Bootstrapped DQN ([Osband et al., 2016]) and Uncertainty Bellman Equation ([O'Donoghue et al., 2017]) on 36/49 games and on 43/49 games respectively.

As it is explained by [Janz et al., 2019], one of the possible weaknesses of the SU algorithm is that it may underestimate the Q-function uncertainty. One way to combat that is to use an information-theoretic approach to quantify the uncertainty. As it is explained in 4.2.1, Information Gain (IG) is a promising avenue for quantifying this uncertainty.

In this thesis we give a thorough explanation of the exploration vs exploitation trade-off and we highlight the importance of efficient deep exploration. The main contribution of this thesis is the creation of two algorithms that leverage the Information Gain (IG) quantity in the Successor Uncertainties (SU) model. For this comparison the proposed methods get compared with other similar algorithms of the RL literature in the binary tree MDP environment.

The rest of this thesis is organized as follows: In Chapter 2, we introduce and motivate the core concepts of RL, explain its framework as well and present some of the most well-known algorithms in the RL literature such as SARSA and Q-learning. In the first part of Chapter 3, we start by explaining how multi-arms bandits work by looking at some very important RL algorithms such as epsilon-greedy, upper confidence bounds, TS as well as gradient bandit algorithms while in the second part we explain the necessity for agents to perform deep exploration and algorithms that do so, such as Bootstrapped DQN. In Chapter 4, we give more details into how the SU algorithm works and we present the extended models as well as their implementation details. In Chapter 5, we show the experiments conducted on the extended models and the results of these experiments. Finally, Chapter 6 summarizes the work and discusses possible ideas for future work.

Chapter 2

Reinforcement Learning

In this Chapter, we motivate the subject of reinforcement learning, introduce its fundamental principles and its framework. More specifically, we explain the concept of dynamic programming and temporal difference learning and we introduce important RL algorithms such as SARSA, expected SARSA and Q-learning.

2.1 Introduction to Reinforcement learning

If we think in terms of evolutionary history, all animals exhibit some kind of behavior: they perform actions according to the inputs that they receive from their environment. Animals adapt based on their observations and their behavior changes over time. This procedure is called learning through experience and it has been vital for the evolution of the human species in the past. Inspired by behaviorist psychology, RL was created as an area of machine learning with the purpose of mimicking the way humans learn. It has revolutionized our understanding of learning in the brain and it has been widely used in many fields as illustrated in 2.1.

RL is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Supervised learning models learn from a training set of labeled examples provided by a knowledgeable external supervisor while unsupervised learning is typically trying to find a hidden structure based on the collection of unlabeled data. Both supervised learning and unsupervised learning create models based on given data. On the other hand, similarly to humans an RL agent creates data (also known as experience) by interacting with the environment and makes decisions based on the knowledge that it has accumulated.

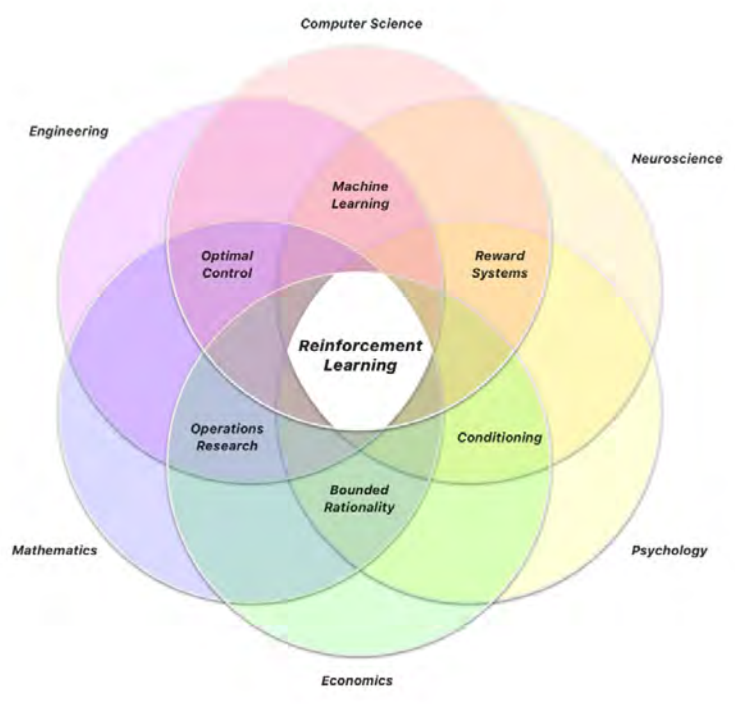

Fig. 2.1 Influence of Reinforcement learning in many scientific fields

Some of the key differences between RL and the other ML paradigms are the following:

1. There is no supervisor that knows the optimal action, there is just a reward signal, thus RL methods learn by trial and error.
2. The feedback is delayed, because the evaluation of an action can change according to what happens in subsequent interactions.
3. Time really matters in RL, most of RL problems are sequential.
4. The agents actions affect the subsequent data that it receives, because by performing specific actions it may explore different parts of the state space.

2.2 The reinforcement learning framework

In standard reinforcement learning, an agent is placed in an environment, performs actions and attains rewards as illustrated in 2.2. This agent-environment interaction can be modelled as a Markov Decision Process (MDP).

In a MDP, the learner and the decision maker is called the agent while the thing it interacts with, which comprises of everything outside the agent, is called the environment. The agent interacts continuously with the environment by selecting actions, while the environment based on these actions presents new situations to the agent. The environment also yields rewards, and the purpose of the agent (also known as goal) is to learn through experience how to select the actions that will make it attain the maximum possible reward in the long run.

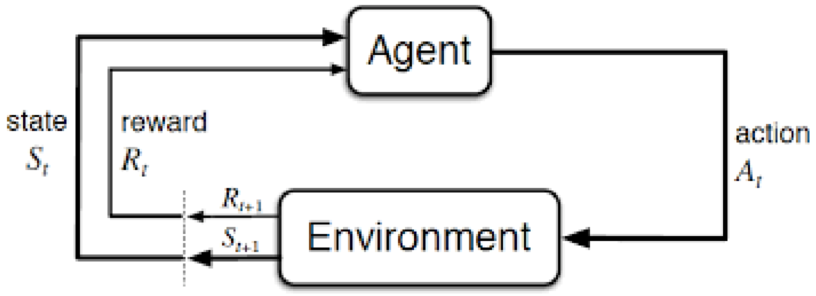

Fig. 2.2 Agent-Environment Interaction

The MDP can be expressed as a tuple ($\mathscr{S}$, $\mathscr{A}$, R, P, γ) where:

1. $\mathscr{S}$ is the state space.
2. $\mathscr{A}$ is the action space.
3. $R(s,a)$ is a random variable which represents the reward in a state action pair (s,a), where $s \in \mathscr{S}$ and $a \in \mathscr{A}$
4. $P : S \times S \times A \to [0,1]$ symbolizes the state-transition probability from a (s,a) pair to the new state s' where it is written as $P(s'|s,a)$, where $s \in \mathscr{S}$ and $a \in \mathscr{A}$.
5. $\gamma \in [0,1]$ is the discount factor .

2.2.1 Policies, Goals and Value Functions

The policy is a mapping from states, to probabilities of selecting each possible action. More specifically a policy $\pi : S \times A \to [0,1]$ describes the probability of being in state $s \in \mathscr{S}$ and selecting an action $a \in \mathscr{A}$.

By iterating the MDP illustrated in 2.2 the agent follows a trajectory that visits a series of states and accumulates a sequence of rewards.

The goal G (also known as return) is a random variable that is defined as the discounted sum of rewards obtained by taking action $a \in \mathscr{A}$ from state $s \in \mathscr{S}$ and acting according to policy π. It is formulated in the following equation:

$$G_t^\pi = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.1}$$

Most of the RL algorithms are dealing with value function estimations which evaluate a specific state s or a state action pair (s,a). In RL there are two kind of value functions, the state value functions and the state-action value functions.

The state value functions are functions of states the estimate how good it is for an agent to be in a particular state. They are mostly used in model-based RL where the dynamics of the model are known. The equation for the state value function is the following:

$$V^\pi(s) = \mathbb{E}[G_t | S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s], \tag{2.2}$$

The state-action value functions are functions of state and actions that estimate how good it is for an agent to be in a particular state and perform a specific action. These value functions are mostly used in model-free RL where the dynamics of the model are unknown. In the tabular case, the complexity of computing the state-action value function is an order higher than the complexity of the state value function, however as we explain later the state-action value function is more practical when it comes to finding the optimal policy in a model-free RL problem. The equation for the state-action value function is the following:

$$Q^\pi(s,a) = \mathbb{E}[G_t | S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1|S_t=s,A_t=a}], \tag{2.3}$$

2.3 Dynamic Programming

Dynamic programming (DP) in RL refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP. Traditional DP algorithms are not used in state of the art RL because they assume a perfect model (knowledge of the dynamics of the environment), however they are still important theoretically because they set the foundation for understanding the algorithms presented in this Chapter.

DP algorithms are obtained by turning Bellman equations as explained below, into update rules that approximate the value functions.

Bellman Equation

In the dynamic programming there are two equations [Bellman, 1954] that express a relationship between the value of a state and values of its successor states.

The first equation, known as Bellman expectation equation describes how expected returns at each state-action pair $(s,a) \in \mathcal{S} \times \mathcal{A}$ are related to the expected returns at the next state-action pair $(s',a') \in \mathcal{S} \times \mathcal{A}$ and it is computed by making simple manipulations on 2.1:

$$Q^{\pi}(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P,\pi}[Q^{\pi}(s',a')], \tag{2.4}$$

In most RL problems, the purpose is to find a policy π that maximises the expected return. The second equation is the method that computes that and it known as the Bellman optimality equation:

$$Q^{*}(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P}[\max_{a' \in \mathcal{A}} Q^{*}(s',a')], \tag{2.5}$$

The following update rules are created from 2.4 and 2.5:

$$Q_{k+1}(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P,\pi}[Q_k(s',a')] \tag{2.6}$$

$$,Q_{k+1}(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P}[\max_{a' \in \mathcal{A}} Q_k(s',a')], \tag{2.7}$$

These two equations work recursively and provide increasingly better approximations of Q^{π}

and Q^* respectively.

