TECHNISCHE UNIVERSITÄT BERLIN

Linear approaches to a stochastic mechanical control problem

by

Mahmoud Mabrouk

Supervisor: Prof. Dr. Manfred Opper

A thesis submitted in partial fulfillment for the
degree of Bachelor of science

in the

Fakultät Elektrotechnik und Informatik

Fachgebiet Künstliche Intelligenz

March 2010

Die selbständige und eigenhändige Ausfertigung versichert an Eides statt.

Berlin den

"Our greatest glory is not in never falling, but in rising every time we fall."

Conficus

Zusammenfassung

Die Arbeit beschäftigt sich mit einem neuen Formulierung des "Optimal Control Problem", in anderen Wörtern, wie ein Agent ein System kontrollieren kann, um ein optimales Verhältnis zu erhalten. Wir beschreiben die State-of-the-art Lösungen für das Problem und ihre Theoretischen Grundlagen. Dann diskutieren wir die neue Lösung entwickelt von Prof. Todorov, in der die Bellman Gleichung für diskrete Systeme linearisiert wird. Hierbei beschreibt die Bellman Gleichung die erwarteten Kosten eines Zustandes. Um die Lösung zu finden, vertauschen wir die diskreten Werte der Aktionen mit ihren Wahrscheinlichkeitsverteilung und können dadurch mit Hilfe mathematischer Verfahren eine lineare "desirability" Gleichung herleiten. Aus dieser Gleichung werden dann zwei Algorithmen abgeleitet: Z-Iteration und Z-Learning.

Z-Iteration ist ein offline Algorithmus, der alle Informationen über die Umgebung braucht. Andererseits ist Z-Learning ein online Algorithmus, der mit "Sampling" (Stichprobenverfahren), also durch direkte Erfahrung lernt. Wir vergleichen die neuen Algorithmen mit den State-of-the-art Lösungen in einer stochastichen Simulation, in der ein Wagen mit minimalen Schritten und Energie zu einem Ziel gebracht werden soll.

Im Vergleich zur alten Lösungen zeigen die beiden Algorithmen schnellere Laufzeit und Konvergenz, und haben eine niedrigere Platzkomplexität. Allerdings haben sie auch einige Nachteile: Zum einen benötigen sie viele Informationen über die Umgebung, zum anderen müssen die Kosten in dem System ein KL divergenz sein. Außerdem lassen sich einige Systeme wegen andere Einschränkungen nicht lösen. Wir schließen daraus, dass das neue Verfahren nur in einigen Fällen ein gutes Potenzial hat.

Abstract

This thesis discusses a new method to linearize the Bellman equation for a special class of problems and tests its resulting algorithm with the state-of-the-art solutions. Reinforcement learning and Dynamic programming are presented and the state-of-the-art algorithms are discussed. The new framework and its mathematical foundations are then introduced. It results in a linear solution to the optimal action both in discrete and continuous domains, and in a new formulation of the cost-to-go function which exchanges the exhaustive search over actions with a linear solution. Later, an online and an offline algorithm are developed from the last results. They are tested against Policy Iteration and Q-Learning in a stochastic variant of the Mountain car problem. Results show a great improvement brought by the new algorithms both in speed and efficiency. Last, the limitations of the new framework are discussed.

Acknowledgements

I would like to take this opportunity to thank Prof. Manfred Opper and Dr. Andreas Ruttor for their assistance, support and great guidance.
Many thanks cousin Mudar and his wife Diana for their support and all the good times we had.
A big thank to Mehdi, Anwer, ElBiz and all my friends in Tunisia, Berlin and Shanghai for making my life joyful.
A special thank to Amir for all his help and support.
Last, all my gratitude to my mother and father for their love, care and support, for their wise advices, and for making me the man I am.

Contents

Chapter 1

Introduction

1.1 The problem

The optimal control theory have wide application in different areas, we find it in economics, engineering and even in biology. All these seemingly unrelated applications share one property: a system that is influenceable by a set of actions. The goal of the control theory is to achieve optimal system behavior though the choice of action. But how to define an optimal behavior? It depends from the definition of the system; it can mean for example the best way to maximize the performance of a portfolio through the choice of investments, or the best way to allocate resources in a computer to minimize the processes waiting time...
The optimal control theory also plays a major role in some sub-areas of machine learning as reinforcement learning, a method which involves an agent learning a certain behavior through trial and error in a dynamic environment [1]. Reinforcement learning in itself encompasses different algorithm that can be used for the choice of the optimal actions. Most of these algorithms are very general and not task specific, which makes them most of the time not practical due to their high complexity. The main issue is the non linearity of the main equation used in all this problems: The Bellman equation (also called the optimal cost-to-go equation). In this thesis we will analyze a new method created by Todorov and published in [2] to linearize the bellman equation and offer an analytical solution to the control problem in a subclass of Markov Decision Processes (MDP). This new framework offers multiple new algorithms that are less complex than the state-of-the-art solutions, thus faster and therefore more adequate for practical problems. In this

thesis we will analyze this new framework and compare the performances of its resulting algorithm with the state-of-the-art solutions.

1.2 Structure of this Thesis

Reinforcement learning and some background knowledge will be presented in chapter 2. Chapter 3 will discuss the basic algorithms in reinforcement learning and their classifications mentioning their specific limitations. The reader familiar with the subject may skip these two chapters and continue reading in chapter 4. Therein Todorov's new framework is presented as well from a theoretical point as from a practical one. An experiment is set in chapter 5 to compare the new algorithms with Policy Iteration and Q-Learning the result are discussed in chapter 6. Lastly, chapter 7 summarizes and discusses the possibilities and limitations of this new approach.

Chapter 2

The Problem

2.1 Markov Decision Process

Markov Decision Process is a mathematical framework created by Andrey Markov to model systems that have stochastic behavior but can be influenced through the actions made by the a decision maker. The main component of MDP are states, each of these describes a situation in the system. Each state is defined by a set of properties or observations in the system, for example in a gambling game we can consider the outcome of the game as the inspected property and set only two states *winning* and *loosing*. But we can also use a more precise approach by adopting the amount that the player processes as the property, thus creating a continuous set of state $s \in \{x \text{ Euro} | x \in [0; 100]\}$.

