Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университут)
Факультет управления и прикладной математики
Вычислительный центр им А. А. Дородницына РАН

03.04.01 Прикладные математика и физика

Изучение пространства признаков в задаче обучения с подкреплением

Выпускная квалификационная работа магистра

Автор .....
Гринчук Алексей Валерьевич
Кафедра интеллектуальных систем

Научный руководитель .....
Оселедец Иван Валерьевич
Доктор физико-математических наук
Профессор

Заведующий кафедрой .....
Рудаков Константин Владимирович
Доктор физико-математических наук
Профессор, академик РАН

Москва
2017

Аннотация

Последние достижения в области алгоритмов обучения с подкреплением, аппроксимирующие функцию полезности при помощи нейронных сетей, успешно применяются для решения ряда практических задач, однако теоретические аспекты их использования всё ещё требуют тщательного изучения. Нейронные сети могут находить хорошие признаковые описания состояний среды, однако до сих пор ведутся споры о том, как именно им это удаётся. Для лучшего понимания механизма работы нейронных сетей мы предлагаем использовать идеи, полученные в результате многих лет исследований в области линейной аппроксимации функции полезности.

Эта работа продолжает теорию линейного сжатия признаков и исследует проблему нахождения множества признаков, в котором линейная аппроксимация функции полезности оптимальна в смысле минимизации ошибки Беллмана. Мы представляем Compressed Value Iteration — алгоритм линейного кодирования, которой последовательно расширяет пространство признаков до момента, когда оптимальное множество признаков найдено. Мы формулируем достаточные условия для того, чтобы набор признаков был оптимальным и доказываем, что этот набор лежит в подпространстве Крылова, образованном аппроксимацией вектора наград и матрицы модели среды.

Наша подход состоит из двух шагов. На первом шаге мы ищем матрицу перехода из пространства высокой размерности (в котором состояния среды представлены в виде картинок) в пространство низкой размерности, признаковое описание состояний среды в котором является оптимальным. На втором шаге мы ищем линейную аппроксимацию функции полезности при помощи алгоритма Least-Squares Policy Iteration (LSPI) и находим оптимальную стратегию поведения агента обучения с подкреплением.

Для тестирования предложенного алгоритма и сравнения его с другими популярными подходами мы провели серию экспериментов на различных задачах обучения с подкреплением. Положительные результаты, полученные в задаче классического контроля “Перевёрнутый Маятник”, а также в игре “Пинг-Понг” из серии игр Атари 2600, с состояниями, представленными в виде картинок, подтверждают работоспособность предложенного подхода.

Содержание

Глава 1

Введение

1.1 Мотивация

The reinforcement learning paradigm was first introduced in the late 1980s as an attempt to apply an optimal control theory to learning by trial and error — a learning method common to both human beings and animals. The idea to develop strict theoretical substantiation for the social phenomena of self-learning had one ultimate goal — to bring the mankind to the creation of artificial intelligence. However, after the development of the necessary mathematical apparatus, scientists faced another conundrum. To solve real-world problems, reinforcement learning algorithms required prodigious computational resources, unavailable at that time. This gap between theoretical grounding and practical implementation persisted until the year of 2015.

During the last two years reinforcement learning experienced a powerful rise and became very popular topic in machine learning field, and computer science in general. Deep neural networks which outperformed state-of-the art approaches in such fields as computer vision, natural language processing, and speech recognition showed extremely good results on a bunch of complicated reinforcement learning problems. For example, a deep reinforcement learning agent developed for playing the game of Go managed to beat the world's best professional players, while all other existing algorithms did not surpass even the amateur dan.

The excessive popularity of deep reinforcement learning at first quickly overcame the gap between theory and practice, but then it started to create the gap in opposite direction. Algorithms based on neural networks began to use a profusion of different heuristics to achieve better results on real-world problems providing no theoretical grounding. Therefore,

despite the successes in practice and, arguably because of them, we need to provide thoughtful study and investigation of theoretical aspects behind the usage of neural networks for solving reinforcement learning problems.

In this work we show that neural networks used for solving reinforcement learning problems nowadays can be considered as linear value function approximations on so called *encoded* feature representations. We investigate a problem of finding a low-dimensional set of features in which linear value function approximation is optimal. We introduce a *compressed value iteration* algorithm for searching for good feature representations of the states and vindicate the eligibility of the proposed approach by computational experiments on various reinforcement learning problems.

1.2 Описание задачи

In this section we provide brief description of the problem and why it is important to solve it. More formal and mathematically strict problem statement is presented in Chapter 2 after we introduce necessary notation and background.

Feature representation of states and actions is of vital importance in reinforcement learning, as the quality of value function or policy is largely determined by corresponding features. However, major part of reinforcement learning environments enables superfluous representations of states, when the number of features is much bigger then the number of effective parameters which determine the dynamics of the whole system. For example, classical problem of balancing 2D inverted pendulum has only two effective parameters, angle and angular velocity, which fully determine the system's behavior. If we draw the pendulum as an image of resolution 30×60 pixels (which is quite low resolution), the number of features will equal 1800.

Training time of reinforcement learning algorithms is directly proportional to the dimensionality of feature space, and, if we manage to reduce the number of features in states representations, we can get significant speed improvement. Despite the variety of methods for feature space dimensionality reduction (e.g. principal component analysis and its modifications), there are no theoretical guarantees that they are applicable to reinforcement learning. This observation brings us to the informal problem statement: to find a transformation from high-dimensional raw feature space to low-dimensional feature space sufficient for learning an optimal value function and policy.

In addition to speeding up existing reinforcement learning algorithms, there is one more very interesting perspective of looking for low-dimensional feature space representation. Deep neural networks trained with stochastic gradient descent find good feature representations, but the discussion of how they do it is still open. Theoretical concepts of linear feature encoding we present in this work, if extended to more complicated case of non-linear neural networks, can provide better understanding of the role of features in reinforcement learning and improve the existing algorithms.

1.3 Обзор литературы

The most complete exposition of reinforcement learning with its achievements and challenges is presented in the book of Sutton and Barto [1]. Basic concepts of Markov decision processes (MDPs), which are a mathematical core of reinforcement learning, can be found in the work of Puterman [2]. Dynamic programming paradigm which preceded reinforcement learning was first introduced by Bellman [3]. A lot of work in the field of approximate dynamic programming was done by Bertsekas and Tsitsiklis [4].

Feature construction has been and remains an important topic for reinforcement learning. Expert features, being used instead of the original ones, demonstrated a huge performance improvement in TD-gammon [5]. In the following years a lot of theoretical and practical work was done on understanding how to generate good features for linear value function approximation. Mahadevan and Maggioni introduced a Laplacian framework for learning representations for linear value function approximation [6], while Parr at al. used Bellman error approach for analyzing and generating features [7]. Petrik [8] conducted further analysis of Laplacian methods and demonstrated that augmented Krylov methods may significantly outperform them. Parr et al. [9] presented elaborate analysis of linear value function approximation theory and proved an equivalence of augmented Krylov methods and Bellman error based methods for feature selection in reinforcement learning.

More recent practical advances which used deep neural networks for state-action value function approximation showed extremely successful results and initiated a new wave of interest in reinforcement learning. Mnih et al. [10] described a reinforcement learning system, referred to as Deep Q-Networks (DQN) which learned to play dozens of Atari 2600 games and managed to outperform a good human player in a number of them. However, the real triumph of deep reinforcement learning happened after a self-trained agent, part of the AlphaGo framework, beat 18-time Go world champion Lee Sedol in a series of five games.

The computer program AlphaGo developed by Google DeepMind [11] managed to find good value function approximation based on DQN and outplay a professional Go player, which had been regarded impossible for a long time.

There are several other very interesting examples of cross-disciplinary research based on deep reinforcement learning. A step forward to creating true artificial intelligence was done in a work on robotics of Gureshi et al. [12] where robot was self-trained to interact with humans. Leibo et al. [13] used reinforcement learning agents in social games to test some aspects of game theory in practice. Olivecrona et al. [14] applied deep reinforcement learning to computational de-novo drug discovery in chemoinformatics.

Despite the insane success of deep reinforcement learning, there is one problem still in existence — neural networks require expensive computational clusters of GPUs and need days of training time to create a really good agent. A great deal of work was done in a direction of optimizing DQN in terms of reducing training time and improving computational efficiency. Mnih et al. [15] introduced Advanced Asynchronous Actor Critic (A3C) model which works on CPU and uses tens of agents interacting with the environment in parallel. Pritzel et al. [16] proposed to use kd-tree based memory layer in neural networks to increase efficiency of training and reduce both training time and training data size.

As an example of the connection between practical neural network techniques and linear value function approximation theory, we note that Oh et al. [17] trained an action-conditional encoder for the next state prediction. Despite the next states prediction is a well-known technique used in neural networks, more recent linear value-function approximation theory proves that predicting next states is sufficient to ensure the existence of optimal value function [9]. This concept was generalized by Song et al. [18] to state-action value functions, where a simple linear encoder trained to predict the next states was able to learn good policies for Inverted Pendulum and Black Jack problems with states represented as raw images.

1.4 Научная новизна

This section outlines main contributions of the proposed work.

