Spring 2019

Feature Selection by Singular Value Decomposition for Reinforcement Learning

Bahram Behzadian
University of New Hampshire, Durham

Feature Selection by Singular Value Decomposition for Reinforcement Learning

BY

Bahram Behzadian

MASTER'S THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

May, 2019

This thesis was examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science by:

Thesis Director, Marek Petrik, Assistant Professor
Department of Computer Science

Wheeler Ruml, Professor
Department of Computer Science

Laura Dietz, Assistant Professor
Department of Computer Science

Ernst Linder, Professor
Department of Mathematics and Statistics

On April 11, 2019

Approval signatures are on file with the University of New Hampshire Graduate School.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Marek Petrik of the Department of Computer Science at the University of New Hampshire. Without his enthusiastic support and input, this work could not have been successfully completed.

TABLE OF CONTENTS

ABSTRACT

Feature Selection by Singular Value Decomposition

for Reinforcement Learning

by

Bahram Behzadian

University of New Hampshire, May, 2019

Solving reinforcement learning problems using value function approximation requires having good state features, but constructing them manually is often difficult or impossible. We propose Fast Feature Selection (FFS), a new method for automatically constructing good features in problems with high-dimensional state spaces but low-rank dynamics. Such problems are common when, for example, controlling simple dynamic systems using direct visual observations with states represented by raw images. FFS relies on domain samples and singular value decomposition to construct features that can be used to approximate the optimal value function well. Compared with earlier methods, such as LFD, FFS is simpler and enjoys better theoretical performance guarantees. Our experimental results show that our approach is also more stable, computes better solutions, and can be faster when compared with prior work.

CHAPTER 1

Introduction

Reinforcement Learning (RL) involves the problem of automatic decision making from experience (Sutton & Barto, 2018). As an agent travels through the state space, it needs to identify and navigate to states with higher rewards. By definition, a value function measures the expected rewards attainable from each state and makes it possible to pick good actions quickly. While value functions can be computed efficiently in small Markov decision processes, computing them in problems with (infinitely) many states is challenging (Puterman, 2014).

Most RL approaches for solving problems with large state spaces rely on some form of value function approximation (Sutton & Barto, 2018; Szepesvári, 2010). Linear Value Function Approximation (VFA) is one of the most frequent and most straightforward approximation methods. It represents the value function as a linear combination of state features. Each feature should represent some essential characteristics of the state space. As with linear regression, it is necessary to use only features that are relevant. Using too many features can cause overfitting and prohibitive computational complexity.

Linear methods are interpretable (easy to understand), easy to implement, and easy to analyze in comparison to black box methods such as artificial neural networks (Lagoudakis & Parr, 2003). Significant effort has been devoted to automating feature construction for linear VFA. Examples of methods that construct features—or select them from a large set—include proto-value functions (Mahadevan & Maggioni, 2007), diffusion wavelets (Mahadevan & Maggioni, 2006), Krylov bases (Petrik, 2007), BEBF (Parr, Li, Taylor, Painter-Wakefield, & Littman, 2008), ℓ_1-regularized

TD (Kolter & Ng, 2009), and ℓ_1-regularized approximate linear programming (Petrik, Taylor, Parr, & Zilberstein, 2010).

1.1 Motivation

Although linear value function approximation is less powerful than modern deep reinforcement learning, it is still important in many domains. In particular, linear models are interpretable and need relatively few samples to compute a value function reliably. Features also make it possible to encode prior knowledge conveniently. Finally, the last layer in deep neural networks, used for reinforcement learning, often calculates a linear combination of the underlying neural network features (Song, Parr, Liao, & Carin, 2016).

Our goal is to compute a small set of useful features from the transition matrix and the reward function. However, such parameters are practically unavailable. Therefore we are compelled to estimate the transition matrix and reward function from samples. Using singular value decomposition over such estimation guarantees that the extracted features are linearly independent (Lagoudakis & Parr, 2003).

1.2 Challenges and Contribution

Recently researchers worked on value function approximation (VFA) problems with high dimensional state space both with deep neural nets (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, & Riedmiller, 2013; Mnih, Badia, Mirza, Graves, Lillicrap, Harley, Silver, & Kavukcuoglu, 2016; Oh, Guo, Lee, Lewis, & Singh, 2015) and linear architectures (Song et al., 2016). In this thesis, we propose Fast Feature Selection (FFS), a new feature selection method that uses Singular Value Decomposition (SVD) to compute a *low-rank* approximation of the transition matrix and approximate the reward predictor from the raw input data.