A MDP is a discrete time process, which means that in each time step the process moves stochastically from a state s to s'. Through the choice of actions, the decision maker can influence the probability of the transition to a state s'. Formally each action is defined by its transition probabilities $p(s'|s, a)$, that is the probability of moving to a state s' in the next step while being in a state s and choosing an action a. In our gambling example, action could be *bet big* and *bet small* describing betting a big amount of money in the game in a round or betting a smaller amount. When defining states we must pay attention that their properties don't divulge information that the agent mustn't know as for example the cards in the deck in a gambling game. An important characteristic that the process must have is the Markov property: the transition probability to a next state must only depend on the present state, that is $p(s_{t+1}|s_t, s_{t-1}, s_{t-2}...s_0) = p(s_{t+1}|s_t)$. In our

example it means that the chance to win (to move to the state *winning*) must only depend on the present situation and not by the past wins or losses.
The last component of the MDP is the rewards: with each transition the decision maker gets a numerical reward (or a cost) as a feedback from the MDP. The reward is formalized by the function $R_a(s, s')$ that describes the immediate reward received after a transition from a state s to a state s' using an action a with a transition probability $p(s'|s, a)$. In our example the transition to the state *winning* could generate a reward of 5 and the transition to the state *loosing* could generate a reward of -5. The reward is used to define how to solve the decision making problem: which action are optimal in a certain state s. The goal is to find an optimal policy π that maps the actions to the states of the MDP. The rewards provides the motivation to the decision makers, its goal is to maximize the rewards in the long run. More formally, a policy π must be chosen to maximize the sum cumulative function of the rewards, we could express the accumulation of immediate rewards over an infinite horizon (we assume that the MDP never ends) as: $\sum_{t=0}^{\infty} \gamma^t R_{at}(s_t, s_{t+1})$ where $0 \leq \gamma \leq 1$ is the discount rate, which is used to determine how much weight does the decision maker gives to the future rewards, a large γ means that the decision maker prioritize rewards in the long run, a small one means that the near future's rewards are more important.

2.2 The Reinforcement Learning Problem

Reinforcement Learning is a machine learning method that was inspired by psychology: the agent learns a behavior through rewards without supervision, it explores different way for solving the problem while trying to maximize a numerical reward. The goal of the RL is learning a policy (a map from states to actions) to maximize the reward in a long term. We will describe in the next sections the foundations of RL and their applications.

2.2.1 The Agent and the Environment

The RL problem is modeled in two parts: the agent and the environment. The agent constitutes the learner and the decision maker interacting with the environment. At the beginning the agent is in a certain state s given by the environment and offered a set of actions a to choose from at each step. The agent then, through the use of its policy, chooses an action affecting the environment which moves to a new state s' and hand back an immediate reward as a result. More formally, the environment in RL problems is typically formulated as a finite MDP $(S, A, P(.|.), R(.|.))$, where S presents the finite set of states in the problem and A the finite set of actions. $R(s', s)$ constitute the immediate reward received after a transition from state s to a state s'. Lastly $P(s'|s, a)$ is the probability that an action a in a state s will lead to the state s'.
The main goal of the agent is to learn an optimal policy that maps the states to the actions, so he can choose an action at each state that enables it to maximize the reward (or minimize the costs). The policy is called π where for each state s $\pi(s) = a$ exists denoting which action to choose.

2.2.2 Rewards

How to define a goal in a learning algorithm? How to make the agent learn to play Backgammon or park a car?
We could see the process of learning to play Backgammon (or more precisely learning to win a Backgammon game) as learning which moves should be done at each situation to end as a winner, analogically parking a car could be represented as controlling the car in a way to reach the states where the position of the car is in the same as the position of the parking lot and the velocity of the car is low. We will call this states goal states.
Reinforcement Learning uses rewards or costs to make the agent learn some behavior: We set numerical rewards in the desired states (for example in goal states) and then let the agent try to maximize the accumulated reward in the long run.
Choosing a reward can be sometimes tricky, most of the time giving the goal states a big reward can be enough to favor this state over others but that isn't always all what we need. For example let us consider an agent walking in a virtual labyrinth, if we want it to learn to reach the exit, then we could give the goal state a positive reward and the rest of the states no reward. But if the goal is that the agent

reaches to the exist rapidly, then we should consider charging the agent for each step it takes without reaching the goal, so we give all states (except the terminal one) a negative reward.

2.3 Bellman Optimality Equation

The Bellman equation constitute a necessary condition for optimality: it defines the value function which in itself represents how good is a certain state, in other words how much rewards can we expect if we start from this state and follow a certain policy for choosing the actions afterwards. Formally the value function is

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^{k} r_{t+k+1} | s_t = s\right\}$$

π denotes the followed policy, E_{π} denotes the expected value given that the agent follows π afterwards, γ is the discount rate used to determine how much weight is given to future rewards and r_t is the reward at time t.
Our goal is to find the best way to solve the problem through maximization of the reward (or minimization of the costs) in the long run . We call the policy that enables this an optimal policy. An optimal policy is a policy that has a better expected return than all other policies, we denote it π^*. The optimal value function is then $V^*(s) = \max_{\pi} V^{\pi}(s)$ for all $s \in S$. Following the optimal policy means choosing in each step the action that maximizes the return, we can therefore dismiss π^* from the optimal policy equation and use $\max_a$ instead:

$$V^*(s) = \max_a E\left\{\sum_{k=0}^{\infty}\gamma^{k} r_{t+k+1} | s_t = s, a_t = a\right\}$$

Bellman's principle of optimality states that in an optimal policy "whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision" [3]. We

use this principle in the second step:

$$V^*(s) = \max_a E\left\{ r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a \right\}$$
$$V^*(s) = \max_a E\left\{ r_{t+1} + V^*(s_{t+1}) | s_t = s, a_t = a \right\}$$
$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)(\mathcal{R}^a_{ss'} + \gamma V^*(s'))$$

The last equation is Bellman's optimality equation where $\mathcal{R}^a_{ss'}$ denotes the reward given after a transition from s to s' with the probability $p(s'|s,a)$.

Chapter 3

Solution Methods

3.1 Dynamic programming

Dynamic programming describes a collection of algorithm which are used to solve a variety of problems using Bellmans's optimality principle: the problem is broken to simpler sub-problems making the solving process simpler and generally faster. In reinforcement learning dynamic programming "is used to compute optimal policies given a perfect model of the environment as a Markov decision process "[4].

3.1.1 Value Iteration

The idea behind the value iteration algorithm is simple: we first try to find the optimal value function V^* then we use it to find the optimal policy π^* where:

$$\pi^*(s) = \arg\max_a \sum_{s'} p(s'|s,a)[\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{3.1}$$

$\arg\max_a$ returns the action a which maximizes the expected return with respect to the transition probability distribution $p(s'|s,a)$. This equation returns the best action to choose for a state s. The best action is the one that has the chance to generate the maximum of rewards in the long run taking in account that all the next choices will be also optimal.