1. We develop further theory of linear feature encoding for reinforcement learning. We prove that it is sufficient to be able to predict encoded next states instead of raw

ones [18] to guarantee the optimality of linear value function approximation with linear encoder and discuss the insights behind this idea.

2. We present *compressed value iteration* – an algorithm which extends the feature set iteratively to solve the optimization problem induced by the idea of the next encoded state prediction. Although this problem is quadratic in general, its form is a bit tricky and there is no straightforward way to solve it with standard methods.

3. We discuss two modifications of compressed value iteration. One of them uses Krylov methods for feature generation similar to Petrik [8], another one benefits from sparse representation of the states common to most reinforcement learning environments. These modifications allow us to achieve significant speed up comparing to original compressed value iteration and other methods for linear value function approximation.

4. In a series of computational experiments we vindicate the eligibility of the presented method and compare it with other popular approaches for feature encoding. As a benchmark we choose two reinforcement learning environments. The first one is Inverted Pendulum classic control problem with states represented by two successive raw images of the pendulum instead of its natural hidden parameters — angle and angular velocity. The second one is the Atari 2600 game Pong with states represented by four successive frames of the gaming screen transformed into gray scale from RGB.

1.5 Структура работы

This chapter is aimed to delineate the current status of reinforcement learning research and introduce the problem of searching for a low-dimensional state space representation, we solve in this thesis. Chapter 2 contains all necessary preliminaries, background, and notation to formulate the problem mathematically. It also describes the Least-Squares Policy Iteration algorithm — a standard method for solving reinforcement learning problems with linear value function approximation.

In Chapter 3 we present *compressed value iteration* — a linear encoding method for solving the aforementioned problem and discuss its peculiarities. Chapter 4 is devoted to computational experiments aimed to prove the eligibility of the proposed approach and compare it to other methods of constructing features in reinforcement learning problems. Finally, Chapter 5 concludes the presented work and discusses its perspective extensions, including possible directions of future work.

Глава 2

Базовые понятия

2.1 Задача обучения с подкреплением

Reinforcement learning problem consists of three basic elements: a learning agent, an environment, and a reward signal. At each time step t, the agent receives some representation of the environment's *state* $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states, and on that basis selects an *action*, $a_t \in \mathcal{A}$ from the set of available actions. The environment reacts to the agent's action by giving it a numerical reward $r_{t+1} \in \mathbb{R}$, and changing its state to a new state s_{t+1} [1]. Through such interactions, the agent collects experience of how the environment works and aims to understand how it should behave to get the maximum amount of reward. Figure 2.1 represents the scheme of agent-environment interaction.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's *policy* and is denoted by $\pi : \mathcal{S} \to \mathcal{A}$, where $\pi(a|s) = \mathbb{P}(a_t = a|s_t = s)$ is the probability of choosing action a while being in state s. To solve reinforcement learning problem means to find such policy π that maximizes the total expected amount of reward:

$$\mathbb{E}_\pi(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots) \to \max_\pi$$

where $0 \leq \gamma < 1$ is a parameter called the *discount rate* — a heuristics which guarantees that the infinite sum of rewards has a finite value for bounded values of rewards r.

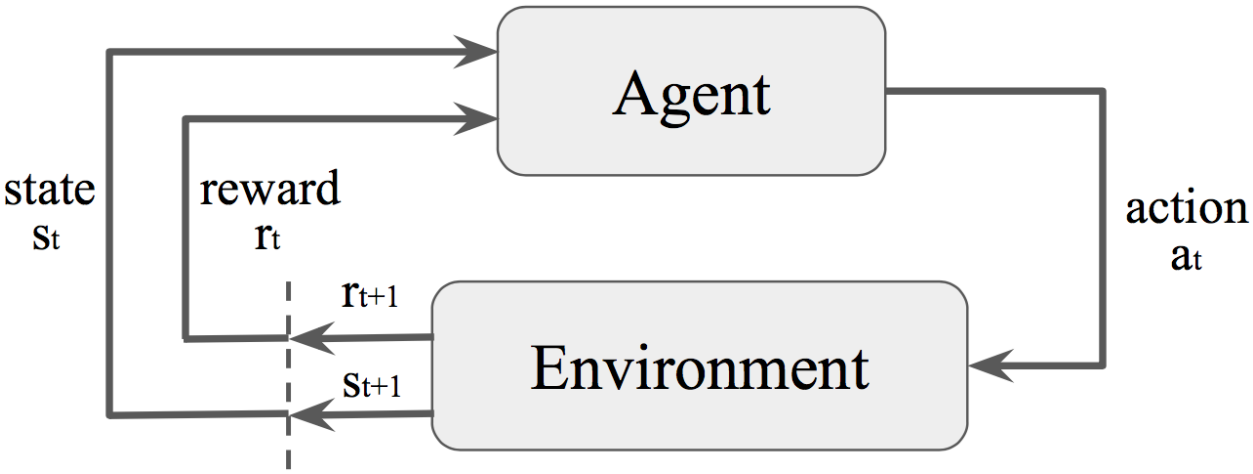


Figure 2.1 — The agent-environment interaction in reinforcement learning. This image was adapted from Sutton and Barto book [1].

2.2 Марковский решающий процесс

The mathematical formalization of the reinforcement learning problem is a Markov Decision Processes (MDPs). In general, a Markov Decision Process (MDP) is a fivetuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is the state set with states represented by vectors of dimensionality l: $s \in \mathbb{R}^l$ in our setting, $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is the action set. The reward function $\mathcal{R}(s_i, a_j)$ denotes the expected immediate reward when taking action a_j in state s_i, the transition function $\mathcal{P}(s_i, a, s_j)$ denotes the probability of transiting from state s_i to state s_j when taking an action a, and $\gamma \in [0, 1)$ is the discount factor for future reward. The policy π in an MDP can be represented in terms of probability of taking action a when in state s: $\pi(a|s) \in [0, 1]$ and $\sum_a \pi(a|s) = 1$.

In this work we use more convenient matrix notation. We define raw state-action matrix $A \in \mathbb{R}^{nm \times ml}$ where each row represents a state-action pair (s_i, a_j). All rows are split into m blocks of size l and only the block corresponding to the action taken is non-zero: it contains a vector of size l which is the vector of corresponding state (see Figure 2.2 for an illustration). The reward function is represented as a vector $R(s, a) \in \mathbb{R}^{nm \times 1}$ and the transition function is represented as a matrix $P((s, a), s') \in \mathbb{R}^{nm \times n}$. Given a policy π, we define $P^\pi(s', a'|s, a) \in \mathbb{R}^{nm \times nm}$ as the transition probability for the state action pairs, where $P^\pi(s', a'|s, a) = P((s, a), s')\pi(a'|s')$.

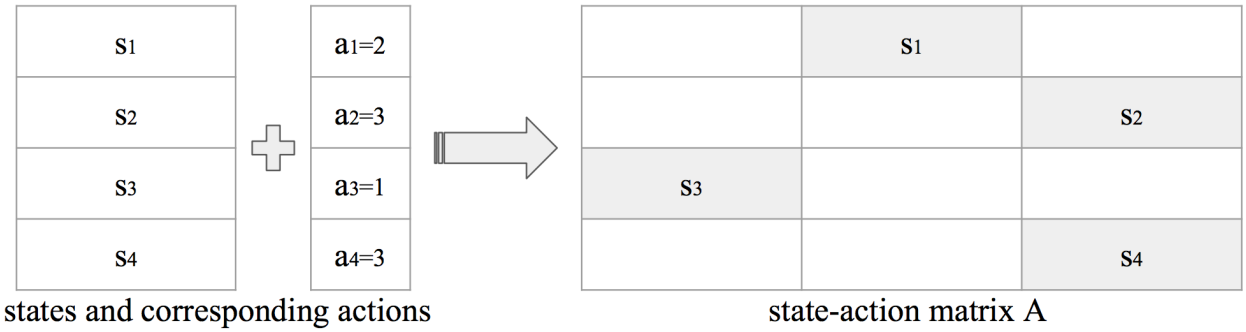


Figure 2.2 — Illustration of how the state-action matrix A is constructed for four different states and three actions.

For any policy π, its Q-function $Q^\pi(s,a)$ is defined over the state-action pairs (s,a) and represents total γ-discounted reward when taking action a in state s and following π afterwards. We consider Q-function as a vector $Q^\pi(s,a) \in \mathbb{R}^{nm\times 1}$. The Q-function satisfies the Bellman equation which has the following form in presented above matrix notation:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s',a'} P^\pi(s,a|s',a')Q^\pi(s',a'). \tag{2.1}$$

2.3 Линейная аппроксимация функции полезности

There are some cases when Q-function can be represented exactly as a closed-form solution to Bellman equation, but in general it is not possible. For example, when the number of states is too big or even continuous, it is impossible to apply the dynamic programming approach because of a huge (or infinite) size of the table containing Q-values. When it is impossible to represent the Q-function exactly, it is usually approximated with a function from some parametric family $\hat{Q}^\pi \in \mathcal{B}(\theta)$.