As with all data-driven methods, feature construction (or selection) must make some simplifying assumptions about the problem structure, which can be used to reduce the number of samples needed. We assume the matrix of transition probabilities can be closely approximated using a *low-rank* matrix. Using the low-rank approximation has led to significant successes in several machine learning domains, including collaborative filtering (Murphy, 2012), natural language word embeddings (Li, Zhu, & Miao, 2015), reinforcement learning (Ong, 2015; Cheng, Asamov, & Powell, 2018), and Markov chains (Rendle, Freudenthaler, & Schmidt-Thieme, 2010). Surprisingly, given the simplicity and popularity of SVD in other areas of machine learning, it has not been previously used for feature compression in reinforcement learning. Our theoretical analysis also leads to new approximation error bounds that translate the singular values of the transition matrix to the approximation error.

A significant limitation of linear value function approximation is that it requires *good* features to approximate the optimal value function well. This is difficult to achieve since good a priori estimates of the optimal value functions are rarely available. Considerable effort has, therefore, been dedicated to methods that can automatically construct useful features or at least select them from a larger set. Examples of such methods include proto-value functions (Mahadevan & Maggioni, 2007), diffusion wavelets (Mahadevan & Maggioni, 2006), Krylov bases (Petrik, 2007), BEBF (Parr, Painter-Wakefield, Li, & Littman, 2007), L_1-regularized TD (Kolter & Ng, 2009), and L_1-regularized ALP (Petrik et al., 2010), and supervised sparse coding (Le, Kumaraswamy, & White, 2017).

This work focuses on *batch reinforcement learning* (Lange, Gabel, & Riedmiller, 2012). In batch RL, all domain samples are provided in advance as a batch, and it is impossible or difficult to gather additional samples. This is common in many practical domains. In medical applications, for example, it is usually too dangerous and expensive to run additional tests, and in ecological applications, it may take an entire growing season to obtain a new batch of samples.

Another assumption that we make is that there are many raw state features available. For example, in video games, raw features can be derived directly from the pictorial representation of the game. Because the raw features are too numerous, using them directly would lead to prohibitive sample complexity. Our goal is to compute a small set of useful features as a linear combination of raw features.

FFS is easy to implement, and our experiments show that it is faster, more robust, and results in a lower Bellman error compared to similar methods. We also establish new theoretical guarantees on the Bellman error as a function of the spectrum of transition matrices.

1.3 Outline

The remainder of the thesis proceeds as follows. Chapter 2 summarizes the framework of linear value function approximation and the relevant Markov decision process properties. In Chapter 3, we review common feature selection and construction methods. Chapter 4 describes our new Fast Feature Selection method (FFS) and analyzes its approximation error and then compares it with other feature construction algorithms. Finally, the empirical evaluation in Chapter 5 indicates that FFS is faster and more robust than competing methods. We discuss future work and conclude this thesis on Chapter 6.

CHAPTER 2

Linear Value Function Approximation

In reinforcement learning (RL) we are interested in automatic decision making under uncertainty using the history of the agent. This decision-making problem can be formulated in an Markov Decision process (MDP) framework. Such environments are assumed to be fully observable. The current state is Markovian (memoryless) and completely characterizes the state of process. In this chapter, we briefly introduce reinforcement learning problems and their components. Then we explain the principle of MDPs and describe some dynamic programming methods that solve them. Finally, we discuss the concept of value function approximation and linear approximation methods.

Reinforcement learning is different from supervised learning since no supervision determines the right or wrong behavior. An RL agent only relies on a reward signal from the environment. Such feedbacks are delayed and not instantaneous so the result of a good action cannot be recognized immediately. Time plays a vital role in RL because the agent is processing data sequentially, which means the training dataset is neither independent nor identically distributed (Sutton & Barto, 2018). Most importantly, any action taken by an agent may influence the environment and change the subsequent data it receives.

A reward signal r_t is a feedback from the environment as a scalar real number that indicates how well the agent is performing at each time step t. In reinforcement learning, the agent's goal is defined by a hypothesis that all goals can be represented by the maximization of the expected cumulative reward (Sutton & Barto, 2018). Therefore, an RL agent has to behave in a certain way to extract the maximum reward from the environment.

2.1 Components of Reinforcement Learning

In this section, we describe the essential components of RL and explain their relationships. We introduce the mathematical framework for MDPs, which is the critical element in solving a reinforcement learning problem.

2.1.1 Agent and Environment

In RL problems, an agent is the brain of the system, which, in each time step t, makes a decision and executes an action a_t. Meanwhile, it receives an observation o_t and a scalar reward r_t from the environment. We can describe the environment as everything else in the world. At each time step t, the environment receives an action a_t from the agent and emits an observation o_{t+1} and a scalar reward r_{t+1}. The time increments in discrete steps.