The equations in 2.7 can also be turned to the Bellman expectation operator $\mathscr{T}^\pi$ and Bellman optimality operator $\mathscr{T}$ respectively:

$$\mathscr{T}^\pi Q(s,a) := \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P,\pi}[Q(s',a')] \tag{2.8}$$

$$,\mathscr{T} Q(s,a) := \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P}[\max_{a' \in \mathscr{A}} Q(s',a')], \tag{2.9}$$

In [Bertsekas and Tsitsiklis, 1996] it is explained that these two operators can be turned to contraction mappings that ensure that a unique fixed point exists for both the Bellman operators described in 2.9 and these fixed points are Q^π and Q^* respectively.

These Bellman operators are very important for explaining the algorithms that are introduced in the next section.

2.4 Temporal Difference Learning

One of the most important ideas in RL is Temporal Difference (TD) learning. TD learning is a combination of Monte Carlo and DP ideas. Similarly to Monte Carlo methods, TD methods can learn directly from experience without needing to know the dynamics of the environment and like DP, TD methods use bootstrap. They update estimates that are based in part on previously learned estimates without waiting for a final outcome. More specifically the approximation of the state-action value function is recursively updated based on the following equation:

$$Q_{k+1}(s_t,a_t) = Q_k(s_t,a_t) + \alpha[R_{t+1} + \gamma Q_k(s_{t+1},a_{t+1}) - Q_k(s_t,a_t)], \tag{2.10}$$

In this equation, $R_{t+1} + \gamma Q_k(s_{t+1},a_{t+1})$ is the target value function, the quantity $\delta_t = R_{t+1} + \gamma Q_k(s_{t+1},a_{t+1}) - Q_k(s_t,a_t)$ is the error and a is the step-size. More specifically the error is used to measure the difference between the estimated value of S_t and the target value function which we are trying to move towards to. α is the learning rate which is used for regulating how much we want to forget our previous estimate of the state-action value function. In non-stationary problems we prefer a higher value of α while in stationary problems we prefer a lower one.

Two of the most popular algorithms are introduced, namely SARSA and Q-learning which use the TD learning methodology to compute the state-action value function.

To start with, the difference between on-policy and off-policy methods is explained. On-policy methods learn about policy π from experience sampled from the same policy π, while off-policy methods learn about policy π from experience sampled from another policy μ. On a higher level, on-policy learning is like learning "on the job" while off-policy learning is like "looking over someone's shoulder and trying to mimic their behaviour".

SARSA

SARSA [Rummery and Niranjan, 1994] is an on-policy algorithm, that takes its name from how an agent interacts with the environment. An agent that starts in a state s, takes an action a, gets a reward r, transitions to a new state s' and selects a new action a' hence the acronym SARSA and it is formulated as a quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$. In SARSA, the concept of bootstrapping is used, which means that the estimate for the state-action value function Q is updated by the error δ_t so that the previous estimate moves towards a new estimate by a factor of α which corresponds to the step size. SARSA uses the update function described in 2.10 and the pseudocode for the algorithm is presented below:

Algorithm 1 SARSA

```
1: Initialise $Q(s,a)$ for all $s \in \mathscr{S}$, $a \in \mathscr{A}$
2: **repeat**(for each episode)
3:     Initialise state $s$
4:     **repeat**(for each step of the episode)
5:         Choose action $a$ from $s$ using policy derived from Q
6:         Take action $a$, observe reward $R$ and transition to a new state $s'$
7:         $Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma Q(s',a') - Q(s,a)]$
8:         $s \leftarrow s'$
9:     **until** s is terminal
10: **until** convergence
```

An interesting variation to the SARSA algorithm is the expected SARSA algorithm which takes into account how likely each action is under the current policy. Expected SARSA has lower variance than SARSA because it eliminates the variance of the random selection for a_{t+1}, however that comes at the cost of a more expensive computation of the expected value. The update rule of expected SARSA is described as shown below:

$$Q_{k+1}(s_t,a_t) = Q_k(s_t,a_t) + \alpha[R_{t+1} + \gamma \sum_{a_{t+1}} \pi(a_{t+1}|s_{t+1}) Q_k(s_{t+1},a_{t+1}) - Q_k(s_t,a_t)], \quad (2.11)$$

Q-learning

Q-learning [Watkins and Dayan, 1992] is an off-policy TD algorithm that is based on Bellman optimality equation 2.5. Q-learning directly approximates the optimal state-action value function $q*$, independent of the policy being followed. This simplifies significantly the analysis of the algorithm and enables proofs of early convergence. The policy still has an effect on this algorithm because it determines which state-action pairs are visited and updated. The update rule is shown below:

$$Q_{k+1}(s_t,a_t) = Q_k(s_t,a_t) + \alpha[R_{t+1} + \gamma \max_{a_{t+1}} Q_k(s_{t+1},a_{t+1}) - Q_k(s_t,a_t)], \quad (2.12)$$

The pseudocode for the algorithm is presented below:

Algorithm 2 Q-Learning

```
1:  Initialise $Q(s,a)$ for all $s \in \mathscr{S}$, $a \in \mathscr{A}$
2:  **repeat** (for each episode)
3:      Initialise state $s$
4:      **repeat** (for each step of the episode)
5:          Choose action $a$ from $s$ using policy derived from Q
6:          Take action $a$, observe reward $R$ and transition to a new state $s'$
7:          $Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma \max_a Q(s',a') - Q(s,a)]$
8:          $s \leftarrow s'$
9:      **until** s is terminal
10: **until** convergence
```

TD learning aims to produce an estimate of the state-action value function for each separate state-action pair by creating a look-up table. Unfortunately there are two major problems with this tabular approach. First of all, if the state or action space is large, it becomes impractical due to the large memory that would be required to store this information. Secondly, if the state space is large, it is too slow to learn the value of each state individually because it will be visited very few times, thus getting a reliable estimate will be very hard.

Both of these problems can be addressed with value function approximations that allow a parametrised representation of the state-action value function such as $Q(s,a;\theta) \approx Q(s,a)$. Value function approximation is described in more detail in 3.2.1 .

Chapter 3

Exploration vs Exploitation

A long-standing problem in the RL community is how to balance the trade-off between exploration and exploitation. An RL agent at any state has to select an action to perform. These actions can be either with the purpose of exploring or with the purpose of exploiting. Exploiting refers to the agent using the knowledge that it has accumulated in the past with the purpose of making the optimal decision based on what he knows, while exploring refers to the agent acquiring more knowledge with the purpose of making the optimal decisions in the future. The dilemma of whether to exploit or explore, happens all the time in our everyday life, where some examples are:

1. Restaurant selection: Going to your favorite restaurant (exploit) vs Sampling a new restaurant (explore).
2. Watching a movie : Watching your favorite movie (exploit) vs Watching a movie that you have never seen before (explore).

RL aims to solve real-world problems such as these, where the optimal decision is not known a priori, thus it uses an evaluative feedback of rewards to indicate how good an action was. Exploiting in RL means getting the immediate best possible reward while exploring refers to sacrificing immediate reward with the intention of getting better information which leads to more rewards in the future. Sometimes, the best long-term strategy may involve short term sacrifices for the sake of accumulating information.

In this Chapter, we study the trade-off of exploration vs exploitation in two sections. In the first section, we simplify the problems that we are trying to solve in RL into a pure decision making problem by introducing the concept of multi-armed bandits. To be more precise in this first section we introduce and explain algorithms that approach the exploration and exploitation trade-off with an Explore-then-commit (ETC) approach. After that, we dive deeper into more sophisticated algorithms that use the concept of "Optimism in the face of uncertainty" to encourage continuous exploration and we finish the first section with gradient bandit algorithms, which are methods that learn a numerical preference and use it for action selection. In the second section, we explain why the bandit learning theory does not adequately address the full reinforcement learning problem and we showcase the necessity for deep exploration. Additionally, in this section, we explain the problems that occur when tabular representations are used and how function approximations can solve these problems. Then we showcase the concept of Randomized Value Functions (RVF) and the reason why RVF algorithms incentivize deep exploration. Finally we conclude this section and this chapter with the explanation of Bootstrapped DQNs which has achieved very impressive results in problems which require deep exploration.

3.1 Exploration with Multi-Armed Bandits

The multi-armed bandit problem is a classic reinforcement learning problem that exemplifies the exploration exploitation trade-off dilemma. More specifically in a multi-armed bandit problem the agent has to select between many competing choices of which it has no idea what rewards each one yields. To solve that, the agent selects actions with the purpose of simultaneously learning and acquiring as much reward as possible. The agent's goal is to select the actions that maximize the expected reward. Thus, this problem can be thought as a simplification for the MDP framework as it was introduced in Chapter 1 where the simplication is that the agent does not transition between states. Because of this simplication, a lot of research has been conducted in RL using multi-arm bandits trying to find the optimal strategy for decision making. Even with this simplification, the problem of decision making still remains very hard and explaining this framework is extremely important. The framework for the multi-armed bandit is the following:

1. A multi-armed bandit is a tuple $(\mathscr{A}, \mathscr{R})$
2. $\mathscr{A}$ is a known set of actions (also known as "arms")
3. $\mathscr{R}^a(r) = \mathscr{P}(R = r, A = a)$ is an unknown probability distribution over rewards
4. At each time step t, the agent selects an action $A_t \in \mathscr{A}$
5. Based on action selected, the environment generates a reward $R_t \sim \mathscr{R}^{A_t}$
6. The goal of the agent is to maximize the cumulative reward in the long run $\sum_{\tau=1}^{t} R_t$

In multi-armed bandits the action value function evaluates how much reward R we are expected to get if we choose action a and it is defined as:

$$q(a) = \mathbb{E}[R|A = a], \tag{3.1}$$

and the optimal value $q(a_*)$ is defined by taking the maximum over the $q(a)$ values as shown:

$$q(a_*) = \max_{a \in \mathscr{A}} q(a), \tag{3.2}$$

Whenever we make a step, we get some opportunity loss which is known as *regret* and it is computed as:

$$l_t = \mathbb{E}[q(a_*) - q(A_t)], \tag{3.3}$$

and the *totalregret* is the total opportunity loss which is accumulated over time and is defined as:

$$l_t = \mathbb{E}[\sum_{\tau=1}^{t} q(a_*) - q(A_t)], \tag{3.4}$$

Trying to minimise the total regret is the same as what we showed in Chapter 2 with the MDPs, where we tried to maximise the cumulative reward.