The Bellman operator T^π [18] on the Q-functions is defined as

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s',a'} P^\pi(s,a|s',a')Q^\pi(s',a').$$

In terms of T^π, the Bellman equation can be rewritten in a form of $T^\pi Q^\pi = Q^\pi$, where Q^π is a fixed point of T^π. We define the *Bellman error* for an approximated Q-function $\hat{Q}^\pi$ as $\mathrm{BE}(\hat{Q}^\pi) = T^\pi\hat{Q}^\pi - \hat{Q}^\pi$. When the Bellman error is 0, the Q-function is at the fixed point. Otherwise, we have [19]:

$$\|\hat{Q}^\pi - Q^\pi\|_\infty \leq \frac{1}{1-\gamma}\|\hat{Q}^\pi - T^\pi\hat{Q}^\pi\|_\infty,$$

where $\|\mathbf{x}\|_\infty$ refers to the l_∞ norm of a vector $\mathbf{x}$.

In this work, we consider linear Q-function approximation: $\hat{Q}^\pi = A\mathbf{w}_A^\pi$, where $\mathbf{w}_A^\pi \in \mathbb{R}^{ml\times 1}$ is the weight vector. The *linear fixed point* methods [20–22] find $\mathbf{w}_A^\pi$ as a solution of so-called fixed point equation:

$$A\mathbf{w}_A^\pi = \Pi(R + \gamma P^\pi A\mathbf{w}_A^\pi), \tag{2.2}$$

where $\Pi = A(A^TA)^{-1}A^T$ is the orthogonal l_2 projector on span(A). Solving 2.2 leads to the following linear fixed-point solution:

$$\mathbf{w}_A^\pi = (A^TA - \gamma A^TP^\pi A)^{-1}A^TR. \tag{2.3}$$

2.4 Теория линейного сжатия

Computing $\mathbf{w}_A^\pi$ in a form of 2.3 exhibits two serious drawbacks. Firstly, it contains matrix inversion which is undefined if a matrix is not full rank. This happens, for example, if a number of features is greater than a number of states. This situation is vacuous if we have access to all states, as in this case we can simply reduce number of features, eliminating superfluous ones. In practice we usually do not have access to all states and have to use samples from the agent's experience. Secondly, computational complexity of matrix inversion in this case is $\mathcal{O}(\{ml\}^3)$ which makes this method inapplicable to real-world problems, as they usually have $ml \approx 10^4 - 10^5$.

The main idea of feature encoding is to find another, low-dimensional feature representation of the state-action pairs. Similar to [18], we refer to the transformation which reduces a number of features using an *encoder*.

Definition 1. The encoder $\mathcal{E}_\pi$ is a transformation $\mathcal{E}_\pi : \mathbb{R}^{nm\times ml} \to \mathbb{R}^{nm\times k}$ from high-dimensional raw feature space of matrices A to low-dimensional encoded feature space of matrices $\Phi = \mathcal{E}_\pi(A)$. If the encoder is linear, $\Phi = \mathcal{E}_\pi(A) = AE_\pi$ where $E_\pi \in \mathbb{R}^{ml\times k}$.

In terms of the encoded feature space representation, linear Q-function approximation and solution to the fixed point equation 2.3 have the following form:

$$\hat{Q}^\pi = \Phi \mathbf{w}^\pi_\Phi, \qquad \mathbf{w}^\pi_\Phi = (\Phi^T\Phi - \gamma\Phi^T P^\pi \Phi)^{-1}\Phi^T R$$

Instead of approximating the Q-function linearly and solving the fixed-point equation 2.2, we can approximate the reward function R and the expected policy-conditional next feature $P^\pi\Phi$ to force the solution of approximated Bellman equation to be linear. According to Parr et al. [9], we can do it, using the following linear model:

$$\hat{R} = \Phi r_\Phi = \Phi(\Phi^T\Phi)^{-1}\Phi^T R, \tag{2.4a}$$

$$\widehat{P^\pi\Phi} = \Phi P^\pi_\Phi = \Phi(\Phi^T\Phi)^{-1}\Phi^T P^\pi\Phi. \tag{2.4b}$$

Interestingly, the linear fixed-point solution and the linear model solution are the same [7], so there is no need to differentiate them. However, the linear model paradigm gives us an opportunity to decompose the Bellman error into a sum of reward error and policy-conditional per-feature error:

$$\Delta_R = R - \hat{R} = R - \Phi r_\Phi, \tag{2.5a}$$

$$\Delta^\pi_\Phi = P^\pi\Phi - \widehat{P^\pi\Phi} = P^\pi\Phi - \Phi P^\pi_\Phi. \tag{2.5b}$$

Theorem 1. (Parr et al.) For any MDP $\mathcal{M}$ with feature representation Φ, and policy π represented as the fixed point of the approximate Q-function, the Bellman error can be represented as

$$\mathrm{BE}(\hat{Q}^\pi) = \Delta_R + \gamma\Delta^\pi_\Phi, \mathbf{w}^\pi_\Phi$$

From Theorem 1 it is clear that condition $\Delta_R = 0$, $\Delta_\Phi = 0$ is sufficient (but not necessary) to achieve zero Bellman error and a perfect Q-function. Specifically, it requires that the features of the approximate model capture the structure of the reward function, and that the features of the approximate model are sufficient to predict expected next features [9].

2.5 Обучение по выборке

In practice we do not have access to the model of the environment P^π, but do have access to samples from agent-environment interaction history, obtained after exploitation of

some policy. In such setting, the environment is considered a "black box" — it receives some policy π as an input and outputs a desired number of fourtuples (s, a, r, s').

In our notation, state-action matrix A and reward vector R will now have N rows, instead of nm, where N is a number of samples. We define state-action matrix of raw states $A \in \mathbb{R}^{N \times ml}$, built from pairs (s, a); reward vector $R \in \mathbb{R}^{N \times 1}$, built from rewards r; and state-action matrix of raw next states $A' \in \mathbb{R}^{N \times ml}$, built from pairs (s', a'), where actions a' are sampled from the environment with some policy π, correspondingly to states s'. We refer to $\Phi = AE_\pi \in \mathbb{R}^{N \times k}$ as to a matrix of encoded states and to $\Phi' = A'E_\pi \in \mathbb{R}^{N \times k}$ as to a matrix of encoded next states.

2.6 Least-Squares Policy Iteration

Least-Squares Policy Iteration (LSPI) algorithm [23] is a method of discovering the optimal policy for any given MDP, which belongs to the class of so-called *policy iteration* methods [24] — a standard way to solve MDPs. LSPI starts from some policy π_0, usually random, and discovers the optimal policy by generating a sequence of monotonically improving policies.

Each learning iteration of LSPI consists of two steps. At the first step we evaluate current policy π_i and find linear Q-function approximation $\hat{Q}^{\pi_i} = \Phi \mathbf{w}_\Phi^{\pi_i}$ with the LSTDQ procedure, which simply calculates the vector of weights $\mathbf{w}_\Phi^\pi$. At the second step we construct improved greedy policy π_{i+1} over $\hat{Q}^{\pi_i}$ as

$$\pi_{i+1}(s, a) = \begin{cases} 1, & a = \arg\max_{a \in \mathcal{A}} \hat{Q}^{\pi_i}(s, a), \\ 0, & \text{otherwise.} \end{cases}$$

Policy π_{i+1} is a deterministic policy which is better than π_i, if the policy evaluation step is exact. However, if the policy evaluation step is approximate, as it is in the case with the LSTDQ, then the new policy is not guaranteed to be uniformly better. These two steps (*policy evaluation* and *policy improvement*) are repeated until convergence. Algorithm 1 represents LSPI algorithm, its block diagram is portrayed as Figure 2.3.