To find the optimal value function we must solve:

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)(\mathcal{R}^a_{ss'} + \gamma V^*(s')) \tag{3.2}$$

As we can see this problem is system of $|S|$ linear equations with $|S|$ unknowns, solving it is possible with classic methods but would require a big amount of computational power for large problems. A fast method to solve it is called power iteration:
We first start with an approximation V_0 chosen randomly (but having final states with a value of 0) and calculate next approximations $V_1, V_2, ..., V_n$ using the equation (3.2) at each update from $V_k$toV_{k+1}. It is proven that he sequence $\{V_k\}$ converges to V^* at $k \to \infty$ [4].
To calculate V^* all we have to do is to create two arrays, one for the old value k and the other for the next approximation $k+1$. We first calculate V_{k+1} at each step with the Bellman's optimality equation for each stat s using the last approximation V^k instead of V^*. After each step we check the difference between the two arrays, if it is small enough, we declare convergence and calculate the vector π^* for each state using (3.1).

Algorithm 1 Value Iteration

Initialize V randomly
repeat
  $\delta \leftarrow 0$
  for all $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[\mathcal{R}^a_{ss'} + \gamma V(s')]$
    $\delta \leftarrow \max(\delta, |v - V(s)|)$
  end for
until $\delta \leq \epsilon$ (a small positve number)
for all $s \in S$ **do**
  $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)[\mathcal{R}^a_{s's} + \gamma V(s')]$
end for

3.1.2 Policy Iteration

In value iteration learning the optimal value function takes a lot of time because we need to consider all actions for each state. Policy iteration uses a little trick to make the process faster, instead of learning the optimal value function V^* in one

run, we learn the value function for a certain policy, then we improve the policy and reuse it to calculate a new value function approximation and redo the same process until convergence:
$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow .... \rightarrow V^* \rightarrow \pi^*$ The algorithm has two parts: policy evaluation and policy improvement. For policy evaluation we compute the value function for a policy π then use it in policy improvement to find the best action to each state with:

$$\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)[\mathcal{R}^a_{s's} + \gamma V(s')]$$

In practice policy iteration is a lot faster than value iteration. In policy iteration the two process of policy evaluation and improvement alternate. In practice starting policy improvement before the convergence of policy evaluation can make the algorithm faster without having any deterioration in the end result, also the two functions typically converge to the optimal value function and the optimal policy.

Algorithm 2 Policy Iteration

Initialize V and π randomly
repeat

1.Policy Evaluation
repeat
$\delta \leftarrow 0$
for all $s \in S$ **do**
$v \leftarrow V(s)$
$V(s) \leftarrow \sum_{s'} p(s'|s,\pi(s))[\mathcal{R}^{\pi(s)}_{ss'} + \gamma V(s')]$
$\delta \leftarrow \max(\delta, |v - V(s)|)$
end for
until $\delta \leq \epsilon$ (a small positve number)

2.Policy Improvement
policy-stable $\leftarrow$ true
for all $s \in S$ **do**
$b \leftarrow \pi(s)$
$\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)[\mathcal{R}^a_{s's} + \gamma V(s')]$
if $b \neq \pi(s)$ **then**
policy-stable = false
end if
end for
until policy-stable = true

3.2 Temporal-Difference Methods

Temporal-Difference (TD) learning could be seen as the novel idea brought by Reinforcement Learning. Policy and value iteration are very interesting theoretically but in real life problems they bring generally no help because they assume a perfect knowledge of the environment: to calculate the value function V we need to know not just the reward, but also the probabilities $p(s'|s,a)$ for all states s' and s and actions a. In most problems we don't have such knowledge, take a game of cards for example, it's quite impossible to know exactly what is the probability of the opponent playing some card in a situation; also a robot learning to move cannot have total knowledge of their environment...
TD methods uses experience as a knowledge source, estimating implicitly at each time step the unknown probabilities: The main idea is to let the agent run in the simulation (as for example playing a round of a poker) several times and try to make estimations. In this section we will discuss two algorithms Sarsa and Q-learning.

3.2.1 The Action-Value Function

We define the function Q, called the action-value function, as the value of taking an action a in a state s and then following the policy π, formally:

$$Q^{\pi}(s,a) = E_{\pi}\left\{\sum_{0}^{\infty}\gamma^{k}r_{t+k+1}|s_t = s, at = a\right\}$$

We can see that the only difference between the value function and the action-value function is that Q takes the next action in consideration before calculating the value function following the policy π. Analogously the optimal action-value function Q^* is defined as:

$$Q^{*}(s,a) = \sum_{s'} p(s'|s,a)[\mathcal{R}^{a}_{ss'} + \gamma \max_{a'} Q^{*}(s',a')]$$

The main purpose of the action-value function is to allow the agent to calculate the different expected cumulative reward for all actions to be able to choose the best one.

3.2.2 Q-Learning

The update rule of Q-Learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \times [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Where α is the learning rate, r_{t+1} the reward gotten after the transition from s to s' using an action a. In Q-Learning the agent apply this update rule at each time step: The new value of $Q(s_t, a_t)$ is equal to the old value plus the learning rate multiplied by the reward and the difference between the action-value function of the next state and the present state. Let us analyze the components of this function:

- α_t represents learning rate with $0 < \alpha \leq 1$, a big α would induce larger changes in the action-value function, making the agent learn very fast but at the same time it has the disadvantages of making the value of Q unstable as improbable transition or reward will have greater changes over Q.
- γ is the discount factor used also in policy and value iteration, with it we can choose how much weight we want to give to future rewards, so if we want to concentrate over near future or long term future.
- $[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$ represents the change over the Q value, we use $\gamma \max_a Q(s_{t+1}, a)$ to propagate the expected reward.

As we can see Q-Learning works with estimation, the probability $p(s'|s, a)$ is implicitly estimate in the $Q(s_t, a_t)$ update rule. Every time a state s' is chosen we add its expected rewards to the Q value. Another property of the update rule is that it back propagates the expected reward: at each update we add the difference $Q(s_{t+1}, a)$ and $Q(s, a)$ to the new estimation.
Q-Learning is an off-policy learning algorithm; it uses two policies, a behavior policy used for choosing which action to take at each time step when learning, and an estimation policy that is used for evaluation and improvement. Of course the two policies meant here are generally derived from the Q function. This separation has the advantage of having a deterministic policy for estimation (for example a greedy policy, in which the best actions are always chosen) and a behavior policy that can favor exploration sampling in the process all possible states and actions. Note here that a greedy policy would be in this case $\pi(s) = \arg\max_a Q(s, a)$, for

exploration policies we could for example alter the greedy policy so that it chooses a random action sometimes (as in the case of ϵ-greedy).