The gap Δ_a is the difference in value between action a and optimal action a_* as shown below:

$$\Delta_a = q_* - q(a), \tag{3.5}$$

Regret can be expressed as a function of gaps and counts as shown below:

$$L_t = \sum_{a \in \mathscr{A}} E[N_t(a)]\Delta_a, \tag{3.6}$$

where the count $N_t(a)$ is the expected number of selections for action a.

Based on this framework, when an algorithm explores forever (uniform random exploration), it accumulates regret over time, thus the algorithm cannot achieve better regret than linear. The same thing holds for an algorithm that exploits forever (such as a greedy algorithm), because it might lock-on to a sub-optimal action and thus the algorithm will not do better than linear regret.

It has been proved that the performance of any algorithm is determined based on the similarity of the optimal action to the other actions, thus it can be formally described by the gap Δ_a and the similarity in distributions, which is measured by the KL-Divergence [Kullback and Leibler, 1951] $KL(R^a||R^{a*})$. The lower bound for asymptotic total regret is at least logarithmic in the number of steps as it has been proved by [Lai and Robbins, 1985]:

$$\lim_{t \to \infty} L_t \geq \log t \sum_{a|\Delta_a > 0} \frac{\Delta_a}{KL(R^a||R^{a*})}, \tag{3.7}$$

3.1.1 ε-greedy

One of the first and most basic algorithms that tries to balance exploration with exploitation is the ε-greedy algorithm. The ε-greedy algorithm selects greedily an action (exploitation) with probability 1-ε and selects randomly an action (exploration) with probability ε. In the limit, as the number of steps increases, every action will be sampled an infinite amount of times, thus ε-greedy ensures that in the limit, the algorithm will know which action is optimal $q(A_t) = q(a_*)$. Unfortunately, ε-greedy cannot surpass the linear regret because even when it knows which is the optimal action, it still selects a suboptimal action with probability ε.

Decayed ε-greedy

A way to improve on the ε-greedy algorithm is to put a decay in ε so that it initially encourages exploration while as the time progresses it shifts towards more and more exploitation. In the limit this algorithm reaches a point where it always selects the optimal action $q(A_t) = q(a_*)$, thus decayed ε-greedy can reach logarithmic asymptotic total regret. Unfortunately in practise, finding the correct ε requires knowledge of the gaps between the actions in advance which in most of the cases are unknown.

Optimistic Initial Values

ε-greedy methods depend to some extent on the initial action-value estimates $Q_1(a)$ which means that these methods are statistically biased. For the methods that use a sample-average, the bias disappears for each action after the first time it has been selected, however for the methods that use a constant α the bias is permanent and decreases over time. This bias becomes a set of parameters that has to be picked by the user, which can also be taken advantage of to encourage exploration. By setting the initial values to the maximum possible value and by using a greedy policy, we encourage the bandit to explore. Whichever actions are initially selected the possible reward for picking them again in a subsequent action selection will be lower than the other actions, thus the agent will switch to other actions. This pattern will get repeated for the first episodes and the result of that is all the actions will be tried several times before the value estimates converge.
This trick can be extremely effective on stationary problems, however it is not suited at all for nonstationary problems because its drive for exploration is inherently temporary.

3.1.2 Upper confidence bounds

In the previous sections the algorithms used an explore-then-commit (ETC) methodology which focuses on exploring at the early stages and exploiting at the later ones based on the accumulated knowledge. The drawback with this methodology is that most of the important problems that we want to solve in RL are non-stationary problems that require constant exploration. To solve that there is a category of algorithms that use upper confidence bounds (UCB) to encourage continuous exploration. To be more precise, these UCB algorithms explore actions using the principle of "Optimism in the face of uncertainty". This principle advocates the agent to pick the action that has the most potential of being the best, rather than the one that is observed to be the best based on the current knowledge. Thus, UCB takes into account both how close the estimates of the actions are to being optimal as well as the uncertainties of those estimates. There are two approaches to compute the UCB, the first one being the frequentist while the second one is the probabilistic approach.

Frequentist approach

In the frequentist approach the UCB algorithm initially estimates an upper confidence bound $U_t(a)$ for each action value such that $q(a) \leq Q_t(a) + U_t(a)$. The upper confidence $U_t(a)$ is chosen depending on the number of times $N(a)$ that the action has been selected. If $N(a)$ is small then $U_t(a)$ must be large because the estimated value still has a lot of uncertainty while

If $N(a)$ is large then $U_t(a)$ must be small because the estimated value is accurate. Finally, the action is selected by maximising the UCB as shown in the following equation:

$$A_t = arg \max_{a \in \mathscr{A}} Q_t(a) + U_t(a), \tag{3.8}$$

There are many variations of the UCB algorithm, where one of them is the UCB1 Auer et al. [2000], which selects actions as shown in the following equation:

$$A_t = arg \max_{a \in \mathscr{A}} Q_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}}, \tag{3.9}$$

The UCB1 algorithm has proved to achieve logarithmic asymptotic total regret equal to:

$$\lim_{t \longrightarrow \infty} L_t \leq 8 \log t \sum_{a | \Delta_a > 0} \Delta_a, \tag{3.10}$$

It is important to note that the UCB that is used in the UCB1 algorithm is very weak because it is produced by the Hoeffding's inequality [Hoeffding, 1963] which does not make any assumption about the reward distribution for the bandits. This in turn means that when more specific distributions are chosen, tighter bounds can be generated.

Bayesian approach

In the Bayesian approach, the bandits take advantage of the prior knowledge they have about the rewards $p(R^a)$ to construct a UCB. If the prior knowledge that we have is accurate and reliable, then the Bayesian approaches are expected to work better than the Frequentist approaches. The framework for this problem which is also known as Bayesian bandits is the following:
Initially, we assume a distribution $p(Q|w)$ over the action-value functions with parameter w which acts as our likelihood. Then, by using Bayesian methods the posterior distribution over the weights w is computed $p(w|R_1, ..., R_t)$ which acts as our prior. By using the Bayes rule and multiplying the prior with the likelihood we get the posterior distribution over the action-values:

$$p(Q(a)|R_1,R_2,...,R_{t-1}) = p(Q(a)|w)p(w|R_1,...,R_t), \tag{3.11}$$

From the computed posterior, we can now estimate the upper confidence bound and our action selection step will become as shown below:

$$A_t = arg\max_{a\in\mathscr{A}} Q_t(a) + c(\sigma(a)), \tag{3.12}$$

where c is the number of standard deviations that we want to add to our action-value estimate. Higher value of c encourages more exploration while lower value of c encourages more exploitation.

A second way to take advantage of the computed posterior distribution 3.11 is to do probability matching. In probability matching, we select an action a according to the probability that this action a is the optimal action,so a policy is created as shown below:

$$\pi(a) = p[Q(a) = \max_{a'} Q(a')|R_1,...,R_{t-1}], \tag{3.13}$$

Probability matching is a heuristic that is optimistic in the face of uncertainty because the uncertain actions have a higher probability of taking the maximum value. The way that probability matching is implemented in practise is an algorithm known as Thompson Sampling.

3.1.3 Thompson Sampling

Thompson sampling (TS), is a sample-based probability matching algorithm that has the following policy:

$$\pi(a) = \mathbb{E}[\mathbb{1}(Q(a) = \max_{a'} Q(a'))|R_1,...,R_{t-1}], \tag{3.14}$$

The algorithm uses Bayes rule as mentioned in the previous section to compute the posterior distribution. Then it samples an action-value function $Q(a)$ from the posterior and it selects an action by maximising the sample:

$$A_t = arg\max_{a \in \mathscr{A}} Q(a), \tag{3.15}$$

It has been proved that in the case of Bernoulli bandits TS achieves Lai and Robins lower bound on regret 3.7.

TS and UCB algorithms share the same principle of "Optimism in the face of uncertainty" and has been proved that one can translate regret bounds established for UCB algorithms to Bayesian regret bounds for TS [Russo and Roy, 2013].

Applying Thompson Sampling to a MDP

TS can be applied to an MDP by using PSRL (Posterior Sampling for Reinforcement Learning), a method that is based on two components:

1. In the first step, a distribution over rewards and transition dynamics $P_{\hat{T}}$ is obtained using a Bayesian modelling approach, treating rewards and transition probabilities as random variables.
2. In the second step, the PSRL algorithm , samples $\hat{T} \sim P_{\hat{T}}$, computes the optimal policy $\hat{\pi}$ with respect to the sampled $\hat{T}$, and follows $\hat{\pi}$ for the duration of a single episode. The collected data are then used to update the $P_{\hat{T}}$ model and the whole process is iterated until convergence.

While PSRL works very well on tabular problems, it does not scale well enough beyond that because the second step becomes very expensive.

This large computational cost of TS in RL has motivated researchers to come up with approximations. A common approach is to work with the posterior distribution over Q-functions rather than with the posterior distribution over transition distributions. This approach is known as Randomised Value Functions (RVF) [Osband et al., 2017] and it is explained in details in section 3.2.2.

3.1.4 Gradient Bandit Algorithms

The methods that we previously used, estimate the action values and use these estimates for action selection. While this is a good approach, it is not the only approach that can be used. Gradient bandit algorithms learn a numerical preference for each action a which is denoted as $H_t(a)$. The higher the numerical preference is the more often that action is selected, however this numerical value has no interpretation in terms of reward value and the only that is important is the relative preference of one action over another. The action probabilities in this model are determined according to a soft-max distribution such as Gibbs or Boltzmann as shown below:

$$P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(a), \tag{3.16}$$

where $\pi_t(a)$ stands for the probability of selecting action a at time step t. The action preferences are initially set to the same value so that all actions have an equal probability of being selected.

By combining the theory mentioned above and the idea of stochastic gradient ascend the bandit gradient algorithm is produced. On each time step, after selecting an action A_t and receiving the reward R_t, the action preferences are updated as shown below:

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \qquad \textit{and} \tag{3.17}$$

$$H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R}_t)(\pi_t(a)), \qquad \forall\ a \neq A_t \tag{3.18}$$

where a is the step-size parameter, and $\bar{R}_t \in \mathscr{R}$ is the average of all the rewards up to and including time t. The important term here is $\bar{R}_t$ because it acts as a baseline with which the reward is compared. If the reward surpasses this baseline, then the probability of taking the action A_t in the future is increased while if the reward is below it the probability gets decreased. The actions that were not selected, always move in the opposite direction of the selected action.