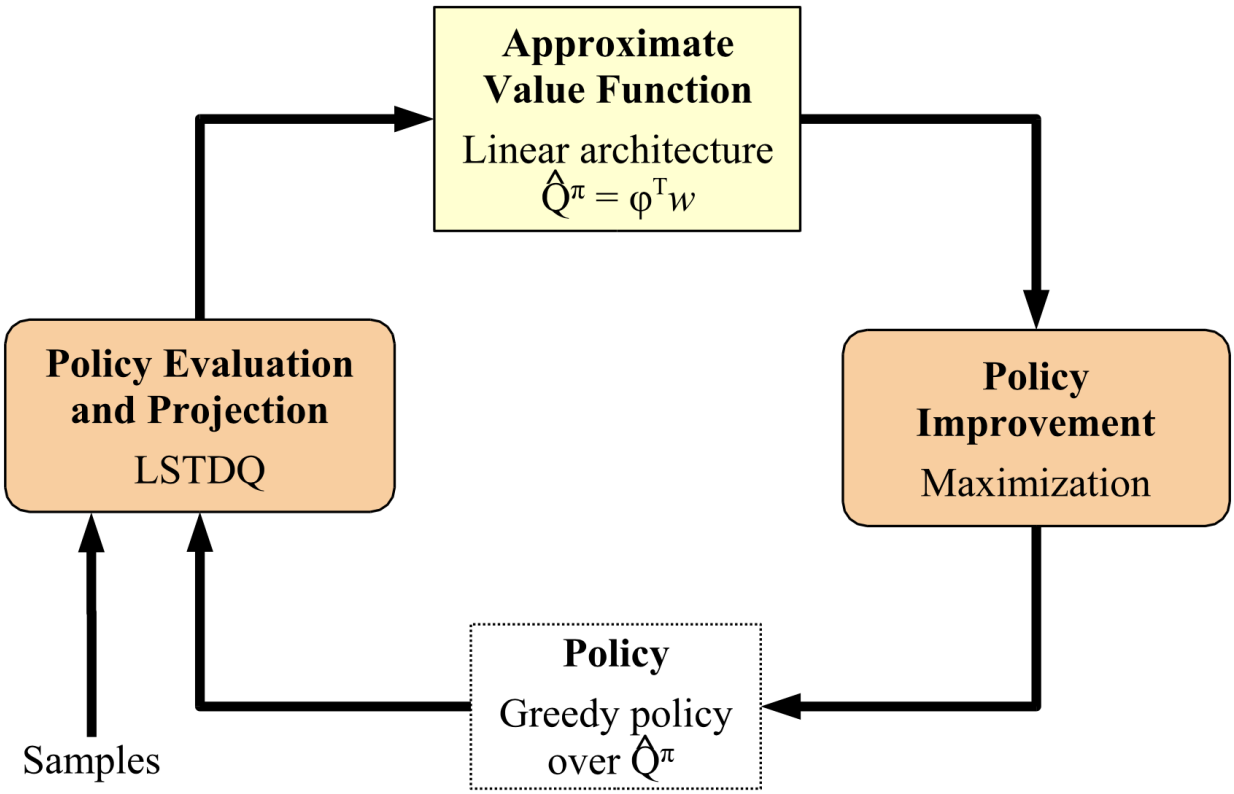


Figure 2.3 — Least-squares policy iteration algorithm block diagram. This picture was borrowed from original LSPI article [23].

The LSPI algorithm is fast if the number of features is small, and it usually converges after several iterations. In this work we use the off-policy version of the LSPI algorithm, when learning takes places after the training samples have been collected.

Algorithm 1 Least-Squares Policy Iteration

Input: Samples (s, a, r, s'), discount factor γ, initial policy π_0
Construct state-action matrix Φ from samples (s, a)
$\pi' \leftarrow \pi_0$
repeat
  $\pi \leftarrow \pi'$
  Sample actions a' for s' with policy π
  Construct state-action matrix Φ' from samples (s', a')
  $\mathbf{w}_{\Phi}^{\pi} = (\Phi^T \Phi - \gamma \Phi^T \Phi')^{-1} \Phi^T R$
  $\pi' \leftarrow$ greedy policy over $\hat{Q} = \Phi \mathbf{w}_{\Phi}^{\pi}$
until $\pi \approx \pi'$ $(\|\mathbf{w}_{\Phi}^{\pi} - \mathbf{w'}_{\Phi}^{\pi}\|_2 < \epsilon)$
return π
Here $\mathbf{w'}_{\Phi}^{\pi}$ represents the vector of weights from the previous iteration.

Глава 3

Compressed Value Iteration

3.1 Предсказательно оптимальное сжатие признаков

Theorem 1 suggests a sufficient condition for a set of features to be optimal, which lies in minimization of the model prediction prediction error Δ_Φ^π and reward prediction error Δ_R. If we manage to predict both the reward vector and encoded next features — which means that there exists some predictor matrix D^π (we refer to as *decoder*), such that $AE^\pi D^\pi = [R, A'E^\pi]$ — then we can get a perfect value function $\hat{Q}$, such that $\mathrm{BE}(\hat{Q}) = 0$ (see Theorem 3 below for justification). In practice we will minimize the Frobenius norm of the difference:

$$\|AE^\pi D^\pi - [R, A'E^\pi]\|_F \to \min_{E^\pi, D^\pi} \tag{3.1}$$

This optimization problem is a bit tricky to solve, because E^π appears inconveniently on both sides of 3.1 making it difficult to rearrange terms to solve for E^π as an optimization problem with a fixed target. Song et al. [18] proposed to solve another optimization problem and predict raw next states, instead of the encoded ones, which is easier. They introduced theory of *predictively optimal feature encoding* and proved that such shift from one optimization problem to another does not violate the equation $\mathrm{BE}(\hat{Q}) = 0$. Below we provide the key results of this theory.

Definition 2. Low-dimensional representation $\Phi = \mathcal{E}^\pi(A)$ is predictively optimal for *raw features* with respect to A, A', and π if there exists $D^\pi \in \mathbb{R}^{N \times (ml+1)}$ such that $\mathcal{E}^\pi(A)D^\pi = [R, A']$. In case of linear encoder, we have $AE^\pi D^\pi = [R, A']$.

Theorem 2. (Song et al., [18]) For any MDP $\mathcal{M}$ with predictively optimal for raw features $\Phi = AE^\pi$ with respect to A, A', and π, if the linear fixed point for Φ is $\hat{Q}^\pi$, then $\mathrm{BE}(\hat{Q}^\pi) = 0$.

Theorem 2 implies the following optimization problem:

$$\|AE^\pi D^\pi - [R, A']\|_F \to \min_{E^\pi, D^\pi} \tag{3.2}$$

which can be solved with Alternating Least-Squares (ALS) algorithm [18], which starts from random matrix D^π and alternatively updates E^π and D^π, according to the updating rules $E^\pi \leftarrow A^\dagger[R, A'](D^\pi)^\dagger$ and $D^\pi \leftarrow (AE^\pi)^\dagger[R, A']$. Here, $\dagger$ represents (truncated) Moore-Penrose pseudoinverse.

Proposed ALS for solving the optimization problem 3.2 shows good results on Inverted Pendulum and Black Jack environments with states represented as raw images, which has always seemed to be a challenge for a simple linear architecture. It outperforms the random projection method for feature selection, introduced by Ghavamzadeh [25].

However, there is one unpleasant drawback, connected with the idea to replace prediction of encoded next states with prediction of raw next states. The presented above theory can not be easily generalized to the case where non-linear encoder is used. Unfortunately, it is impossible to apply it directly to deep neural networks with linear output layer (linear Q-function approximation) and sophisticated non-linear feature extraction modules (all layers preceding output layer). This shortcoming drives us to the idea of returning back to solving 3.1 and to the development of our *compressed value iteration* algorithm.

3.2 Compressed Value Iteration: общий вид

In this section we present the central part of the thesis — *compressed value iteration* (CVI) algorithm for linear feature encoding. First of all, let us introduce one more definition and prove formally the assertion made at the beginning of the previous section.

Definition 3. Low-dimensional representation $\Phi = \mathcal{E}^\pi(A)$ is predictively optimal for *encoded features* with respect to A, A', and π if there exists $D^\pi \in \mathbb{R}^{N\times(k+1)}$ such that $\mathcal{E}^\pi(A)D^\pi = [R, \mathcal{E}_\pi(A')]$. In case of linear encoder, it is $AE^\pi D^\pi = [R, A'E^\pi]$.

Theorem 3. For any MDP $\mathcal{M}$ with predictively optimal for encoded features $\Phi = AE^\pi$ with respect to A, A', and π, if the linear fixed point for Φ is $\hat{Q}^\pi$, then $\mathrm{BE}(\hat{Q}^\pi) = 0$.

Proof. (Inspired by Song et al. [18] proof of Theorem 2.) For predictively optimal for encoded features $\Phi = AE^\pi$, there exists a decoder $D^\pi \in \mathbb{R}^{N\times(k+1)}$, such that

$$AE^\pi D^\pi \;=\; [R, A'E^\pi]. \tag{3.3}$$

We denote the first column and the last k columns of matrix D^π as D^π_r and D^π_s respectively. In such notation, we can rewrite Equation 3.3 as a system of two linear equations:

$$\begin{cases} AE^\pi D^\pi_r = R, \\ AE^\pi D^\pi_s = A'E^\pi. \end{cases}$$

Now, we pick $P^\pi_\Phi = D^\pi_s$ and $r_\Phi = D^\pi_r$, and get the following inference:

$$\Delta^\pi_\Phi = P^\pi\Phi - \Phi P^\pi_\Phi = \Phi' - AE^\pi D^\pi_s = A'E^\pi - A'E^\pi = 0,$$

$$\Delta_R = R - \Phi r_\Phi = R - AE^\pi D^\pi_r = R - R = 0,$$

From Theorem 1, we have $\mathrm{BE}(\hat{Q}_\pi) = \Delta_R + \gamma\Delta^\pi_\Phi \mathbf{w}^\pi_\Phi = 0$. □

Usually we are not able to solve Equation 3.3 directly and find exact solution in a closed-form, so we undermine the condition of strict equality and solve the optimization problem 3.1 instead.

The idea of compressed value iteration is to start with less difficult problem of predicting the vector of rewards only and then increase the number of features iteratively, until we get the optimal set of features. Initially, our set of features is empty.

At the first step we solve the following optimization problem:

$$\|AE^\pi_1 D^\pi_1 - R\|_F \to \min_{E^\pi_1, D^\pi_1}$$

The solution of this problem in terms of $X_1 = E^\pi_1 D^\pi_1$ is a least-squares solution $X_1 = A^\dagger R \in \mathbb{R}^{ml\times 1}$. As we see, the product $E^\pi_1 D^\pi_1$ is a matrix of rank 1, so we can pick $E^\pi_1 = X_1 = A^\dagger R$ and $D^\pi_1 = 1$. As a result of the first iteration we get 1-dimensional encoder E^π_1 and corresponding set of features $\Phi_1 = AE^\pi_1$, which consists of one feature only.