Algorithm 3 Q-Learning

Initialize $Q(s, a)$ randomly $\forall s \in S$ and $a \in A$
repeat
  Initialize s
  repeat
    Choose action a from s using policy derived from Q (behavior policy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha_t \times [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until end of the episode (s is terminal)
until Q Converge or number of the episode exceeds a maximum

3.2.3 Sarsa

Like Q-Learning, Sarsa tries to approximate the action-value function however, unlike Q-Learning; Sarsa is an on-policy algorithm, which means it uses the same policy it is improving as a behavior policy. The update rule for Sarsa is very similar to the one used in Q-Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \times [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

We can notice that the only change is that the term $\gamma \max_a Q(s_t, a)$ from Q-Learning is changed to $\gamma Q(s_{t+1}, a_{t+1})$, which means that instead of using the best possible $Q(s', .)$ for the next state we use the $Q(s', a')$. In which a' is the action chosen by a policy derived from Q and is also the action that will be executed in the next time step.

Algorithm 4 Sarsa

Initialize $Q(s, a)$ randomly $\forall s \in S$ and $a \in A$
repeat
  Initialize s
  Choose a from s using a policy derived from Q
  repeat
    Take action a, observe r, s'
    Choose a' from s' using a policy derived from Q
    $Q(s, a) \leftarrow Q(s, a) + \alpha_t \times [r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until end of the episode (s is terminal)
until Q Converges or number of the episode exceeds a maximum

Chapter 4

The New Approach

4.1 Motivation

As we have seen, there are already many different algorithm and solution methods to the optimal action problem, so why do we need a new method?
The problematic that all the old methods share is the non-linearity of the solution: The optimal-cost-to-go function, also called the Bellman equation, is the basis for each of the solution ways discussed in the last chapter. This function compresses all the needed information like the future choice of actions or the future expected cost and gives us a simple way for defining "the expected cumulative cost for starting at x and acting optimally thereafter" [2].

$$v(x) = \min_u \left\{ l(x,u) + E_{x' \sim p(.|x,u)}[v(x')] \right\} \tag{4.1}$$

with: $E_{x' \sim p(.|x,u)}[v(x')] = \sum_{x'} p(x'|x,u) v(x')$

This equation (with γ set to 1 and $l(x,u)$ being the cost of being in a state x and using u) implicate the search over all actions u for each new state x. The bother is that the number of future next states grows exponentially with time, what means that all algorithm originating from this formulation will have to search for each new state through all the action (what could take a very long time in discredited continuous domains) for each state.
The pursued goal in the development of this new framework is to "construct [a]

general class of MDPs where the exhaustive search over actions is replaced with an analytical solution"[2]. The steps that will be taken are simple: first of all, a solution for the optimal action u given v is found, then this same formulation is used in the cost-to-go function to linearize it. In other words, we find an equation that expresses the optimal action u^* given v, then it to substitute the operator $\min_u$ in (4.1). But to achieve this goal we will have to put some limitations, creating in the process a new class of MDPs.

4.2 Theoretical Foundations in the Discrete Case

The first step for linearizing the optimal cost-to-go function is to find a solution for the optimal action u^* given v. However, in the discret case, we do have a set of independent actions u expressed as action-symbols, for example *move* or *stop*. Thus finding a linear equation that expresses the optimal action and returns a result like *move* is extremely hard, moreover such solution couldn't be generic as every new problem has different set of action-symbols.

To get out of this dilemma we must find a way to represent actions in a continuous manner. The solution is to use the transition probabilities instead of the symbolic actions; let's take an example:

In an environment were an agent can have two states *moving* and *stopped*, the passive dynamics being $p(x' = s|x = s) = 0.9$ with $s \in moving, stopped$ and the two actions *move* and *stop*. In the old approach the agent chooses a symbolic action as *move* for example, and as a consequence to his choice the passive dynamics, characterizing the behavior of the system in the absence of controls, will be for example altered to $p(x' = moving|x = s, u = move) = 1$ with $s \in moving, stopped$. The key here is to represent the actions by means of its underlying transition probability instead of its symbol. Formally, we set $u(x' = moving|x = s) = p(x' = moving|x = s, u = move) = 1$. This way we allow the agent to reshape directly the passive dynamics instead of passing through the symbolic actions. So, rather than choosing a symbolic action the agent chooses a new transition probability u.

But that also means that the agent have freedom to do what it wants, as choosing a $u(x' = goal|x = s)$. To prevent the agent from jumping to final states illegally a limitation is introduced: whenever $p(x'|x) = 0$ we require that $u(x'|x) = 0$. Now we must find a way to tax the agent through the immediate cost function $l(x, u)$ for the magnitude of the reshaping: the larger reshaping it does, the higher the price it pays. Namely, we must find a way to measure the difference between $u(.|x)$ (the reshaped distribution probability), and $p(.|x)$ (the passive dynamics in the absence of control). A solution is to use the Kullback-Leiber (KL) divergence:

$$l(x, u) = q(x) + \mathrm{KL}(u(.|x)\|p(.|x)) = q(x) + E_{x'\sim u(.|x)}[log\frac{u(x'|x)}{p(x'|x)}] \tag{4.2}$$

$q(x)$ is the state cost function encoding how much different states are desirable or undesirable.
Let us try now and integrate the last equation in the cost-to-go function. To simplify the results found below, we introduce the desirability function:

$$z(x) = \exp(-v(x)) \tag{4.3}$$

As we can see z is larger at states where the cost-to-go is small, reflecting the fact that the states "closer" to the goal states are more desirable. In the next steps we will try to find a way to express u^* with help of z, a formulation analogous to the cost-to-go function. Next we will try to substitute $z(x)$ in the cost-to-go function (4.1) (not forgetting to change $p(x'|x, u)$ with $u(x'|x)$):

$$\begin{aligned}
z(x) = &\exp(-v(x)) \iff v(x) = -\log(z(x)) \\
\text{and } v(x) = &\min_u \left\{l(x, u) + E_{x'\sim p(.|x,u)}[v(x')]\right\} \\
\iff -\log(z(x)) = &\min_u \left\{l(x, u) + E_{x'\sim u(.|x)}[-\log(z(x'))]+\right\} \\
\iff -\log(z(x)) = &\min_u \left\{q(x) + E_{x'\sim u(.|x)}[\log(\frac{u(x'|x)}{p(x'|x)})] + E_{x'\sim u(.|x)}[-\log(z(x'))]\right\} \\
\iff -\log(z(x)) = &\, q(x) + \min_u \left\{E_{x'\sim u(.|x)}[\log(\frac{u(x'|x)}{p(x'|x)})] - \log(z(x'))]\right\} \\
\iff -\log(z(x)) = &\, q(x) + \min_u \left\{E_{x'\sim u(.|x)}[\log(\frac{u(x'|x)}{z(x')p(x'|x)})]\right\}
\end{aligned}$$