3.2 Deep Exploration

As described in the first section, the bandit methods select actions without thinking about the delayed consequences. When these methods are applied to a RL problem they usually select the action a_t only if the expected value $Q(s_t, a_t)$ is large or the observed rewards and/or transitions are expected to provide useful information. This approach is called in the RL literature as myopic exploration, because it incentivizes exploration over a single time step. Due to their myopic exploration the bandits do not fully address the RL problem, where selecting an action can be desirable even if it does not yield immediate rewards or information just because this action may place the agent in a state that can lead to subsequent learning opportunities or rewards. This is the core reason and motivation behind deep exploration. The agent needs to consider how the actions influence downstream learning opportunities. So, the agent needs to search deep in his decision tree.

3.2.1 Value function approximation

In all the methods that we described in Chapter 2, we used a tabular representation for the value functions, however when it comes to real world complex problems these representations are both expensive and impractical. They are impractical because they do not generalize well. An example of the impracticality is, if we have a robot that moves on a grid we may not want it to count as different states, positions that are one millimeter apart from one another. They are also expensive because in these problems we need a lot of memory to store them and a lot of computational power to manage them. To combat these problems, we turn to value function approximations.

In value function approximation the value function is represented in a parameterized functional form with weight vector $w \in \mathscr{R}^d$. We write $\hat{v}(s, w) \approx v_\pi(s)$ to represent the approximate value of state s given weight vector w. The most common function approximators $\hat{v}$ in RL are those that are based on a linear combination of features or on artificial neural networks.

Prediction Objective

In the tabular case there was no need for an error term because the value function could be equal to the true value function. That is not the case when we use a value function approximation because an update at one state affects many other states and it is not possible to get the true value for all the states. By assumption the states are far more than the weights, so making one state's estimate more accurate means that we make another state's estimate less accurate. This in turn leads to an error term which is called the Mean Squared Value Error and is denoted as $\overline{VE}$:

$$\overline{VE}(w) = \sum_{s\in\mathscr{S}} \mu(s)[v_\pi(s) - \hat{v}(s,w)]^2 \tag{3.19}$$

where $\mu(s) \geq 0$ represents how much we care about the error in each state s. The squared root of this measure gives a rough measure of how much the approximate values differ from the true values. In RL lowering this error does not guarantee us that we will find the optimal policy which is what we are trying to find, however minimizing this quantity can be very fruitful.

SGD in Value function approximation

One of the most popular methods that can help to minimize the $\overline{VE}$ is stochastic gradient descent (SGD). After each time we observe a state, the weight vector is adjusted by a small amount in the direction that would reduce the error the most as it is shown in the following expression:

$$w_{t+1} = w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t,w_t)]\nabla\hat{v}(S_t,w_t) \tag{3.20}$$

where α is the step-size parameter and $\nabla f(w)$ denotes the column vector of partial derivatives of the expression with respect to the components of the vector.

SGD methods have been proven to be successful both in the linear and the non-linear cases of value function approximation.

To summarize, in value function approximation the agent maintains a point estimate of a function mapping state-action pairs to expected cumulative reward. This estimate approximates the agent's expectation of the true value function and uses this expectation to perform action selection. The hope of value function approximation is that this process converges quickly on a mode in which the agent selects optimal or near optimal actions while the new observations reinforce the dominant value estimates.

By using this value function estimate to guide actions the agent can operate according to a greedy policy in which, at any state, the agent performs the action that maximizes the estimated value function. Unfortunately, this policy does not investigate poorly-understood actions that have been assigned with unattractive point estimates which may lead the agent to forgo enormous potential value because some actions remain unexplored. Thus, value function approximation does not solve the problem of deep exploration.

3.2.2 Randomized Value Functions

Build upon value function approximations, RVFs encourage deep exploration.

RVFs are inspired by Thompson Sampling as it was described in 3.1.3 and they work in two steps:

1. In the first step they directly model a distribution over Q functions, $P_{\hat{Q}}$.
2. In the second step, the agent acts greedily with respect to a sample $Q \sim P_{\hat{Q}}$ which is drawn at the beginning of each episode, which removes the main computational bottleneck that existed when sampling from the transition distributions.

At a high level, RVFs sample from a proxy of the posterior distribution over value functions. RVFs incentivize experimentation with actions that the agent is uncertain about, where this uncertainty translates into variance in the sampled value estimate. This randomness often generates positive bias and because of that it promotes deep exploration.

3.2.3 Deep Q-Networks

Deep Q-networks (DQN) were designed with the aim of learning successful policies directly from high-dimensional sensory inputs using end-to-end RL.

In [Mnih et al., 2015] the authors approximate the optimal state-action value function Q^* with a deep Convolutional Neural Network (CNN) which achieves performance equal to that of human professionals in the Atari 2600 games. CNNs were used due to past results that prove that they are a promising method for effectively processing visual images ([Lee et al., 2009], [Krizhevsky et al.])

DQN architecture

The DQN architecture consists of three convolutional layers followed by a non-linear fully connected layer. The figure that is illustrated below is taken from [Mnih et al., 2015]:

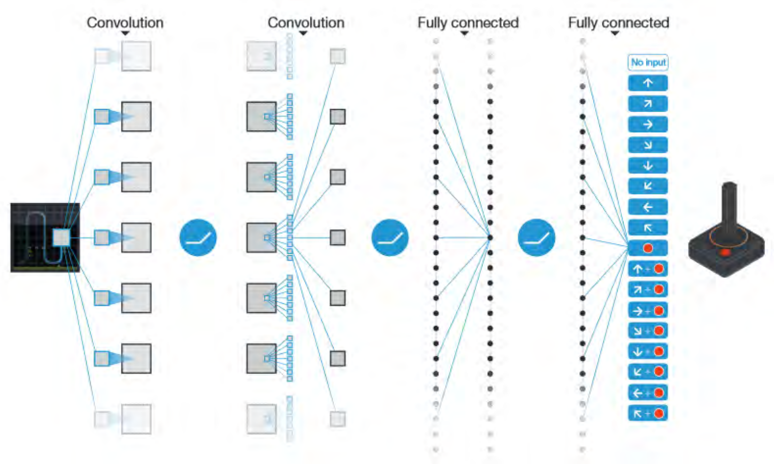

Fig. 3.1 Q-network architecture

DQN model

DQN models the optimal state-action value function $Q^*(s,a;\theta)$ where $s \in \mathscr{S}$, $a \in \mathscr{A}$ and θ the Q-network parameters. It initially considers the state s as the input of the network and the number of nodes of the output layer corresponds to the number of actions that the agent can perform. For a specific input state, each individual action gets assigned a Q-function value just by performing a single forward pass through the network.

In DQN, we parameterize the Q-network using the deep CNN as it is depicted in 3.1, in which θ_i are the parameters of the Q-network at iteration i. To perform experience replay, at each time step, we store the agent's experience as $e_t = (s_t, a_t, r_t, s_{t+1})$ in the replay buffer and we get a dataset D that contains the agents experience $e_1, ..., e_t$. During the learning phase, we apply Q-learning updates on samples (or mini-batches) of experience $(s,a,r,s') \sim U(D)$,drawn uniformly at random from the pool of stored samples. The Q-learning update at each iteration results in the following loss function:

$$L(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)^2], \tag{3.21}$$

where γ is the discount parameter, θ_i are the Q network parameters at iteration i and θ_i^- are the parameters used to compute the target at iteration i. The parameters of the target are updated with the Q network parameters every C steps while in between individual updates they remain fixed.

By differentiating the loss function with respect to the weights we get the following gradient:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i)) + \nabla_{\theta_i} Q(s,a;\theta_i)] \tag{3.22}$$

Instead of computing the expectation in the above gradient, it is more preferable to optimize the loss function with SGD.

The DQN algorithm improves upon the standard online Q-learning in two ways and makes it more suitable for training large neural networks:

1. 1) Experience Replay: Experience replay is a technique that at each time step t stores the agent's experiences in the form of tuples $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D = (e_1, ..., e_t)$. Then for the optimization step, random samples or mini-batches are sampled from the D. The advantages that this method has over the traditional Q-learning are two. Firstly, every step of experience can be used into more than one update of the weights, thus it is data efficient. Secondly, taking experiences randomly breaks the correlation between consecutive samples resulting into a reduced variance. Finally, the distribution is averaged over many of its previous states leading to more smooth learning without unwanted oscillations or divergences in the parameters.

2. 2) Network cloning: Every C updates, the Q-network is cloned and we obtain a target network $\hat{Q}$ which is used for generating the targets y_i for the subsequent C updates to Q.

3.2.4 Bootstrapped Deep Q-Networks

Bootstrapped DQN modifies DQN to approximate a distribution over Q-values via the bootstrap principle. Bootstrapped DQN performs deep exploration which leads to a substantial improvement in learning speed and cumulative performance across most Atari games.

The bootstrap principle is to approximate a distribution with a sample distribution as mentioned in [Efron and Tibshirani, 1993]. The most common way to implement bootstrap is to take as input a dataset D of size N and generate a sampled dataset $\tilde{D}$ also of size N drawn uniformly with replacement from D. The bootstrap serves as a great form of data-based simulation and it also fancies strong convergence guarantees ([Bickel and Freedman, 1981],[Fushiki, 2005])

At the start of each episode, bootstrapped DQN samples a single Q-value function from its approximate posterior. Then, the agent follows the policy which is optimal based on that sample for the duration of the episode. This is a natural adaptation of the Thompson Sampling heuristic to RL that promotes deep exploration.

The algorithm creates $K \in \mathcal{N}$ bootstrapped estimates of the Q-value function in parallel. Each of these value function heads $Q_k(s,a;\theta)$ is trained against its own target network $Q_k(s,a;\theta^-)$ providing a temporally extended estimate of the value uncertainty via TD estimates. A flag is stored $w_1,..,w_K \in \{0,1\}$ which indicates which data belong to which bootstrap head. A bootstrap sample is approximated by selecting $k \in 1,..,K$ uniformly at random and following Q_k for the duration of that episode.