Now we proceed to the second step. At the previous step we have learned to predict the vector of rewards at the cost of adding one extra feature to our (empty) feature set. Initial optimization problem 3.1 implies that we need to predict not only the rewards but also the encoded features. So, now we are going to search for another encoder-decoder pair

which allows us to predict both the rewards R and encoded next states $A'E_1^\pi$ introduced on the previous iteration. Corresponding optimization problem has the following form:

$$\|AE_2^\pi D_2^\pi - [R, A'E_1^\pi]\|_F \to \min_{E_2^\pi, D_2^\pi}.$$

Similar to the first iteration, we have $X_2 = E_2^\pi D_2^\pi = A^\dagger[R, A'E_1^\pi]$ and the product $E_2^\pi D_2^\pi$ is a matrix of rank not greater than 2. Let $U_2T_2 = X_2$ be a QR-decomposition of matrix X_2. We pick $E_2^\pi = U_2$ and $D_2^\pi = T_2$ to get orthogonal encoder $E_2^\pi \in \mathbb{R}^{ml\times 2}$ and upper-diagonal decoder $D_2^\pi \in \mathbb{R}^{2\times 2}$. If decoder is a singular matrix, then we have finished and E_1^π is desired encoder. Otherwise, we claim E_2^π as the current encoder and proceed to the next step.

Before describing general form of the iteration (which is easily implied from transition $E_1^\pi \to E_2^\pi$), let us clarify why we pick $E_2^\pi = U_2$, not $E_2^\pi = U_2T_2$, or $E_2^\pi = U_2\sqrt{T_2}$, or some other matrix that satisfies $E_2^\pi D_2^\pi = Q_2R_2$, of which are infinite number. Because of linear nature of LSPI algorithm we use to find policy on the encoded features, we may not differentiate between various encoders, distinct up to a product by non-singular matrix. Orthogonal matrices, in turn, possess many good properties. Below we provide justification of this statement.

Lemma 1. For any encoder E^π and non-singular matrix Y, the resulting policies of LSPI algorithm for two sets of features $\Phi = AE^\pi$ and $\Phi_Y = \Phi Y = AE^\pi Y$ are the same.

Proof. For two different sets of features, the only difference in LSPI algorithm lies in computation of the vector of weights $\mathbf{w}$ and approximate Q-function $\hat{Q}^\pi$. Let us look at the corresponding vectors for both Φ and Φ_Y for arbitrary iteration of LSPI:

$$\mathbf{w}_\Phi^\pi = (\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi^T R,$$

$$\mathbf{w}_{\Phi_Y}^\pi = ({\Phi_Y}^T\Phi_Y - \gamma{\Phi_Y}^T\Phi_Y')^{-1}{\Phi_Y}^T R = (Y^T\Phi^T\Phi Y - \gamma Y^T\Phi^T\Phi' Y)^{-1}Y^T\Phi^T R =$$

$$= \left(Y^T[\Phi^T\Phi - \gamma\Phi^T\Phi']Y\right)^{-1}Y^T\Phi^T R = Y^{-1}(\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}(Y^T)^{-1}Y^T\Phi^T R =$$

$$= Y^{-1}(\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi^T R.$$

Here are corresponding Q-functions:

$$\hat{Q}_\Phi^\pi = \Phi(\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi^T R,$$

$$\hat{Q}_{\Phi_Y}^\pi = \Phi YY^{-1}(\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi^T R = \Phi(\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi^T R.$$

As we see, $\hat{Q}^\pi_\Phi \equiv \hat{Q}^\pi_{\Phi_Y}$ for any iteration of LSPI algorithm. Corresponding greedy policies over $\hat{Q}^\pi_\Phi$ and $\hat{Q}^\pi_{\Phi_Y}$ are equal too, as well as resulting policies.

□

We return to describing a general form of the iteration of our algorithm. Assume that we have an encoder $E^\pi_k \in \mathbb{R}^{lm \times k}$, and we want to predict the rewards R and encoded next states $\Phi' = A'E^\pi_k$. We face the following optimization problem:

$$\|AE^\pi_{k+1}D^\pi_{k+1} - [R, A'E^\pi_k]\|_F \to \min_{E^\pi_{k+1}, D^\pi_{k+1}}$$

In terms of $X_{k+1} = E^\pi_{k+1}D^\pi_{k+1}$ we have a least-squares solution $X_{k+1} = A^\dagger[R, A'E^\pi_k]$ and the product $E^\pi_{k+1}D^\pi_{k+1}$ is a matrix of rank not greater than $(k+1)$. Let $U_{k+1}T_{k+1} = X_{k+1}$ be a QR-decomposition of matrix X_{k+1}, then we pick $E^\pi_{k+1} = U_{k+1}$ and $D^\pi_{k+1} = T_{k+1}$.

Now, we know how to increase the number of features iteratively and construct the encoder E^π_k for any k, and the only thing left is to describe stopping criteria for our algorithm. Our iterative process terminates if one of the following conditions is met:

1. The number of iterations k reaches the maximally allowed one K.
2. The residual $\frac{\|AE^\pi_k D^\pi_k - [R, A'E^\pi_k]\|_F}{\|R, A'E^\pi_k\|_F}$ is below a threshold ϵ, where $D^\pi_k = (AE^\pi_k)^\dagger[R, A'E^\pi_k]$.

Aforementioned procedure of encoder training is summarized in Algorithm 2.

Algorithm 2 Iterative Encoder Training

Input: State-action matrices A and A', vector of rewards R, K, ϵ
$k \leftarrow 1$
$E_1 = A^\dagger R$
while Convergence Conditions Not Satisfied **do**
  $X_k = A^\dagger[R, A'E_{k-1}]$
  $U_k, T_k = QR(X_k)$
  $E_k = U_k$
  $k \leftarrow k + 1$
end while
return E_k
See text for termination conditions.
† is the (truncateds) Moore-Penrose pseudoinverse.

Combining the idea of iterative encoder training with LSPI, we come to the Compressed Value Iteration algorithm (Algorithm 3).

Algorithm 3 Compressed Value Iteration

Input: Samples (s, a, r, s'), discount factor γ, initial policy π_0, K, ϵ
Construct state-action matrix A from samples (s, a)
Sample actions a' for s' with policy π_0
Construct state-action matrix A' from samples (s', a')
Train encoder E^π from A, R, and A'
$\pi' \leftarrow \pi_0$
repeat
  $\pi \leftarrow \pi'$
  $\Phi = AE^\pi,\ \Phi' = A'E^\pi$
  $\mathbf{w}_\Phi^\pi = (\Phi^T\Phi - \gamma\Phi^T\Phi')^{-1}\Phi R$
  $\pi' \leftarrow$ greedy policy over $\hat{Q} = \Phi\mathbf{w}_\Phi^\pi$
  Construct state-action matrix A' from samples (s', a')
  Sample actions a' for s' with policy π'
  Train encoder E^π from A, R, and A'
until $\pi \approx \pi'$
return π

3.3 Compressed Value Iteration: подпространства Крылова

The compressed value iteration algorithm allows for multiple modifications, as we can use any other algorithms than Algorithm 2 for encoder training. In this section we discuss one modification that uses vectors from Krylov subspace generated by an $ml \times ml$ matrix $A^\dagger A'$ and a vector $A^\dagger R$ of dimensionality ml.

Let us return to the k_{th} iteration of the iterative encoder training procedure (Algorithm 2), used in general form of CVI. Lemma 1 allows us to pick any other combination of E_k^π and D_k^π, so we pick $E_k^\pi = U_k T_k$ and $D_k^\pi = I_k$, where I_k is identity matrix of rank k. At the first iteration column space of our encoder matrix consists of one vector, which is $A^\dagger R$. At the second iteration $E_2^\pi = A^\dagger[R, A'E_1^\pi] = A^\dagger[R, A'A^\dagger R]$, and column space of our encoder matrix consists of two vectors: $\{A^\dagger R,\ A^\dagger A' A^\dagger R\}$. At the k_{th} iteration column space of our encoder matrix consists of k vectors and has the following form $\{(A^\dagger A')^i A^\dagger R\}_{i=0}^{k-1}$. As we see, column space of our encoder matrix is exactly the order-k Krylov subspace generated by matrix $A^\dagger A'$ and a vector $A^\dagger R$. This is an interesting observation, as Petrik [8] has already proposed to use augmented Krylov method for features generation, however, he came to such result from other considerations.

The observation that columns of our encoder lie in Krylov subspace let us rewrite iterative encoder training procedure and speed it up, as we eliminate QR-decompositions that can be computationally expensive for large values of k. Another consideration to speed

up iterative encoder training is to precompute vector $A^\dagger R$ and matrix $A^\dagger A'$ before iterating over k, as we use them often. Described procedure is outlined as Algorithm 4.

Algorithm 4 Krylov Iterative Encoder Training

Input: State-action matrices A and A', vector of rewards R, K, ϵ
$k \leftarrow 1$
Precompute $A^\dagger R$ and $A^\dagger A'$
$e_1 = A^\dagger R$
$E = \{e_1\}$
while Convergence Conditions Not Satisfied **do**
    $e_k = A^\dagger A' e_{k-1}$
    $E = E \cup e_k$
    $k \leftarrow k + 1$
end while
$E_k = [e_1, e_2, \ldots, e_k]$
return E_k
See text for termination conditions.
† is the (truncateds) Moore-Penrose pseudoinverse.