We notice here that $\log(\frac{u(x'|x)}{z(x')p(x'|x)})$ resembles a KL divergence between u and pz,

the only problem is that pz is not a probability distribution yet. So we need to normalize pz to sum 1 by multiplying and dividing our expression by a normalization term:

$$G[z](x) = \sum_{x'} p(x'|x)z(x') = E_{x' \sim p(.|x)}[z(x')] \tag{4.4}$$

$$\begin{aligned} -\log(z(x)) &= q(x) + \min_u \left\{ E_{x' \sim u(.|x)}[\log(\frac{u(x'|x)G[z](x)}{z(x')p(x'|x)G[z](x)})] \right\} \\ \iff -\log(z(x)) &= q(x) + \min_u \left\{ E_{x' \sim u(.|x)}[\log(\frac{u(x'|x)G[z](x)}{z(x')p(x'|x)}) - \log(G[z](x))] \right\} \\ \iff -\log(z(x)) &= q(x) - \log(G[z](x)) + \min_u \{\mathrm{KL}(u \| zp/G[z])\} \end{aligned}$$

What is remarkable in this equation is $\min_u \{\mathrm{KL}(u\|zp/G[z])\}$, it means that the optimal action u^* is achieved when $\mathrm{KL}(u\|zp/G[z])$ is minimal. And we know that a KL divergence achieves its minimum 0 only if the two distributions are equal. That means that the solution of the optimal action u^* is :

$$u^*(x'|x) = \frac{p(x'|x)z(x')}{G[z](x)} \tag{4.5}$$

We can now exchange the $\min_u$ operator in the desirability function by the use of u^* instead of u:

$$\iff -\log(z(x)) = q(x) - \log(G[z](x)) + \underbrace{\mathrm{KL}(u^* \| zp/G[z])}_{=0}$$

And the final result is:

$$z(x) = \exp(-q(x))G[z](x) \tag{4.6}$$

4.3 Algorithm

4.3.1 Compact Notation

The linear equation (4.6) can be written in a more compact notation compressing all the states. For that we will use matrices:
First we enumerate the states from 1 to n and create an n-dimensional columns vector for z(x) and q(x), we call them $\mathbf{z}$ and $\mathbf{q}$. For $p(x'|x)$ we create an n-by-n matrix $\mathbf{P}$ with rows representing x and columns representing x'. (4.6) becomes

then $\mathbf{z} = \text{diag}(\exp(-\mathbf{q}))P\mathbf{z}$. Such problem can actually be seen a an eigenvector problem and can be solved by power iteration (see next section), but it can be made simpler:
We know that the cost-to-go $v(x) = q(x)$ at terminal states, which means that $z_T = \exp(-q_T)$ at terminal states x_T. z_T being known, the only unknown becomes z_N, the z-value at non-terminal states. We will note the probability transition from non-terminal to terminal states and from non-terminal to non-terminal states P_{NT} and P_{NN} respectively, the same goes for q_N and q_T.

$$z = \text{diag}(\exp(-q))Pz$$

$$\iff \text{diag}(\exp(\mathbf{q}))z = Pz$$

(we distinct z into the z value of terminal states and the z value of non-terminal states)

$$\iff \text{diag}(\exp(q_N))z_N = P_{NN}z_N + P_{NT}z_T$$

(we replace z_T by its own value)

$$\iff \text{diag}(\exp(q_N))z_N = P_{NN}z_N + P_{NT}\exp(-q_T)$$

Using this equation we can compute the z_N vector, the vector of desirabilities at the non-terminal states, using matrix factorization or an iterative linear solver [2].

4.3.2 Z-Iteration

Let $M = \text{diag}(\exp(-\mathbf{q}))P$, our equation becomes $\mathbf{z} = M\mathbf{z}$ a eigenvector problem that can be solved using the power iteration method $\mathbf{z} \leftarrow M\mathbf{z}$. The algorithm is simple, we start with a non zero vector z and do the multiplication each step until $\mathbf{z}$ converges. This algorithm is called Z-Iteration.

Algorithm 5 Z-Iteration

Initialize z ,q and p
Calculate $M = diag(exp(-q))P$
repeat
  z=z'
  z'=Mz
until $|z - z'| < \epsilon$

To use Z-iteration, all information about the control problem must be known beforehand: the probability distribution of the passive dynamics and the state costs.

The algorithm works by initializing all the vectors and matrices (probability distribution, reward and the z-matrix) and iteratively updating the z-matrix through the power iteration method until convergence. It should be noted here that the desirability value at the terminal states aren't changed in the updates and have always the value of $exp(-q_T)$ (because $v(x) = q(x)$ at terminal states).
Practically, the algorithm is seldomly implemented in this form; matrix multiplication operators in many programming languages can't handle big matrices, which make this form unusable for large problems with a big number of states and action (for example the large set of discretized continuous control problem).

4.3.3 Z-Learning

A major limitation to Z-Iteration is the necessity of knowledge of all the information about the problem in hand, specifically the rewards and the probability distributions. Z-Learning is an algorithm designed to overcome this problem. Analogically, Z-Learning is to Z-Iteration what Q-Learning is to Policy Iteration. The similarity between both (Q-Learning and Z-Learning) can be summarized in two points: both are off-policy methods and both learn by sampling the environment. Learning by sampling the environment means that, vis-a-vis to Z-Iteration, Z-Learning don't work by iterating the Z-Matrix, but update instead the z-values online from raw experience without a model of the environment's dynamics: The agent will run in a simulation updating the z-value of each state online after each visit. This is the update function, note that we note the approximation of the desirability function with $\hat{z}$:

$$\hat{z}(x_t) \leftarrow (1 - \eta_t)\hat{z}(x_t) + \eta_t \exp(-q_t)\hat{z}(x_{t+1}) \tag{4.7}$$

η_t represents the learning rate which decreases over time. Being an off-policy learning method, the Z-Learning follows another policy for the sampling than the learned one. Two variant exists for this algorithm: Random and Greedy.
Random Z-Learning, follows a random policy for sampling an episode, which means that actions are chosen randomly at each step, insuring that the passive dynamics are sampled. Greedy Z-Learning, in the other hand, uses a greedy policy for sampling, choosing the optimal action according to the last approximation $\hat{u}(x)$ at each step. Using Greedy Z-Learning requires importance sampling; a little change

in the update function must be made:

$$\hat{z}(x_t) \leftarrow (1-\eta_t)\hat{z}(x_t) + \eta_t \exp(-q_t)\hat{z}(x_{t+1})\frac{p(x_{t+1}|x_t)}{\hat{u}(x_{t+1}|xt)} \tag{4.8}$$

Using a greedy policy encourages early exploration of interesting states, making the algorithm (implicitly) produce a good policy in lesser time. But by favoring certain states over others while sampling we end up with a biased estimation of the desirability function. To compensate it we use importance sampling which requires access to the probability distributions of the system.