Bootstrapped DQN explores in a similar way as PSRL with the difference being, that bootstrapped DQN directly samples a value function meaning that it does not require further planning steps. Bootstrapped DQN uses random initialization of the network weights as a prior to induce diversity which leads to the agent trying random actions. When a head finds a good state and generalizes to it, some of the heads will learn from it, because of the bootstrapping. Eventually, other heads will either find better states or end up learning the best good states found by the other heads. So, the architecture explores well and once a head achieves the optimal policy, it will eventually lead all the other heads to achieve that same policy.

One special case of bootstrapped DQN where the algorithm would not perform as desired is the following. In bootstrapped DQN, if the rewards are sparse (for example as we will see in the environment presented in Chapter 5.1 , then bootstrapping the targets will not generate any diversity and thus the algorithm will not be able to learn efficiently. To solve that [Osband et al., 2018] provided a solution which is to add a random untrainable prior function to each member of the ensemble. This prior function provides the necessary diversity that is required for the algorithm to work as intended.

Chapter 4

Information Theoretic exploration with Successor Uncertainties

The first section of this Chapter explains and motivates the SU framework and its practical implementation as presented in Janz et al. [2019]. The second part of this Chapter introduces the usefulness of Information Gain (IG) as a way to guide exploration. The third section introduces and explains two algorithms that combine the SU framework with the IG principles. These two algorithms and the experiments that are conducted in the subsequent Chapter are the main contributions of this thesis.

RVFs as discussed in Chapter 3, have proven to be quite successful in practise, leading to algorithms that achieve efficient exploration. Many of the latest exploration methods such as ([Moerland et al., 2017],[O'Donoghue et al., 2017] [Azizzadenesheli and Anandkumar, 2018]) can be interpreted as the combination of RVFs with neural network function approximations. While the neural network function approximations are necessary for scaling problems, they do introduce conceptual difficulties that were not present before in PSRL and tabular RVF methods. To be more precise, getting a posterior distribution over Q-functions is not trivial because we perform off-policy RL, which means that we do not have direct access to the data in the forms of inputs and output values for the Q-function. As noted in the Janz et al. [2019], many methods that combined RVF with neural network function approximations ignored these difficulties, which made them suffer from either one of the following two problems:

1. The distribution over Q-functions ignores the dependencies between function values such as those given by the Bellman equation.

2. The distribution captures these dependencies but at a large computational cost.

SU has managed to solve this problem since it captures the dependencies given by the Bellman equation while also having a low computational cost. SU is a highly scalable, cheap and easy to implement RVF algorithm. SU has managed to outperform its competitors on tabular benchmarks and Atari games.

4.1 Successor Uncertainties Framework

SU uses a probabilistic model over Q-functions that directly takes into account the correlations that exist between Q-function values as induced by the Bellman equation.

In the SU framework, we initially assume that we are given an embedding function $\phi : S \times A \rightarrow \mathscr{R}^d$, such that for all (s,a), $\|\phi(s,a)\|_2 = 1$ and $\phi(s,a) > 0$ elementwise. In addition to that, we also assume a linear Gaussian reward model $r_{t+1} = w^\top \phi(s_t, a_t) + \varepsilon_t$, where $\phi(s_t, a_t)$ is a feature representation of states and actions and $\varepsilon_t \sim \mathscr{N}(\varepsilon|0, \beta)$ is the Gaussian noise.

This previous linear model results into the following linear model for the Q-function:

$$Q^\pi(s_t, a_t) = \mathbb{E}\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_{t+1}\right] = \mathbb{E}\left[\sum_{i=t}^{\infty} \gamma^{i-t} w^\top \phi(s_t, a_t)\right] = w^\top \psi(s_t, a_t), \tag{4.1}$$

where $\psi(s_t, a_t)$ are known in the literature as successor features ([Dayan, 1993],[Barreto et al., 2016]), i.e. the discounted expected occurrences of $\phi(s_t, a_t)$ under policy π.

The first quantity, $\psi(s_t, a_t) = \phi(s_t, a_t) + \gamma \mathbb{E}[\psi(s_{t+1}, a_{t+1})]$ which is an estimator of the successor features can be obtained by applying standard temporal different learning techniques as they were presented in Chapter 2.

The other quantity, w can be estimated by regressing embeddings of observed states $\phi(s_t, a_t)$ onto the corresponding rewards. More specifically by assuming a Gaussian prior over the weights $p(w) = \mathscr{N}(0, \theta I)$ and $p(r_{t+1}|w) = \mathscr{N}(r_{t+1}|w^\top \phi(s,a), \beta)$ as the likelihood, we infer the posterior distribution $p(w|r_{t+1}) = N(w|\mu_w, \Sigma_w)$.

This induces a fully correlated Gaussian posterior for the Q-function given by:

$$\hat{Q}^\pi_{SU} \sim \mathscr{N}(\hat{\Psi}^\pi \mu_w, \hat{\Psi}^\pi \Sigma_w (\hat{\Psi}^\pi)^\top), \tag{4.2}$$

where $\hat{\Psi}^{\pi} = [\hat{\psi}^{\pi}(s,a)]^{\top}_{(s,a)\in(S,A)}$. This is our SU model for the Q-function.

The covariance matrix Σ_w can be updated online according to Bayes Rule:

$$p(w|\mu_{\text{new}}, \Sigma_{\text{new}}) \propto p(r_{t+1}|w)p(w), \tag{4.3}$$

which is equal to:

$$\Sigma_{\text{new}}^{-1} = \Sigma^{-1} + \beta^{-1}\phi(s_t,a_t)\phi(s_t,a_t)^{\top}, \tag{4.4}$$

In the case where the embedding function ϕ is not known beforehand, it can be jointly estimated with the other quantities with the usage of network function approximations.

To be more exact, we want to estimate the Q-function of some given policy π, where $\hat{\phi} : S \times A \to \mathscr{R}^d_+$ is the current estimate of ϕ, (s_t, a_t) is the state action pair observed at time step t and r_{t+1} is the reward attained after selecting action a_t in state s_t. To achieve that, the SU algorithm jointly learns $\hat{\phi}$ and $\hat{\psi}$ by using the relationships between ϕ_t, ψ_t^{π} and $\mathbb{E}[R_{t+1}]$. Thus the problem becomes an optimization problem where the loss is defined below:

$$L(\hat{\phi}, \hat{\psi}, \hat{w}) = \|\hat{\psi}_t - \hat{\phi}_t - \gamma(\hat{\psi}_{t+1})\|_2^2 + (\mu_w \phi_t - r_{t+1})^2 + (\mu_w \psi_t - \gamma\mu_w^{\top}\psi_{t+1} - r_{t+1})^2, \tag{4.5}$$

The object above includes three different loss terms that we want to minimize. The first one is the TD error for the successor features, also known as successor feature loss which enforces ψ to converge to the discounted expected feature occurrence of ϕ. The second term is known as the reward loss and it sets the values of μ_w and ϕ so that the error in the reward prediction is low. The third and last term is the TD error in the estimation of the Q-function and it enforces ψ to have good values for predicting the Q-function values.

The first two losses stem directly from the definition of the SU model. If we assume that there exists a (ReLU) network that achieves zero successor feature and reward loss then the third term would be unnecessary. In practise it was observed that finding these optimal solutions is very difficult so they added the Q value loss which lead to improved performance.

4.1.1 SU Neural network model

In the practical implementation of the SU model, the initial features $\phi(s_t, a_t)$ and the successor features $\psi(s_t, a_t)$ are computed through the usage of a multi-head neural network. In that network each head represents a possible combination of an action value and feature type (ϕ or ψ). The output of each head is multiplied by μ_w so that we can get predictions for the reward and the Q-function. This multi-head neural network is illustrated as shown bellow:

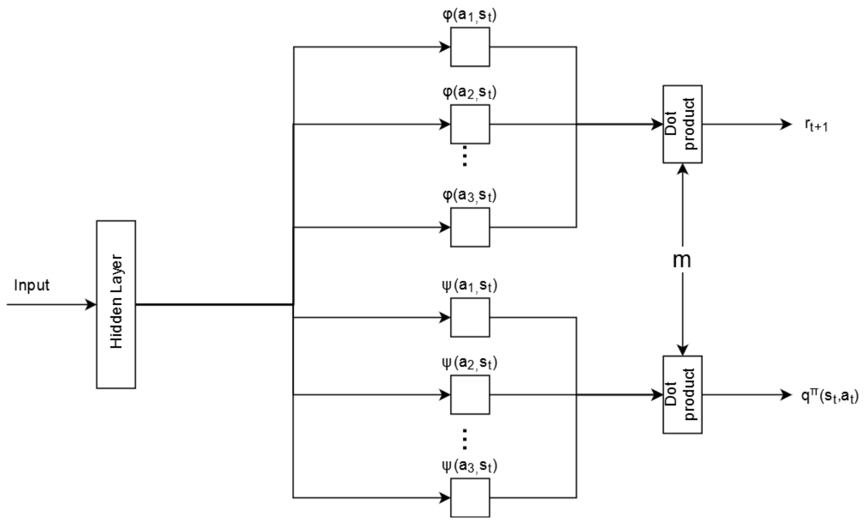

Fig. 4.1 SU model

where in the case that we have a tabular environment, the input for the model is based on the one-hot encoding of the state while in the case of the Atari games the input is given by a convolutional neural network.

4.2 Information-theoretic exploration

In this section, we introduce basic concepts of Information Theory and we explain how incorporating them can lead RL problems to achieve more efficient exploration. More specifically, we introduce the concepts of entropy, Kullback-Leibler (KL) divergence and mutual information (MI). The definitions introduced here are based on Chapter 2 of [Cov] when all the random variables are discrete and on Chapter 5 of [Gray, 2011] for the general random variable case.

Entropy

In information theory, the entropy of a random variable can be interpreted as the average level of "uncertainty" "information" or "randomness" that can be associated with the variable's possible outcomes. Information entropy as a concept was introduced by Claude Shannon in [Shannon, 1948].

Given a discrete random variable X, that can take as possible values $x \in \mathscr{X}$ with a probability mass function $p(x) = Pr(X = x)$, the entropy of X is denoted as $H(X)$ and it is defined as shown below:

$$H(X) = -\sum_{x \in \mathscr{X}} p(x) \log p(x) \tag{4.6}$$

where Σ denotes the sum over all the variable's possible outcomes and log is the logarithm with a base that varies between different applications. To be more precise, the base of 2 which is the most common base gives the unit of bits, a base of e gives the "natural units" nats while a base of 10 gives a unit called "dits" or "hartleys". An equivalent definition of entropy is the expected value of the random variable $\log \frac{1}{p(X)}$ as shown in the following equation.

$$H(X) = \mathbf{E}_p \log \frac{1}{p(X)} \tag{4.7}$$

where this definition is related to the definition of entropy in thermodynamics.