However, Krylov methods exhibit one shortcoming. If we construct the Krylov subspace directly multiplying the forming vector by matrix every time, the obtained vectors usually soon become almost linearly dependent due to properties of power iteration. Methods, relying on Krylov subspace frequently involve some orthogonalization scheme, such as Lanczos iteration [26] for Hermitian matrices or Arnoldi iteration [27] for more general matrices. In order to preserve the win in speed, we can perform orthogonalization, for example, every 5 or 10 iterations, which brings us to some combination of Algorithm 2 and Algorithm 4 for encoder training.

Глава 4

Вычислительный эксперимент

4.1 Описание экспериментов

The goal of our computational experiments is to demonstrate practicality and effectiveness of the proposed approach. We apply our method to two popular benchmark domains in reinforcement learning — classic control Inverted Pendulum and the Atari 2600 game Pong. This section describes baseline we compare to, training data we use, and how we organize our experiments in general.

We implemented the compressed value iteration (CVI) algorithm and, for comparison, the alternating least-squares (ALS) algorithm for feature encoding in [18]. We chose the ALS algorithm as a baseline as it outperformed other methods for feature selection, such as the random projection model in Ghavamzadeh et al. [25] or OMP-TD in Painter-Wakefield et al. [28].

We concatenated vectorized gray-scaled images of several successive frames of our environments and used them as raw features. We ran a simulation with a random policy for a particular number of time steps to form the training sample. The code was written in Python and tested on a machine with a 2.9 GHz Intel Core i7 and 16 GB 2133 MHz LPDDR3 RAM. All specifications of the environments we used are described in corresponding sections.

As it is not clear in advance how many features (a number of columns in E^π) we need, we tested different models with different values of k and chose the best one based on their performance. We note that CVI algorithm is better designed for such model selection. For some maximum number of iterations K, one run of iterative encoder training procedure in CVI may output the whole sequence of encoders $E_1, E_2, \ldots, E_K$, while ALS requires K runs for the same purpose.

4.2 Перевёрнутый маятник

4.2.1 Описание среды

Inverted Pendulum is a classic control problem — a cart is moving on an infinite rail upon which an inverted pendulum is mounted. The dynamics of the environment is determined by the following system of non-linear differential equations [29]

$$\begin{cases} \dot{\theta} = \omega, \\ \dot{\omega} = \frac{g\sin(\theta) - \alpha ml\omega^2 \sin(2\theta)/2 - \alpha\cos(\theta)u}{4l/3 - \alpha ml\cos^2(\theta)}, \end{cases}$$

and has only two natural parameters — vertical angle θ and angular velocity ω of the pendulum. Here u is a force applied to the cart (control), g is the gravity constant ($g = 9.8m/s^2$), m is the mass of the pendulum ($m = 2.0$ kg), M is the mass of the cart ($M = 8.0$ kg), l is the length of the pendulum ($l = 0.5$ m), and $\alpha = 1/(m + M)$.

There are 3 discrete actions LEFT ($u = -50$ Newtons), RIGHT ($u = +50$ Newtons), and NOOP ("no operate", $u = 0$ Newtons) and continuous state space in the Inverted Pendulum environment. To be precise, a switch from pure angles and angular velocities to raw pixel images of a pendulum discretizes state space, but a number of states is too large, so we cannot apply dynamic programming to this problem and have to search for some Q-function approximation anyway.

A noise sampled from uniform distribution $\mathbf{U}[-10, 10]$ is added to each chosen action to incorporate some randomness common to real-world problems. For each time step the pendulum is balanced, there is a reward of 0, and a penalty of -1 for allowing the pendulum to fall (we consider the pendulum to be fallen if $|\theta| \geq \pi/2$). The discount factor γ in this problem is set to 0.95.

Training data consists of a desired number of trajectories with starting angle and angular velocity chosen randomly from $\mathbf{U}[-0.2, 0.2]$. Each state is represented by two successive grayscale images of the pendulum. The simulation step is set to 0.1 seconds. Each frame has 30×60 pixels, the raw state is a $30 \times 60 \times 2 = 3600$ dimensional vector, and hence the matrices A and A' have 10800 columns. Similar to Song et al. [18], we forced the angular velocity to match the change in angle per time step to ensure that the two successive frames are a Markovian state.

Figure 4.1 shows several examples of how our Inverted Pendulum looks like.

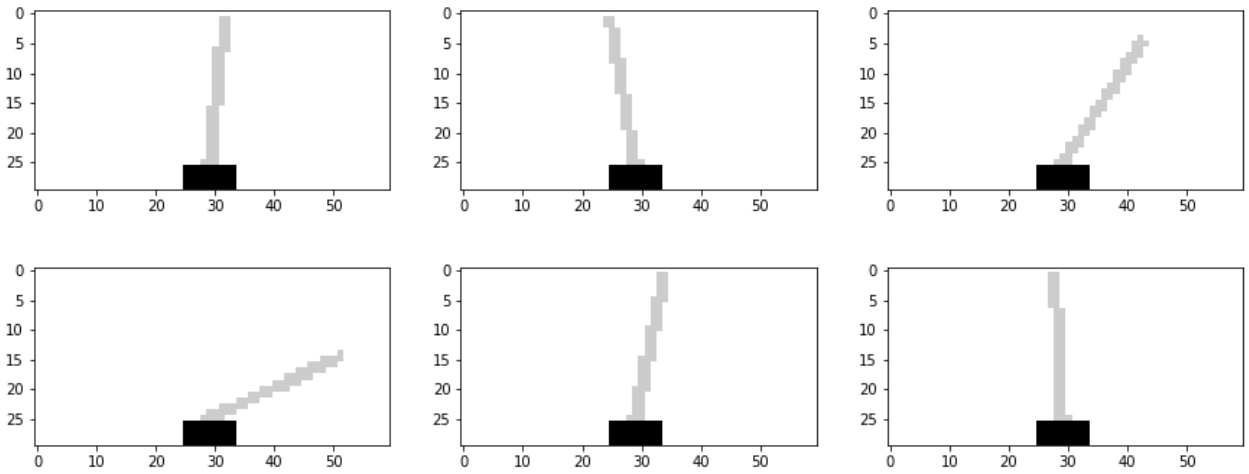
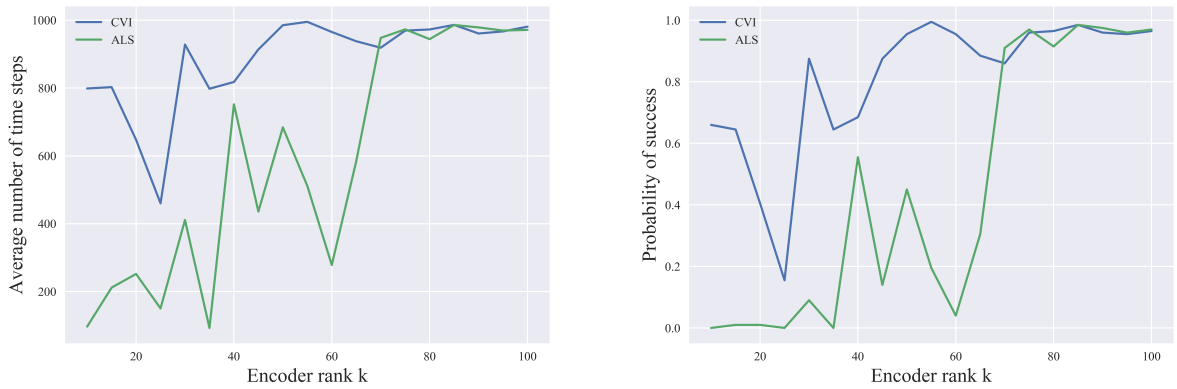


Figure 4.1 — Some images of the pendulum. Two successive images are used as the state.

4.2.2 Размерность кодировщика k

We compare the policies learned with CVI and the policies learned with ALS for a different number of features k. Each policy is evaluated 200 times to obtain the average number of balancing steps. If the pendulum does not fall after $T = 1000$ time steps, we claim it a success and terminate testing simulation. The maximum allowed number of iterations T was chosen to be much bigger than the average number of time steps before the pendulum, driven by random policy, falls. In our experiments this average number was equal to 11.5, the maximum number of balancing time steps before random policy failed was 35. In our experiments we used 1000 training episodes.

Figure 4.2 shows the results, given different numbers of encoder dimensionalities k. We see that CVI finds good sets of features that are smaller than the ones ALS finds, which seems to be logical. Instead of trying to predict big matrix $[R,\ A'] \in \mathbb{R}^{N\times(lm+1)}$ given matrix $A \in \mathbb{R}^{N\times lm}$ we are trying to predict much smaller matrix $[R,\ A'E^{\pi}] \in \mathbb{R}^{N\times(k+1)}$.



(a) Balancing steps (b) Prob. of success

Figure 4.2 — Number of balancing steps and probability of success, versus encoder rank k.

CVI predicting encoded next features finds more reasonable features than ALS predicting raw next features, as it needs fewer features to learn a good policy. CVI is also better in terms of numerical stability. ALS starts from random initialization of D^π, while CVI is fully deterministic for particular training sample.