Chapter 5

Experiment

5.1 Motivation

Todorov's new framework enabled us to linearize the cost-to-go function by removing the $\max_a$ argument and consequently eliminating the search over the best action in each update. The resulting desirability function (4.6) was used to construct two algorithms: Z-Iteration and Z-Learning (with its two variations, greedy and random). In this section we will test the promises of these two algorithms by comparing their performance to the state-of-the-art algorithms in reinforcement learning.

Two main candidates were suitable for comparison with Z-Iteration, Value Iteration and Policy Iteration. Both, just like Z-Iteration, are offline learning algorithms. We choose the later since it performs better in comparison to Value Iteration. In the other hand, the choices for a suitable algorithm to run against Z-Learning were scarce. The only online, off-policy, learning algorithm (hence a learn-by-sampling algorithm using two different policies one for sampling and one for normal use) is Q-Learning. We choose the car-on-hill (also called Mountain car) problem described in [5] to test the performances of the algorithms against each other. In the next sections we will describe the problem and the laws governing it and then (in 5.3) expose the details of its implementation.

5.2 The problem

The task in the car-on-hill problem is to drive and park a car at the summit of a curved road in a minimal number of steps and a least possible use of energy (see Figure 5.1). The car can be controlled by the agent through the use of a continuous value of tangential acceleration. Newtonian laws govern this simulation, which means, among other things, that the car is subject gravity. The difficulty in this problem is that the gravity is stronger than the car's engine, which means that a stationary car in the bottom of the hill needs first to move in the opposite direction to the parking lot to gain inertia in order to be able to climb the goal's hill. This variant of the mountain car problem originally proposed in [4] differs by being stochastic; this was achieved by adding a Gaussian noise to the equations governing the dynamics of the system. In practice, it means that the position and the velocity of the car in the next time step are not deterministic but subject to random variation.

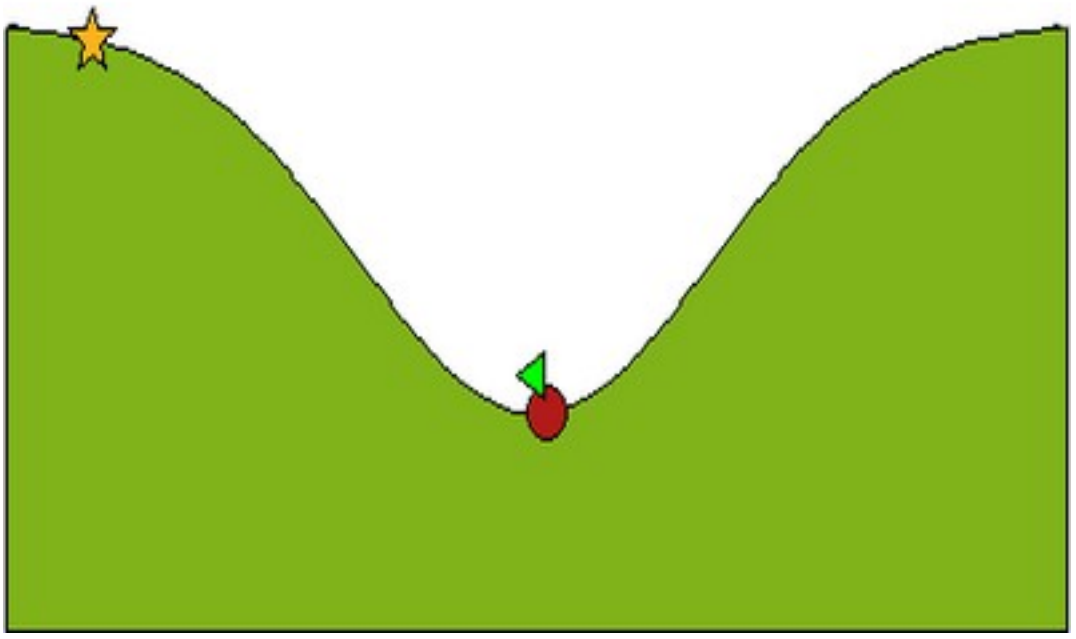

FIGURE 5.1: The mountain car graphical simulation: the red circle represents the car driven over the green hill, the arrow over it represents the acceleration direction, and the star is the parking lot

5.2.1 The Dynamics

The state of the system is determined by the horizontal position x and the tangential velocity v of the car at time t, we define the state vector $\vec{s} = \begin{pmatrix} x \\ v \end{pmatrix}$. The hill elevation in respect to the horizontal position x is:

$$h(x) = 2 - 2\exp(-x^2/2)$$

with $x \in [-3, +3]$. The slope (describing the inclination of the hill in x) is $h'(x)$ and the angle relative to the horizontal plane is therefore $\arctan(h'(x))$.
The variation in the horizontal position of the car through time depends on its speed and the angle relative to the horizontal plane:

$$dx = v\cos(\arctan(h'(x)))\,dt$$

The car is subjected to two forces: the gravity, the friction (resulting in the damping), moreover two other variable influences its tangential acceleration: the control signal u and the Gaussian noise.

$$dv = -g\ sgn(x)\sin(\arctan(h'(x)))\,dt - \beta v\,dt + u\,dt + d\omega$$

The term $-g\ sgn(x)\sin(\arctan(h'(x)))$ represents the value of the tangential component of the gravitational force with $g = 9.8$ and because $x \in [-3, +3]$, we must use $sgn(x)$ to make sure that $\sin(\arctan(h'(x)))$ doesn't change of sign in case that x is negative . The term βv with $\beta = 0.5$ represents the damping, which reduces the speed of the car over time through friction. $u \in [-30, +30]$ is the control signal and $d\omega$ represents the Gaussian noise. The task is to park the car at the summit of the hill with low velocity, we define the terminal states such as $|x - 2.5| < 0.05$ and $|v| < 0.2$.
The costs in our MDP as we have seen is $l(x, u) = q(x) + \mathrm{KL}(u(.|x)\|p(.|x))$. In our case both passive and controlled dynamics are Gaussian sharing the same covariance but different means. This can be explained by the fact that the controlled dynamics don't reshape but only shift the passive dynamics by $u dt$. The KL divergence between both can be shown to be $\frac{1}{2}u^2$, thus the immediate cost of being in state s and applying control u is $l(s, u) = q + \frac{1}{2}u^2$ with $q = 5$ in non-terminal states. The constant q determines the relative importance of time and energy: if q have a high value, the agent will prioritize time over energy, trying to reach the

goal in minimum state giving less importance to energy saving, in the other case saving energy would be its priority.