The notion of entropy can extend to two variables with the definition of the joint entropy. The joint entropy $H(X,Y)$ of a pair of discrete random variables (X,Y) is defined with a joint distribution $p(x,y)$ in the same way as the entropy of a single random variable as shown below:

$$H(X,Y) = -\sum_{x \in \mathscr{X}} \sum_{y \in \mathscr{Y}} p(x,y) \log p(x,y) \tag{4.8}$$

The conditional entropy of a random variable given another random variable is defined as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable.

$$H(Y|X) = -\sum_{x\in\mathscr{X}}\sum_{y\in\mathscr{Y}} p(x,y)\log p(x|y) \tag{4.9}$$

It has been proven that the entropy of two random variables is equal to the entropy of the one plus the conditional entropy of the other as shown:

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y), \tag{4.10}$$

Kullback-Leibler Divergence

The KL divergence (also known as relative entropy) is a measure of dissimilarity between two distributions. For two discrete probability distributions with probability mass function $p(x)$ and $q(x)$ the KL divergence is defined as:

$$D(p||q) = \sum_{x\in\mathscr{X}} p(x)\log\frac{p(x)}{q(x)}, \tag{4.11}$$

Some important properties of the KL divergence are:

1. KL divergence is non-negative, and it achieves equality if and only if the two distributions are the same $p(x) = q(x)$
2. KL divergence is non-symmetric which means that $D(p||q) \neq D(q||p)$
3. The KL divergence is well-defined for continuous distributions, and thus it is invariant under parameter transformations.

Information Gain

Information gain (IG) or mutual information (MI) is a measure of dependence between two random variables. To be more exact IG quantifies the amount of information that one random variable contains about another random variable. It can also be explained as the reduction in the uncertainty of one random variable due to the knowledge of the other.

If there are two random variables X and Y with a joint probability mass function $p(x,y)$ and marginal probability mass functions $p(x)$ and $p(y)$ respectively, then the IG $I(X;Y)$ is defined as the relative entropy between the joint distribution and the product distribution $p(x)p(y)$ as shown:

$$I(X;Y) = \sum_{x\in\mathscr{X}} \sum_{y\in\mathscr{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}, \tag{4.12}$$

The aforementioned equation can be equivalently expressed as the difference between the marginal entropy and the conditional entropy as shown:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X), \tag{4.13}$$

4.2.1 Motivation for Information Gain

As it was stated in Chapter 3, the reason that we want to explore in the first place is so that we can accumulate more information. Whenever we explore, we forego immediate rewards so that we can learn more about the environment and use this knowledge to get more rewards in the future. Thus, the value of information can be perceived as a secondary auxiliary reward that motivates us to visit more uncertain states. The IG is higher in the states that we are most uncertain, thus it makes sense for us to use it as a quantity that guides us in terms of which action to pick next so that we can explore. In this thesis, we leverage this idea in combination with the original SU framework in hopes of improving the SU model by creating an algorithm that explores more efficiently.

4.3 Information Theoretic approach to Successor Uncertainties

In this section, we present the main contribution of this thesis, which is the creation of two information-theoretic algorithms that extend the original SU model. The two new algorithms that we present, leverage the information gain in two different ways with the purpose of improving the exploration aspect of the SU algorithm. In these two algorithms we extend the original SU algorithm by also computing an auxiliary reward which accounts for the IG between states. Our hopes are that in the limit, these algorithms should both converge to the normal Q function as per SU but before they do they will give an exploration bonus to the state-action pairs that we have the most uncertainty about. We name these methods SU-IG1 and SU-IG2. The main difference of these two methods is in the way that they compute the IG.

In both algorithms, at every time step we sample a state action pair (s,a) from the replay buffer and we compute two Q functions. We compute Q using Expected Sarsa as it is computed in the original SU model and we also compute the expected cumulative discounted information gain function Q_{IG}. The computation of this function is treated as solving a second independent MDP.

To be more precise, whenever we want to select an action at any given state we have two things to estimate: the Q function for the reward and the Q_{IG} function for the auxiliary reward which accounts for the information gain.

4.3.1 Successor Uncertainties - Information Gain 1st Algorithm

In the first algorithm which we call SU-IG1, we compute the IG quantity as the reduction of the uncertainty in the Gaussian distribution of the Q-function.

The expected cumulative discounted information gain Q_{IG} is computed as shown below:

$$Q^{\pi}_{IG}(s_t,a_t) = \mathbb{E}\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_{IG}(s,a)\right], \tag{4.14}$$

where $r_{IG}(s,a)$ is defined as an auxiliary reward for the IG accumulated when the state-action pair (s,a) is used to update the Q-function, as shown below:

$$r_{IG}(s,a) := IG(Q;s,a), \tag{4.15}$$

Based on the definition of IG that we gave in equation 4.13, we compute that $IG(Q;s,a) = H(Q) - H(Q|s,a)$, which stands for the information gained about Q when the state-action pair (s,a) is selected to update the Q-function distribution from $\mathcal{N}(m,S)$ to $\mathcal{N}(m,S')$ where m and S are the mean and the covariance matrix of the Q-function distribution respectively.

Thus, $r_{IG}(s,a)$ can be written as a function that depends only on S and S' which can be defined as $IG(S,S')$ as shown in the following equation:

$$IG(S,S') = \frac{1}{2}log(1+(\phi')^\top S(\phi')) \tag{4.16}$$

where ϕ' are the information gain features and they are computed through a multi-head neural network as shown in figure 4.2. The proof that 4.16 holds is shown in the first section of the Appendix.

The way we select the optimal action a^* in any state s is through the following equation:

$$a^* = arg\max_a(\mathrm{Q} + \lambda Q_{IG}) \tag{4.17}$$

where λ determines how much we want to factor the information gain of the successor states.

The loss function is computed similarly to 4.5. The only difference here is the addition of one more term that accounts for the Q_{IG} loss which is equal to $(Q_{IG} - \gamma Q_{IG} - r_{IG})^2$.

Neural Network Model

In the practical implementation of the SU-IG1 algorithm, the initial features $\phi(s_t,a_t)$ and the successor features $\psi(s_t,a_t)$ are computed through the usage of a multi-head neural network similarly to how they were computed in the SU model. The only additional quantity that needs to be computed is the $\phi'(s_t,a_t)$ which is computed by adding an additional hidden layer that gets as input the output of the initial hidden layer. Thus based on this the multi-head neural network for the SU-IG1 is illustrated below:

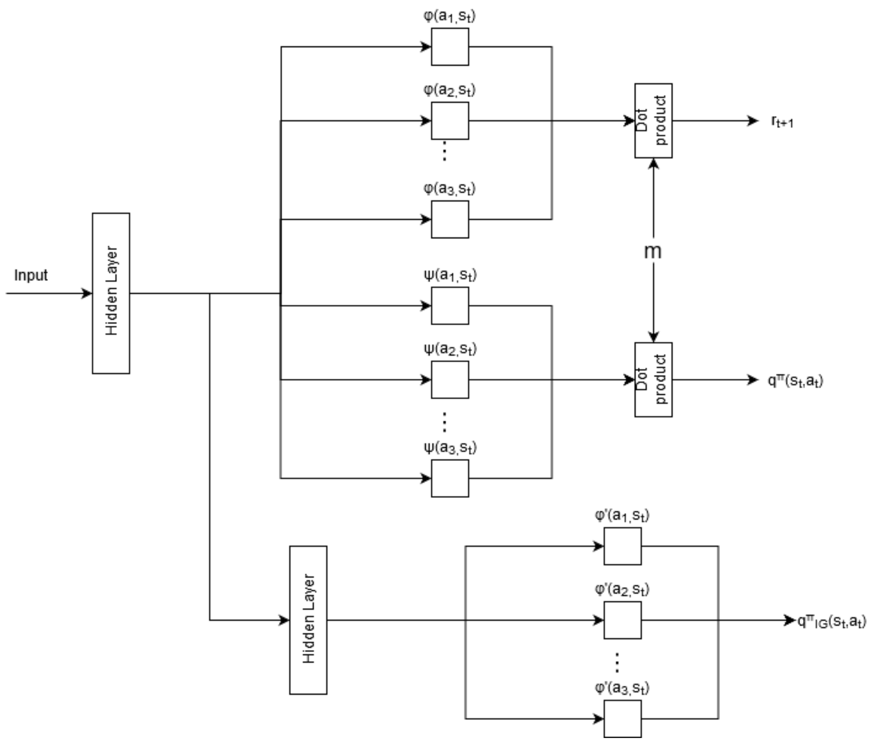

Fig. 4.2 SU-IG1 model

Similarly to the SU model, SU-IG1 learns the neural network parameters and m by optimizing the loss function.

4.3.2 Successor Uncertainties - Information Gain 2nd Algorithm

In the second algorithm, which we name SU-IG2, Q_{IG} is computed similarly to equation 4.14 with the difference being on the computation of the IG.

In the SU-IG1 algorithm, the IG is computed as the reduction of the uncertainty in the Gaussian distribution of the Q-function, while in SU-IG2 the IG is computed as the reduction of the uncertainty in the Bernoulli distribution of the optimal action.

On the SU-IG2 algorithm, $r_{IG}(s,a)$ is defined as an auxiliary reward and it is computed based on the probability of an action at a specific state being optimal as shown below:

$$r_{IG}(s,a) := IG(a^{opt};s,a), \tag{4.18}$$

Similarly to SU-IG1, based on the definition of IG that we gave in equation 4.13 we compute the IG quantity and it is equal to $IG(a^{opt}; s, a) = H(a^{opt}|s) - H(a^{opt}|s, a)$.