Table 4.1 compares computational times of ALS and CVI for 1000 training episodes. We exclude time needed for the computation of the truncated pseudoinverse $A^\dagger$, as it is the same for both algorithms. It shows that ALS is a bit more computationally efficient than CVI. However, because of the randomness in the training data, the dependence between policy and encoder rank (Figure 4.2) is not monotonic and a series of encoders (with different values of k) is needed to choose the best model. Therefore, we can say that ALS requires K times more computational time to solve reinforcement learning problems if we do not have any prior knowledge of encoder dimensionality.

k	10	20	30	40	50	60	70	80	90	100
ALS	22.9	25.3	29.9	31.1	35.2	38.2	42	44.9	48.5	53.2
CVI	18.9	23.4	30.9	38.1	51.8	58.7	70	84	99	107

Table 4.1: Comparison of ALS and CVI in terms of computational time (in seconds) for different values of encoder rank k.

4.2.3 Размер обучающей выборки

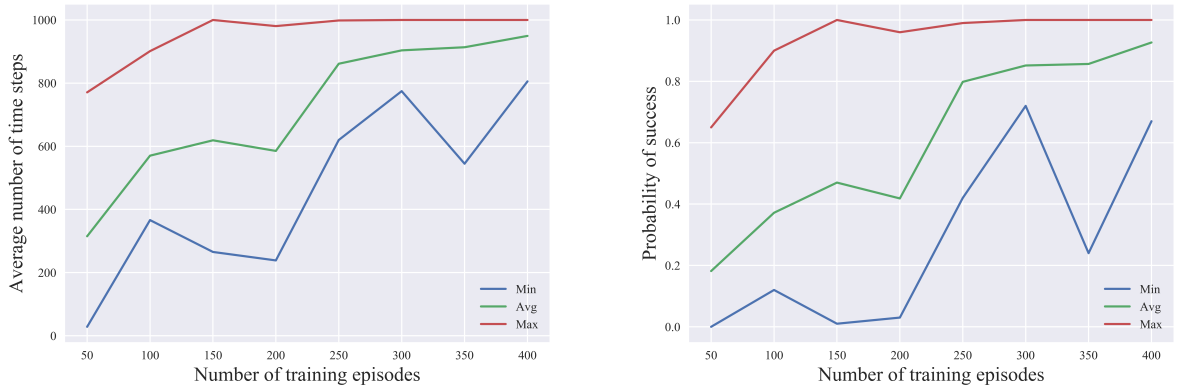
Experiment settings are the same as in the previous subsection, but now we compare the policies obtained for different number of training episodes. Table 4.2 shows the results of this experiment. We can say that CVI requires less training data than ALS to train good policy, what makes it more memory-efficient.

training episodes	100	200	300	400	500	600	700
ALS, $k = 150$	38	45	185	448	615	998	905
CVI, $k = 40$	95	426	537	574	561	966	998
CVI, $k = 70$	117	455	659	597	913	941	987

Table 4.2: Comparison of ALS and CVI in terms of average number of balancing time steps for different number of training episodes.

Figure 4.3 provides more thorough analysis of how the performance depends on the number of training episodes for CVI only. As we see, for small amounts of training data,

quality of learned policies largely depends on quality of the samples. If samples contain almost all possible configurations of the pendulum (the whole spectrum of possible states), we can learn great policy even if we have only 150 episodes (matrix A has approximately 1500 rows).



(a) Balancing steps

(b) Prob. of success

Figure 4.3 — Number of balancing steps and probability of success, versus number of training episodes for CVI averaged over 6 different samples of training data.

4.2.4 Визуализация весов кодировщика

We also created a heat map of encoder weights for encoder trained on 1000 episodes with $k = 40$, which is presented on Figure 4.4. If we define a vector of encoding weights as $\mathbf{u}_\Phi^\pi = E^\pi \mathbf{w}_\Phi^\pi$, we can conditionally divide it into six equal parts — three pairs which correspond to each action (each pair represents the environment state which consists of two successive frames).

Figure 4.4 shows that our encoder has managed to train something interesting, rather than simply picking features from the state-action matrix A. From the top images we can observe that if the pendulum tilts right, the action RIGHT will have large weights (dark red color), forcing the agent to choose this action and stabilize the pendulum. The same picture is observed for the action LEFT. The weights for the action NOOP are almost symmetric, which is also reasonable, as this action should not depend on the side to which the pendulum tilts.

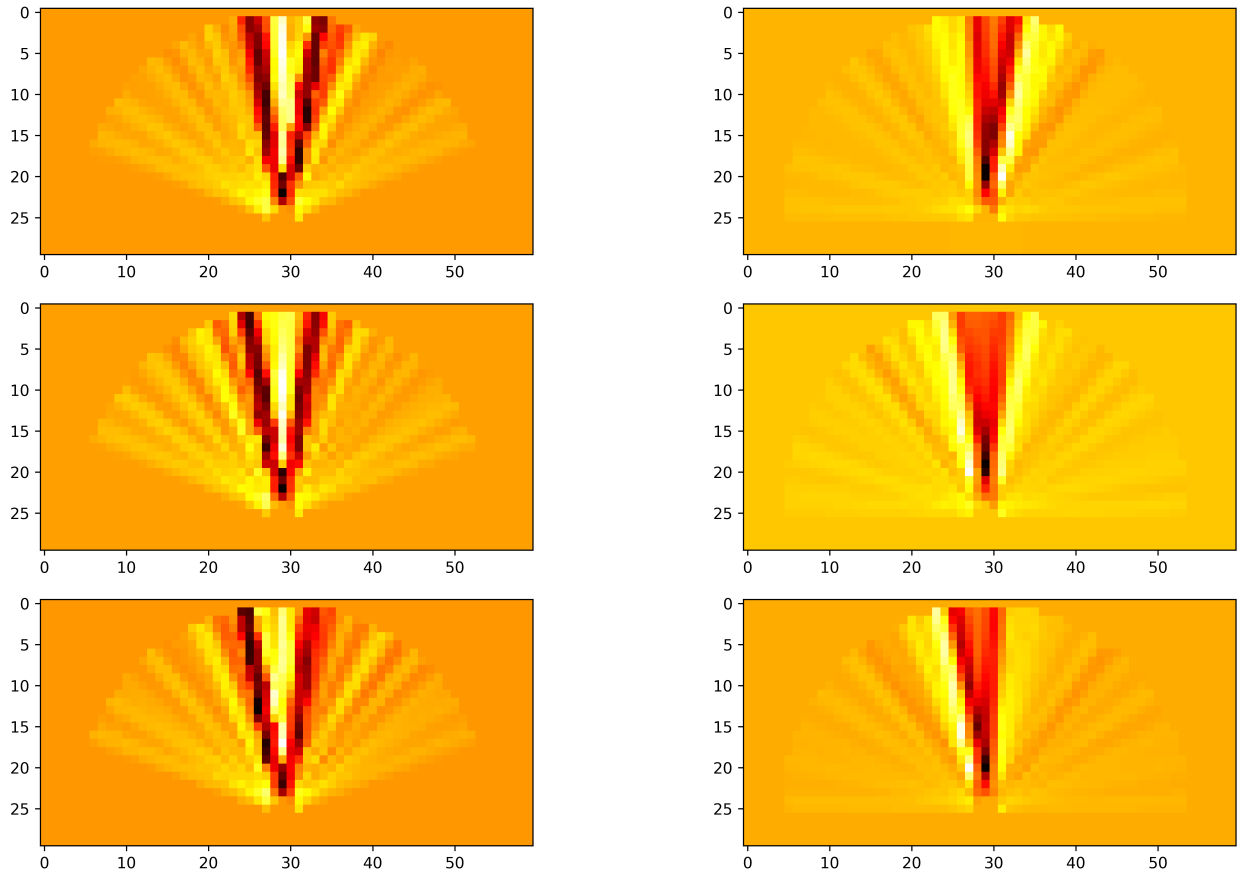


Figure 4.4 — Linear Q-function approximation weights $\mathbf{u}^{\pi}_{\Phi} = E^{\pi}\mathbf{w}^{\pi}_{\Phi}$. Each state is a pair of two succesive frames (the first is on the left, the second is on the right). Top images correspond to the action RIGHT, middle – NOOP, botton – LEFT.

4.3 Атари 2600 Пинг-Понг

4.3.1 Описание среды

Atari 2600 arcade games are another popular benchmark domain in reinforcement learning, which consists of 57 arcade games developed by the eponymous company in the 1980s. Pong is a simple two-dimensional sports game that simulating table tennis. In this game the two players controlled by either humans or computer return the ball back and forth in order to score a goal. We used a Python implementation of the Pong environment provided by OpenAI Gym[1]. Figure 4.5 shows how this game looks like.