5.2.2 Discretisation

We approximate this continuous problem through the same discretisation used in [5]. For the states we use a 101-by-101 grid spanning with $x \in [-3; +3]$,$v \in [-9; 9]$, the step between two x values is 0.06 and 0.18 between two v values. For the control we use also a 101 point grid spanning $u \in [-30; +30]$. We discretize the time axis using a time step $\tau = 0.05$.
The noise distribution is discretized at 9-points spanning $[-3\sqrt{h}; +3\sqrt{h}]$ using a ± 3 standard deviation, giving us 2×9 possible next states, except the on the edges.

5.3 Implementation

5.3.1 Common part

5.3.1.1 Overview

All the algorithms have an episodic architecture: The main function *startExperience* calls at each iteration the learning method, which iterates the learning algorithm (updating the function offline in the case of policy iteration and Z-Iteration or online through sampling in the case of Q-Learning and Z-Learning). After the learning, the performance of the algorithm is tested in the simulation (*startEpisode*): we start the car-on-hill in each state and run the algorithm until it reaches a terminal state or the number of steps exceed 200.
We will discuss in this section the simulation part of the algorithm, below is the simplified pseudo-code of the simulation method:

Algorithm 6 startEpisode

```
Initialize comsumedEnergie
Initialize presentState with the starting state
for steps = 1 to maxsteps do
  action ← findBestAction()
  findBestAction is unique for each learning algorithm
  nextPossibleStates ← findNextStates(presentState, action)
  presentState ← doAction(action, presentState, nextPossibleStates)
  consumedEnergie ← consumedEnergie +1/2 · action^2
  if presentState is terminal then
    break
  end if
end for
return steps, consumedEnergie
```

5.3.1.2 Computation of next states

The computation of next states is the most important part of the simulation; it delivers a list of the next possible states according to the chosen action and the present state. The list contains, as we have seen in the last section, 2×9 states except the on the edges. This list is needed for the assignment of probabilities and the stochastic execution of action in the next step of our algorithm, described in the next section.

To compute the list, we need first to determine the spanning of the noise distribution: we use a discretized ± 3 standard distribution, which works by shifting v by $[-3\sqrt{h}, +3\sqrt{h}]$. We start by calculating

$$min.dv = -g\ sgn(x) \sin(\arctan(h'(x)))\tau - \beta v\tau + u\tau - 3\sqrt{h}$$

Notice here that we have swapped the term $d\omega$ by the minimum possible noise, which is $-3\sqrt{h}$. We use the minimum value of dv to calculate the minimum possible v and x and then add the resolution (the smallest possible change in) v for 9 iterations while recalculating the respective values of x for each. In case of stumbling upon states outside the spacial edges of the simulation ($x > 3$ or $x < -3$) or exceeding the possible speed ($v > 9$ or $v < -9$) we simply remove them and note where they are. Below is a simplified pseudo-code of the function:

Algorithm 7 computeNextStates

Input x,v,u and *vResolution*
Initialize *nextStates*
min-dv $\leftarrow -g\ sgn(x)\sin(\arctan(h'(x)))\tau - \beta v\tau + u\tau - 3\sqrt{h}$
$v' \leftarrow v + \text{min-dv}$
for $i = 1$ to 9 **do**
  dx $\leftarrow v'\cos(\arctan(h'(x)))\tau$
  $x' \leftarrow x + dx$
  if $\begin{pmatrix} x' \\ v' \end{pmatrix}$ is a valid state **then**
    add $\begin{pmatrix} x' \\ v' \end{pmatrix}$ to *nextStates*
  end if
  $v' \leftarrow v' + vResolution$
end for
return *nextStates*

5.3.1.3 Execution of Actions

We determine the next state following an action using the list of possible next states that we've computed and a random number generated by a standard distribution. The random number determines which state is chosen. In the case that we have less than 9 states (some states were invalid), we simply normalize the distribution after removing the values representing the invalid states.

5.3.2 Policy Iteration and Z-Iteration

Policy Iteration and Z-Iteration are both offline learning method, which mean that they compute the value and desirability functions respectively using the dynamics and the foreknown rewards. The resulting policies are then used in the simulation. We compute the discretized probability distribution empirically through generation of a big number of random Gaussian standard values and assigning each to one of the sets representing the discrete values then normalize their probabilities. We used the algorithms 2 and 5 that we've described in the last sections for the implementation with minor changes that we will discuss here.
For the implementation of Policy Iteration we used a discount factor $\gamma = 1$ to have equivalent results for Z-Iteration. Because of the slowness of the algorithm we were compelled to resort to some optimizations: the policy evaluation step was repeated only 20 times and not until convergence as in the original algorithm,

furthermore, the list of future states for each action accordingly to the present states and their probability distributions was computed for all states beforehand and saved in a 3-dimensional Matrix to reduce redundant calculations in the main algorithm. This resulted in an increased space complexity and a reduced the computational complexity, which is favorable in our case because our problem doesn't need much memory but takes long time to run. We must bring attention though, that such optimizations may not be possible or beneficial in larger problems. The used reward was $r(x,u) = -q(x) - \frac{1}{2}u^2$ which is equivalent to the cost function used for Z-Iteration.

Z-Iteration was implemented without any optimizations, we just used a loop iterating over all state to update their values instead of matrix multiplication. The choice of action was not implemented using the optimal action function developed by Todorov, instead we opted for choosing the action that maximize the expected desirability, formally $u^*(x) = \arg\max_u \sum_{x'} p(x'|x,u)z(x')$, we calculate u^* by iterating over all discrete values of u and choosing the one maximizing the right side of the equation. This change makes choosing action slower than the original method but doesn't affect the performance measurement of the algorithms.

5.3.3 Q-Learning and Z-Learning

Q-Learning was implemented using the algorithm 3 described in the third chapter, the update rule is:

$$Q(x,u) \leftarrow (1-\alpha_t)Q(x,u_t) + \alpha_t \times [r(x',u_t) + \gamma \max_u Q(x',u)]$$

we used a discount factor $\gamma = 1$ as in the formulation of Z-Learning. The learning rate $\alpha_t = \frac{40000}{40000+t}$ decreases over time. The reward (cost) for using an action u in a state x is $r(x,u_t) = -q(x) - \frac{1}{2}u^2$. Two variant of the algorithm were used for sampling, one using random sampling policy and the other using an ϵ-Greedy policy with $\epsilon = 0.15$. The ϵ-Greedy policy choose the best action $\arg\max_u Q(x,u)$ with probability $1-\epsilon$ and a random action the rest of the time (with probability ϵ). This policy performs better than the random policy because it favors promising actions while exploring new possibilities from time to time, it is also proved to converge.