Thus, $r_{IG}(s, a)$ is written as shown in the following equation:

$$r_{IG}(s, a) = H(a^{opt}|s) - H(a^{opt}|s, a), \tag{4.19}$$

where the first term is constant with respect to a. If we focus on two actions per state, we compute the probability distribution for the first entropy $H(a^{opt}|s)$ as shown:

$$p(a^{opt}|s) = Ber(a|p(Q(s, a_1) > Q(s, a_2))), \tag{4.20}$$

with the Bernoulli distribution being computed by the following equation:

$$p(Q(s, a_1) > Q(s, a_2)) = \Psi\left(\frac{m_1 - m_2}{\sqrt{S_{11} + S_{22} - S_{12} - S_{21}}}\right) \tag{4.21}$$

where m and S are the mean and covariance matrix of the Q-function posterior distribution, while $\Psi(z)$ is the Gaussian Cumulative distribution function (CDF). The probability distribution $p(a^{\text{opt}}|s, a)$ is computed analogously to Equation 4.20, using the updated Q-function posterior, which is computed for taking action a in state s.

The rest of the algorithm works in the same way as SU-IG1.

Chapter 5

Experiments and Results

In this Chapter, we conduct experiments on the algorithms that we developed in Chapter 4 using the binary tree MDP environment where we measure their performance in comparison to methods that combine Q-function modeling with neural network function approximation.

5.1 Environment

The environment that we use to evaluate the performance of the algorithms is the binary tree MDP of size L, as it is shown in the following figure:

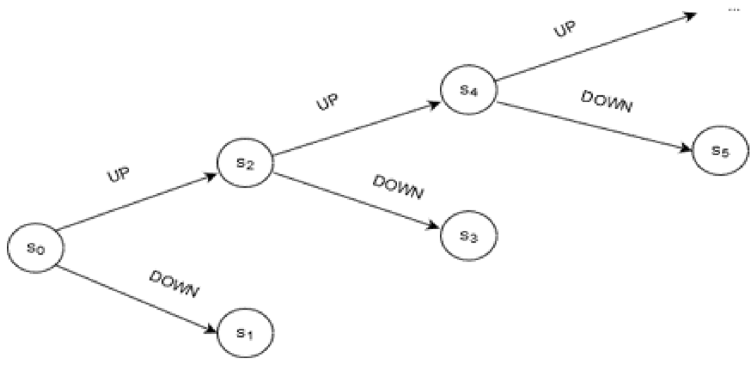

Fig. 5.1 Binary Tree MDP

This environment consists of $2L+1$ states S that are one-hot encoded. The actions which are two $A = a_1, a_2$ are mapped to movements $\{\text{UP}$, $\text{DOWN}\}$ according to a random mapping drawn independently for each state. The agent receives a reward of one when he reaches state S_{2L} and zero otherwise. The states with odd indices and S_{2L} are terminal states. For an agent to solve the MDP and find the reward, it needs to make L consecutive correct decisions.

5.2 Experiments

In this part of the thesis, we compare the algorithms that we created with other similar methods from the RL literature. The methods that we are comparing with are methods that combine the Q function models with neural network function approximation. This category has two popular subclasses, methods that rely on Bayesian linear Q-function models such as SU Janz et al. [2019], Bayesian Deep Q-Networks(BDQN) [Azizzadenesheli and Anandkumar, 2018], Uncertainty Bellman Equation (UBE) [O'Donoghue et al., 2017] and methods that that are based on bootstrapping such as Bootstrapped DQN [Osband et al., 2016]. For this comparison to be fair, we use the same hyperparameters for all the algorithms without tuning them for any specific method.

In all the experiments that are conducted in this Chapter, we compute for each algorithm the median number of episodes that are required to learn the optimal policy on the tree MDP. We use 5 seeds for each experiment and 5000 as the number of episodes for each seed. Blue points indicate that all 5 seeds succeeded within 5000 episodes,orange indicate that only some of the runs succeeded while red indicates that all runs failed. Dashed lines correspond to the median number of episodes that a uniform exploration policy requires. Lastly, the hyperparameters that we used for these experiments are provided in the Appendix B. It is important to note that in the case of Bootstrapped DQN, we use a prior as it was discussed in 3.2.4.

Figures 5.2 and 5.3 show the empirical performance of the UBE and BDQN algorithms in comparison to an agent that uses a uniform exploration policy in the binary tree MDP environment.

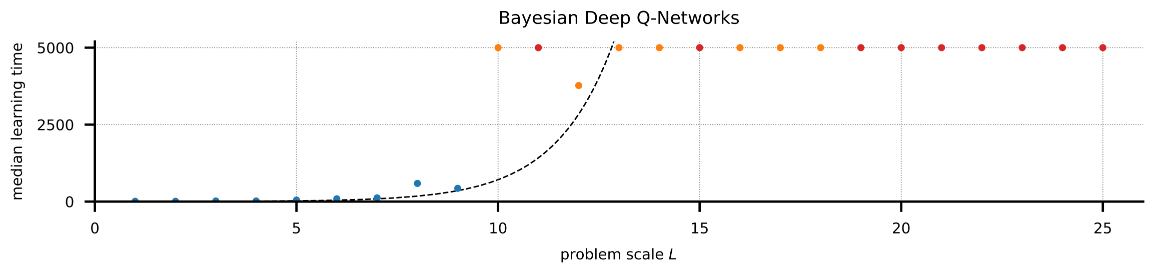

Fig. 5.2 Median number of episodes required for BDQN algorithm to find the optimal policy

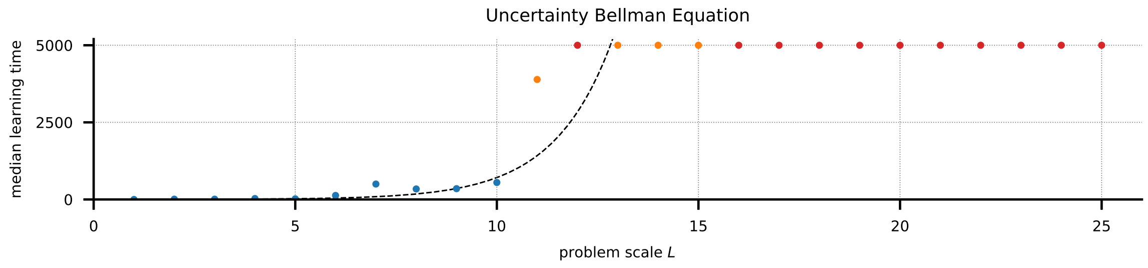

Fig. 5.3 Median number of episodes required for UBE algorithm to find the optimal policy

As it can be observed from these two figures both BDQN and UBE fail to outperform the uniform exploration policy. For UBE the reason for this low performance is explained in [Janz et al., 2019], where it states that any algorithm that combines factorised symmetric distributions with posterior sampling such as UBE, can solve the binary tree MDP with probability of at most 2^{-L} per episode. BDQN seems to be suffering from similar problems.

Figures 5.4, 5.5, 5.6 and 5.7 show the empirical performance of SU, SU-IG1, SU-IG2 and Bootstrapped DQN algorithms in comparison to an agent that uses a uniform exploration policy in the binary tree MDP environment.

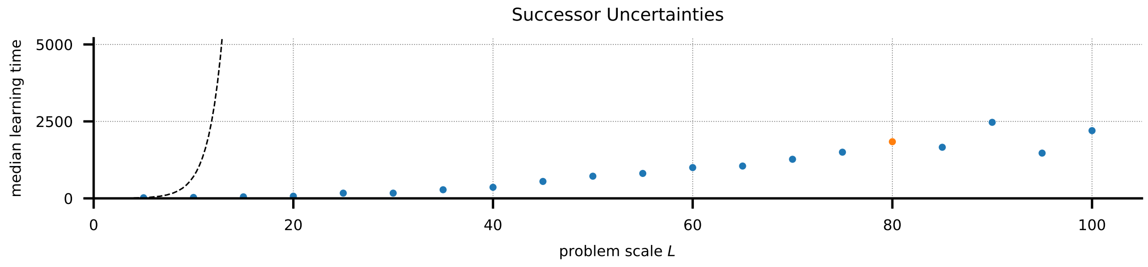

Fig. 5.4 Median number of episodes required for SU algorithm to find the optimal policy

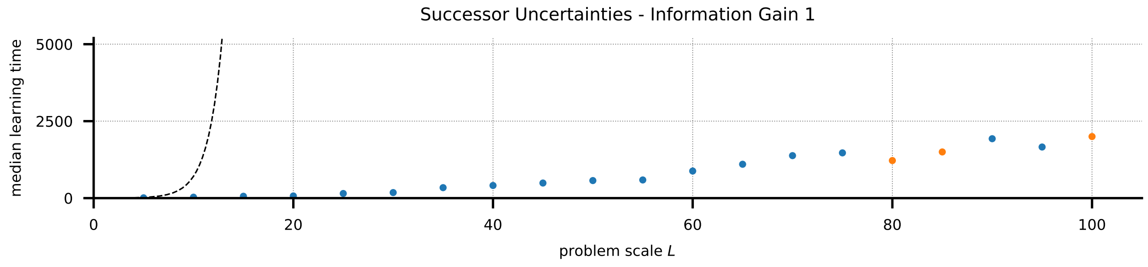

Fig. 5.5 Median number of episodes required for SU-IG 1 algorithm to find the optimal policy

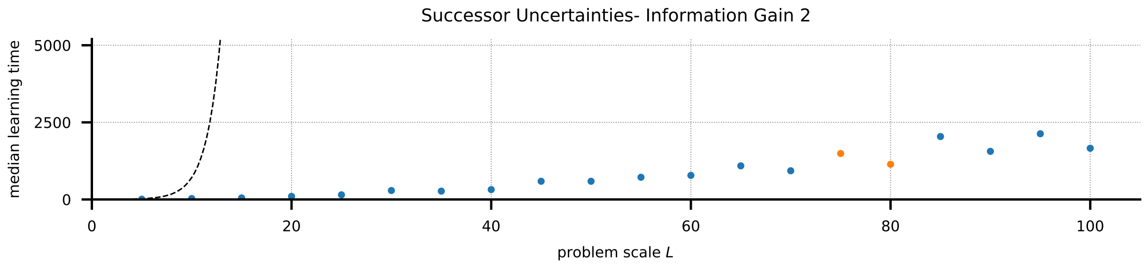

Fig. 5.6 Median number of episodes required for SU-IG 2 algorithm to find the optimal policy

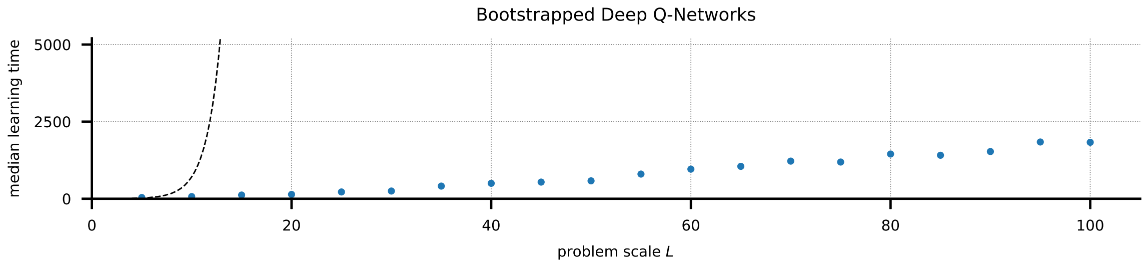

Fig. 5.7 Median number of episodes required for Bootstrapped DQN algorithm to find the optimal policy

As it can be observed from these four figures, all four algorithms massively outperform the BDQN and UBE algorithms by consistently managing to find the reward even when the environment has a size of $L = 100$. All four methods manage to succeed on large scale binary trees with a very sparse reward structure and randomised action effects. However as it is noted in [Janz et al., 2019], for Bootstrapped DQN to achieve the same performance it requires significantly more computations. Both SU-IG1 and SU-IG2 need more computations because they do the same thing as the SU algorithm with the addition of computing an additional Q-function that accounts for the IG. Thus, the best suited algorithm for this particular environment is SU. Between SU-IG1 and SU-IG2 there are very minor differences in performance which can be attributed to the randomization of the seeds.