[1] https://gym.openai.com

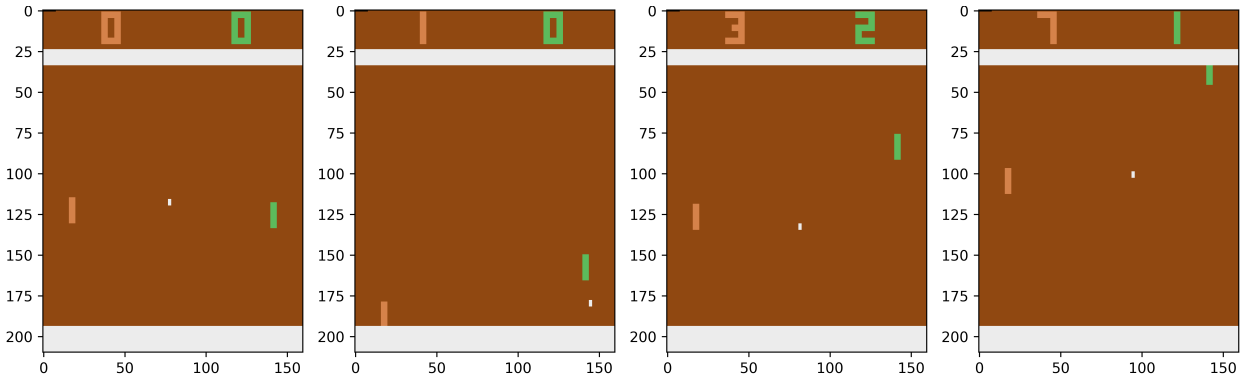


Figure 4.5 — Some images of Pong. Four successive images are used as the state.

Similar to the pendulum, there are three actions in the pong environment: LEFT (move paddle down), RIGHT (move paddle up), and NOOP (do nothing). The chosen action is applied during four successive frames and each state is represented by four successive frames of the game. To simplify the calculations we truncated the scoreboard, switched from RGB to gray scale, and reshaped images to be 80×80 pixels. Hence each raw state is a $80 \times 80 \times 4 = 25600$ dimensional vector, and state-action matrix A has 76800 columns.

For each time step of the simulation, there is a reward of 0. Our agent also gets a reward of 1 if it wins a score and -1 if it loses it. We form training data by running a simulation for the desired number of time steps, choosing actions at random.

4.3.2 Разреженные состояния

Despite our simplifications, the dimensionality of the raw states remains too big to talk about memory efficiency and fast computations. We either use a small number of training episodes (which is bad), or need to allocate tons of memory and wait hours while the model trains (which is not better). However, this problem, as well as a lot of other reinforcement learning problems, exhibits one peculiarity we may benefit from.

If we look at how the frames of Pong look like, we may notice that a huge part of the screen displays a background. After binarization of these frames (paddles and pongs become ones, pixels of the background become zeros), raw states representations appear to be very sparse. Then we incorporate actions and get state-action matrices A and A' which are even sparser. In our experiments these matrices appeared to be more than 99.5% sparse.

We implemented a modification of CVI algorithm which works with *scipy.sparse* [30][2] Python package and supports the usage of sparse matrices. This extension showed huge improvement in speed and memory, being approximately 30 times faster and 100 times more

[2] https://docs.scipy.org/doc/scipy-0.18.1/reference/sparse.html

memory-efficient, which allowed us to try a lot of different parameters configurations and learn good policies for Pong.

4.3.3 Результаты экспериментов

We conducted a series of experiments for three different settings. We played till any player won 1 score in the first setting, till both players together scored 2 points in the second, and till both players scored 3 point in the third one. To train the encoder and learn policies we used 1000 training episodes which resulted in approximately 70000, 115000, and 155000 rows of the matrix A respectively.

Because of extremely large state space and randomness in training data we ran CVI 100 times for each setting and chose the best policies for demonstration of the results. As ALS and other baselines didn't manage to learn policies much better than random policy we compare our results with random policy. The results are presented in Table 4.3.

1-score game		2-score game		3-score game	
Random	CVI, $k = 50$	Random	CVI, $k = 35$	Random	CVI, $k = 50$
$<$3.5%	46%	$<$2.5%	37%	$<$1.5%	11%

Table 4.3: Comparison of policies produced by CVI and random policies in terms of the percentage of scores won.

Unfortunately, we didn't manage to learn playing Pong as well as Inverted Pendulum. However, the obtained results look quite promising. We showed that high-dimensional states representations we used can be effectively supplanted by small sets of several dozens of features. Although deep neural networks may learn good policies for Pong, it is not fair to compare them with CVI. DQN requires several days of training, much greater computational resources, and tons of training data, while our algorithm finds reasonable policies in several minutes and uses relatively small training samples.

Глава 5

Обсуждение

5.1 Заключение

In this work, we have presented *compressed value iteration* — a linear encoding method for solving reinforcement learning problems, which searches for a low-dimensional representation of state-action pairs. Although our method uses rather simple linear architecture and can not compete with sophisticated deep neural networks, it produces reasonable results and may provide some insights on how to search for good feature representations. The proposed method is fast, as it needs only several minutes of computations on a laptop with regular CPU to learn rather good policies, while neural networks train for several days and require expensive GPUs. It is also very data- and memory-efficient as it needs relatively small amount of training data and RAM.

The proposed method can be potentially applied to real-world problems with limited data availability which require fast and working solution. For each particular problem CVI may be adjusted to benefit from problem specification and used data format. Our algorithm may be also extended to a non-linear case, providing many directions of the future work.

5.2 Дальнейшая работа

We should move on to solving the optimization problem

$$\|\mathcal{E}^\pi(A)D^\pi - [R, \mathcal{E}^\pi(A')]\|_F \to \min_{\mathcal{E}^\pi, D^\pi},$$

instead of 3.1, where $\mathcal{E}^\pi$ is some non-linear transformation to low-dimensional encoded feature space, to find an optimal set of features.

It may be difficult to train a neural network to discover an optimal set of features using gradient descent for a variety of reasons. The nature of the optimization problem (with $\mathcal{E}^\pi$ on both sides of the equation) makes the optimization more challenging than typical supervised learning tasks. (Indeed, our initial experiments with the linear case suggest that this is the case.) To address this, we will adopt ideas from deep neural networks and from linear value function approximation theory.

Curriculum learning [31] is a technique for neural networks that trains networks on easier problems before training them on harder problems. The incremental nature of CVI could combine naturally with the curriculum learning approach to neural network training. For example, we would first train an encoder network to produce a single feature and linear decoder that predicts the immediate reward. At each iteration, we would check if the decoder can predict the reward and all next feature values. If it cannot, then we would train the network to produce a new feature to predict the reward and existing feature set.

Благодарности

The author would like to thank Ron Parr, a professor and department chair of Computer Science Department at Duke University, for collaboration, mentoring, and support in the research which has developed into this work. The author thanks Skoltech for providing necessary funding to accomplish the four-month research trip to Duke University during fall semester. The author wish to thank Ivan Oseledets, an associate professor at Skoltech, for wise supervising and joint fruitful work during the two years of author's Master's studies at Skoltech. Finally, the author thanks the whole group of Scientific Computing at Skoltech for valuable pieces of advice and discussions.

Список литературы

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

[2] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[3] Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

[4] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[5] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[6] Sridhar Mahadevan and Mauro Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(Oct):2169–2231, 2007.

[7] Ronald Parr, Christopher Painter-Wakefield, Lihong Li, and Michael Littman. Analyzing feature generation for value-function approximation. In *Proceedings of the 24th international conference on Machine learning*, pages 737–744. ACM, 2007.

[8] Marek Petrik. An analysis of laplacian methods for value function approximation in mdps. In *IJCAI*, pages 2574–2579, 2007.

[9] Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 752–759. ACM, 2008.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] Ahmed Hussain Qureshi, Yutaka Nakamura, Yuichiro Yoshikawa, and Hiroshi Ishiguro. Robot gains social intelligence through multimodal deep reinforcement learning. In *Humanoid Robots (Humanoids), 2016 IEEE-RAS 16th International Conference on*, pages 745–751. IEEE, 2016.

[13] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. *arXiv preprint arXiv:1702.03037*, 2017.

[14] Marcus Olivecrona, Thomas Blaschke, Ola Engkvist, and Hongming Chen. Molecular de novo design through deep reinforcement learning. *arXiv preprint arXiv:1704.07555*, 2017.

[15] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[16] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. *arXiv preprint arXiv:1703.01988*, 2017.

[17] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

[18] Zhao Song, Ronald E Parr, Xuejun Liao, and Lawrence Carin. Linear feature encoding for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4224–4232, 2016.

[19] Ronald J Williams and Leemon C Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Citeseer, 1993.

[20] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[21] Steven J Bradtke and Andrew G Barto. Linear least-squares algorithms for temporal difference learning. *Machine learning*, 22(1-3):33–57, 1996.

[22] Huizhen Yu and Dimitri P Bertsekas. Convergence results for some temporal difference methods based on least squares. *IEEE Transactions on Automatic Control*, 54(7):1515–1531, 2009.

[23] Michail G Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of machine learning research*, 4(Dec):1107–1149, 2003.

[24] Ronald A Howard. Dynamic programming and markov processes.. 1960.

[25] Mohammad Ghavamzadeh, Alessandro Lazaric, Odalric Maillard, and Rémi Munos. Lstd with random projections. In *Advances in Neural Information Processing Systems*, pages 721–729, 2010.

[26] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[27] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[28] Christopher Painter-Wakefield and Ronald Parr. Greedy algorithms for sparse reinforcement learning. *arXiv preprint arXiv:1206.6485*, 2012.

[29] Robert H Cannon. *Dynamics of physical systems*. Courier Corporation, 2003.

[30] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[31] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.