We implemented also two version of Z-Learning, while changing the learning rate

α_t to $\frac{7000}{7000+t}$ because of the faster convergence of Z-Learning. For the Greedy version, the policy always chose the best action (unlike the ϵ-greedy policy) because exploration is already done (implicitly) by the algorithm: Z-Learning uses the passive dynamics for sampling . Below is the algorithm used for Z-Learning with a random behaviour policy (for one episode), for the greedy Z-Learning we just have to multiply the right term in the update rule by $\frac{p(x_{t+1}|x_t)}{\hat{u}(x_{t+1}|xt)}$ and of course choose the optimal action for behaviour according to the desirability approximation $\hat{z}$.

Algorithm 8 Z-Learning

Input $\hat{z}$
Initialize x
repeat
  u = randomPolicy()
  executeAction(u) and observe new state x' and $q(x')$
  $\hat{z}(x) \leftarrow (1-\alpha_t)\hat{z}(x) + \alpha_t \exp(-q(x'))\hat{z}(x')$
  $x \leftarrow x'$
until x is terminal or maximal number of steps exceeded
return $\hat{z}$

Chapter 6

Results and Discussion

6.1 Z-Iteration and Policy Iteration

In our benchmarks Z-Iteration performed better then Policy Iteration by reaching convergence in lesser steps and needing less time to run one iteration.
The two algorithms resulted in policies with similar performance, needing in average 30 steps to finish one episode and 3000 Energy points (calculated as $\frac{1}{2}u^2$). Z-Iteration converged in our tests after approximately 15 iterations. Policy Iteration needed 9 updates, but we know that an update is in fact 20 policy improvement iterations. So Policy Iteration converged approximately after $9 \cdot 20 = 180$ iterations. In other words, Policy Iteration took 10 times more time to converge.

6.2 Z-Learning and Q-Learning

We've run our simulation for Z-Learning and Q-Learning, each using two behavior policy to sample the data: greedy and random. In our test greedy Z-Learning outperformed the greedy variant of Q-Learning and random Z-Learning outperformed random Q-Learning. But before discussing further the results, we should remark that the comparison between the two algorithms is not totally fair: first, random Z-Learning, although it doesn't need any information about the environment for its update rule, needs the probability distributions of the dynamics to determine the optimal action. In other words, random Z-Learning can only compute the desirability function without the needed information about the environment in

opposite to the Q-Learning algorithm, which can also find the optimal actions without further knowledge. Second, greedy Z-Learning requires also importance sampling based on the passive dynamics for the update rule.

The computational complexity of random Z-Learning in one episode with n steps is $O(n)$, for random Q-Learning it is $O(n \cdot a)$, with a the number of possible actions $|u|$. The reason behind this is that the update rule of Q-Learning, requires search for the maximal $Q(s', u')$ in each update:
$Q(s, u) \leftarrow Q(s, u) + \alpha_t \times [r + \max_{u'} \gamma Q(s', u') - Q(s, u)]$
Another advantage for the Z-Learning is that it learns directly the desirability $\hat{Z}(s)$ of a state and not for each state-action tupel $\hat{Q}(s, u)$, which spare us time. The space complexity is $O(n)$ for Z-Learning and $O(n \cdot a)$ for Q-Learning with n the number of states s, and a the number of possible actions u.
In our tests, greedy Z-Learning needed 1000 steps to converge against arround 11000 for greedy Q-Learning, in the other hand random Z-Learning needed aproximatly 10000 steps, random Q-Learning didn't reach convergence even after 25000 iterations. The resulting policies had the same performance, an episode needed in average 30 steps to be finished and 3000 Energy points (Energy is $\frac{1}{2}u^2$)

6.3 Discussion

Both Z-Learning and Z-Iteration performed better in the car-on-hill problem: their computational complexity was lower than the state-of-the-art variants, which made their iterations run much faster, also they've also converged faster to the sought policy. But does this makes them candidates for solving all future reinforcement learning problems? Unfortunately no, the class of problems that can be solved by these two algorithms is limited.
The costs must be a KL divergence, which limits us to the quadratic energy cost in continuous settings and adds another constraint: the state transitions provoked by the actions must be the same provoked by the noise, which means that systems with controls acting outside the passive dynamics cannot be modeled in this framework.
The use of Z-Learning as a substitute to Q-Learning is also not practical. Although it performs better, Z-Learning needs far more information about the environment than Q-Learning. Which makes it only advantageous to Z-Iteration when the state cost $q(x)$ is unknown. The problem can be solved if a model of the passive

dynamics p could be learned online without influencing the performance of the algorithm.

To summarize, we think that this new method can be very useful in a limited class of problems. Problems having a large number of actions (for example discretized continuous problems) which could be only solved after long computations can now be solved in a realistic time through these new algorithms. We believe that any future work that can remove or loosen the limitations could make from this algorithm a worthy substitute to the state-of-the-art solutions.

Chapter 7

Conclusion

We've tried in the last chapters to describe the new framework developed by Todorov in a literal and a formal way, noting in the process its theoretical background and mathematical development. This new method resulted in two different algorithms, which we've tried to compare to the state-of-the-art solutions in terms of complexity and speed. The results were generally satisfying, Z-Iteration and Z-Learning performed better than Policy Iteration and Q-Learning. The comparison wasn't always fair, as in the case of Z-Learning and Q-Learning

Unfortunately, the use of these algorithms is restricted to a limited class of problems. First the controls can only evoke state transitions which can be induced with the passive dynamics, in other words the actions cannot cause transition which are not reachable by noise. Second, the control cost must be a KL divergence, which eliminates a great number of systems and restrain us from modeling ones with the goal of minimizing time or anything other than energy. In fact it reduces the class to the systems with quadratic energy costs in continuous settings. Lastly, due to the formulation of costs by using the KL divergence between the passive and the controlled dynamics, the noise amplitude have great influence over the control costs. This can become problematic especially when the noise amplitude is small, which would result in larger control cost. Another restriction which is exclusive to the greedy Z-Iteration algorithm is the need of access to the dynamics of the environment to be able to use the importance sampling corrections that are needed.

Despite its limitations, this new method has great potential in large scale problems where traditional approaches are generally slow. Researchers are starting to use

it to solve problems as in [6], where it's used to control character animations. Hopefully more research would be done to loosen the restrictions and make this framework more appealing.

Bibliography

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[2] E. Todorov. Efficient computation of optimal actions. *PNAS USA*, 106(3): 1147811483, March 2009.

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[5] Todorov. Efficient computation of optimal actions: Supplementary notes. ., 2009.

[6] Marco da Silva, Frédo Durand, and Jovan Popović. Linear bellman combination for control of character animation. *ACM Transactions on Graphics*, 28 (3):82:1–82:10, July 2009.