Chapter 6

Conclusions

In this thesis we started by explaining the framework and the core concepts of RL where we introduced some of the most well-known algorithms of the RL literature such as SARSA and Q-learning. After that, we explained the exploration/exploitation trade-off dilemma in the context of multi-armed bandits and we motivated the necessity for deep exploration in real-world complex problems. Additionally, we explained and motivated various algorithms that perform deep exploration such as SU and Bootstrapped DQN.

6.1 Main contributions

The main contribution of this thesis is the development of two algorithms that leverage the IG with the purpose of improving the exploration aspect of the SU algorithm and perform deep exploration. Both algorithms treat IG as an auxiliary reward and they create a separate Q-function Q_{IG} that promotes exploration. The difference in these two algorithms is in the computation of the IG. In SU-IG1, the IG is computed as the reduction of the uncertainty in the Gaussian distribution of the Q-function, while in SU-IG2 the IG is computed as the reduction of the uncertainty in the Bernoulli distribution of the optimal action. Experiments were conducted on these two algorithms that show both of the algorithms succeeding in solving the binary tree MDP environment with the algorithm's performances being similar to that of the SU algorithm. According to these observations, the addition of the IG did not lead to substantial improvements over the SU algorithm's performance, however the experiments were limited to a particular environment setting. Thus, there is still room for research that can be conducted on the utilization of the IG in the SU framework and beyond.

6.2 Future work

Some interesting ideas for future work are the following:

First of all, it would be interesting to create environments that fit more the information-theoretic approach of this thesis. To be more precise, an idea would be, to carefully construct environments that require exploration strategies that take advantage of the information gain. Deploying the proposed algorithms of this thesis in such an environment could provide interesting results.

Another promising direction for research, is to scale the algorithm to more complex problems, such as ones that have more actions. As we proved in Chapter 4, for a two action setting, we can find an analytic solution that computes the probability of an action being optimal by using a bivariate Gaussian. That is not the case with multiple action, because in that setting the distribution becomes multivariate and finding the probability that one action is better than all the others does not have an analytic solution. In that case, approximations could be found that approximate the probability of an action being optimal.

Another interesting thing to consider is proving theoretically whether it makes sense to discount IG when we compute the Q_{IG} function. Discounting in the traditional RL framework has both practical and theoretical usefulness. It has practical usefulness because it stops the Q-function values from exploding and it has theoretical usefulness because it makes sense to prefer rewards in the present rather than rewards in the future. Similarly to that, Q_{IG} can explode without discounting and because IG acts as an auxiliary reward it also intuitively makes sense to prefer getting information earlier rather than later because you will be able to use the knowledge earlier leading to higher rewards. Thus, finding if its worth discounting IG could be an interesting research avenue for the future.

References

P. Auer, N. Cesa-Bianchi, P. Fischer, and L. Informatik. Finite-time analysis of the multi-armed bandit problem, 2000.

K. Azizzadenesheli and A. Anandkumar. Efficient exploration through bayesian deep q-networks, 2018.

A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. van Hasselt, and D. Silver. Successor features for transfer in reinforcement learning, 2016.

R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954. URL https://projecteuclid.org:443/euclid.bams/1183519147.

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition, 1996. ISBN 1886529108.

P. J. Bickel and D. A. Freedman. Some asymptotic theory for the bootstrap. *Ann. Statist.*, 9 (6):1196–1217, 11 1981. doi: 10.1214/aos/1176345637. URL https://doi.org/10.1214/aos/1176345637.

M. Campbell, A. J. Hoane, and F. hsiung Hsu. Deep blue. *Artif. Intell.*, 134:57–83, 2002.

P. Dayan. Improving generalization for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, 1993.

B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Number 57 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, Boca Raton, Florida, USA, 1993.

T. Fushiki. Bootstrap prediction and bayesian prediction under misspecified models. 2005.

R. M. Gray. *Entropy and Information Theory*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 9781441979698.

W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. ISSN 01621459. URL http://www.jstor.org/stable/2282952.

D. Janz, J. Hron, J. M. Hernández-Lobato, K. Hofmann, and S. Tschiatschek. Successor uncertainties: Exploration and uncertainty in temporal difference learning. *CoRR*, abs/1810.06530, 2019. URL http://arxiv.org/abs/1810.06530.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey, 1996.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1): 79–86, 03 1951. doi: 10.1214/aoms/1177729694. URL https://doi.org/10.1214/aoms/1177729694.

T. L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985. URL http://www.cs.utexas.edu/~shivaram.

H. Lee, R. Grosse, R. Ranganath, and A. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, Proceedings of the 26th International Conference On Machine Learning, ICML 2009, pages 609–616, 2009. ISBN 9781605585161. 26th International Conference On Machine Learning, ICML 2009 ; Conference date: 14-06-2009 Through 18-06-2009.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. 2013. URL http://arxiv.org/abs/1312.5602. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

T. M. Moerland, J. Broekens, and C. M. Jonker. Efficient exploration with double uncertain value networks, 2017.

B. O'Donoghue, I. Osband, R. Munos, and V. Mnih. The uncertainty bellman equation and exploration, 2017.

I. Osband, D. Russo, and B. V. Roy. (more) efficient reinforcement learning via posterior sampling, 2013.

I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. Deep exploration via bootstrapped dqn, 2016.

I. Osband, B. V. Roy, D. Russo, and Z. Wen. Deep exploration via randomized value functions, 2017.

I. Osband, J. Aslanides, and A. Cassirer. Randomized prior functions for deep reinforcement learning, 2018.

G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

D. Russo and B. V. Roy. Learning to optimize via posterior sampling, 2013.

C. E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3):379–423, 1948. URL http://dblp.uni-trier.de/db/journals/bstj/bstj27.html#Shannon48.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017. URL http://dx.doi.org/10.1038/nature24270.

M. Strens. A bayesian framework for reinforcement learning. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950. ICML, 2000.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994. doi: 10.1162/neco.1994.6.2.215.

O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575, 11 2019. doi: 10.1038/s41586-019-1724-z.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

Appendix A

Computation of the IG for the SU-IG1 algorithm

In this part of the Appendix we prove that the equation 4.16 holds. We start this proof by writing some definitions. The differential entropy of a Gaussian distribution $N(\mu,\Sigma)$ is defined as:

$$H(N(\mu,\Sigma)) = \frac{1}{2}\log(2\pi e\Sigma), \tag{A.1}$$

Based on 4.13 we get the following equation which is equal to the information we gain about the Q function when we use a state-action pair (s,a) to update its distribution from $N(\mu,\Sigma)$ to $N(\mu,\Sigma')$:

$$IG(\Sigma,\Sigma') = H(N(\mu,\Sigma)) - H(N(\mu,\Sigma')), \tag{A.2}$$

By multiplying both sides by 2 and then using the logarithmic properties we get:

$$2IG(\Sigma,\Sigma') = \log\left(\frac{\Sigma}{\Sigma'}\right), \tag{A.3}$$

Then we replace the covariance matrix Σ and the updated covariance matrix Σ' with the precision matrices P^{-1} and $P + \phi\phi^{\top}$ respectively based of the definition of the precision matrix $\Sigma = P^{-1}$ and we get the following equation:

$$2IG(\Sigma,\Sigma') = \log\left(\frac{P^{-1}}{(P+\phi'\phi'^{\top})^{-1}}\right), \tag{A.4}$$

where ϕ' are the information gain features which are computed through the multi-head neural network as it shown in 4.2. By inverting we get:

$$2IG(\Sigma,\Sigma') = \log\left(\frac{P+\phi'\phi'^{\top}}{P}\right), \tag{A.5}$$

The numerator of the fraction $P+\phi'\phi'^{\top}$ can be simplified by using linear algebra properties:

$$P+\phi'\phi'^{\top} = P(I+P^{-1}\phi'\phi'^{\top}) = P(I+\phi'^{\top}\Sigma\phi') \tag{A.6}$$

which simplifies equation A.5 to the desired equation:

$$IG(\Sigma,\Sigma') = \frac{1}{2}log(1+(\phi')^{\top}\Sigma(\phi')) \tag{A.7}$$

Appendix B

Hyperparameter selection

The hyperparameters that are used for the SU and SU-IG experiments in Chapter 5 are shown in the following Table:

Number of episodes	5000
Prior variance θ	10^{-4}
Likelihood variance β	10^{-3}
Feature size	50
Gradient steps per episode	10
Covariance matrix decay factor ζ	1

Table B.1 Hyperparameters for SU and SU-IG experiments

The hyperparameters that are used for the Bootstrapped-DQN experiments in Chapter 5 are shown in the following Table:

Number of episodes	5000
Ensemble size K	10
Bootstrap probability	0.5
Gradient steps per episode	10

Table B.2 Hyperparameters for Bootstrapped DQN experiments

Hyperparameter values do not affect UBE and BDQN performance because both algorithms perform uniformly random exploration.