2.1.2 History and State

All observations, rewards, and actions from the first time step until the current time t are referred to as the history and are denoted with $H_t = \{a_1, o_1, r_1, \ldots, a_{t-1}, o_t, r_t\}$. What happens in the future depends on the history and the agent's actions. The environment determines the observations and rewards. A state is the information used to determine what happens next. The state is a sufficient statistic of the history and can be formally written as a function of history $S_t = f(H_t)$ (Sutton & Barto, 2018).

A state can be regarded from three different points of view. From the environment point of view, the state S_t^e is whatever data the environment uses to pick the next observation and reward. The environment state is not usually apparent to the agent. S_t^e may contain much irrelevant information that even if visible to the agent, they do not help for a better decision making.

From the agent's internal representation, the state S_t^a is whatever information the agent uses to pick the next action. An agent's state contains the information that is used by reinforcement learning algorithms and can be any function of history.

Finally, the information state S_t also known as Markov state which represents the useful information from the history. A state is Markovian if and only if $\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$. In another word, the future is independent of the past given the present information state S_t. Therefore, once we have the current state, we can throw away the history and rely on the current state for the future decision making. It is important to note that the environment state and the history are Markov state as well.

2.1.3 Agent's Components

The major components of an RL agent may include one or more of the following:

- *Model*: It describes the agent's representation of the environment. A model predicts what the environment might do next. A model predicts the next state with a transition matrix P and next immediate reward R given the current state $S_t = s$.
- *Policy*: Agent behavior is expressed as a function π known as policy that maps any observed state to an action or a probability distribution over actions. When the policy is deterministic the return of policy for each state is a single action $a = \pi(s)$. A randomized policy return a probability distribution over actions: $\mathbb{P}[A_t = a|S_t = s] = \pi(a|s)$.
- *Value Function*: It maps any state or action to a real number. The value of a state somehow predicts the expected cumulative future rewards that could be achieved from that particular state onwards. The state values are used to evaluate the goodness or badness of being in each state.

2.2 Mathematical Framework

Markov decision processes (MDPs) formally describe an environment for reinforcement learning, where the environment is fully observable. An environment is fully observable when the agent directly observes the environment state in a way that $o_t = S_t^a = S_t^e$. Almost all RL problems can be described as MDPs. For example, optimal control primarily deals with continuous MDPs. Partially observable Markov decision process problems can be converted to MDPs. Bandits are MDPs with a single state.

All states in MDPs are Markov states, which means the future is independent of the past given the present state, and the state holds all necessary information in the history. In this section, we start with explaining the Markov process environments, and we will continue with Markov reward process and Markov decision process.

2.2.1 Markov Process

A Markov process (a.k.a Markov chain) is a memoryless random process with a sequence of states $(S_1, S_2, \ldots)$ with the Markov property. Such processes can be defined as a tuple $\langle \mathcal{S}, P \rangle$, where S is a finite set of Markov states and P is the state transition probability matrix. So we have $p_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s]$.

2.2.2 Markov Reward Process

A Markov reward process (MRP) is a Markov process $(S_1, R_2, S_2, \ldots)$ with rewards for each state. MRP is a tuple $\langle \mathcal{S}, P, R, \gamma \rangle$. $\mathcal{S}$ and P are defined exactly like a Markov process, but $R : \mathcal{S} \rightarrow \mathbb{R}$ is a reward function and can be shown as $r_s = \mathbb{E}[R_{t+1} | S_t = s]$. The discount factor $\gamma \in [0, 1]$ controls the agents preference over short-term rewards and long-term rewards. The return G_t is the total discounted reward from a time-step t for any given sample.

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

When γ is close to zero, we say the agent is myopic or short-sighted and prefers to collect rewards as fast as possible. On the other hand, when γ is close to one, we say the agent is far-sighted and tries to collect maximum cumulative rewards possible even though it may take much more time than a short-sighted agent. The discount factor adjusts the value of the rewards in future for the following reasons (Sutton & Barto, 2018; Puterman, 2014):

- It is mathematically convenient to discount the future rewards since we can estimate a bounded return for every state.
- To avoid infinite reward loops in cyclic Markov processes.
- The model may not account for uncertainty in the future very well, so it is less likely to gain future rewards compare to immediate rewards.

As outlined earlier, the value function $v(s)$ represents the long-term value of each state s. In MRPs, the state value function is the expected return starting from any state s.

$$v(s) = \mathbb{E}[G_t | S_t = s] \ .$$

The value function can be decomposed into two parts: The immediate reward R_{t+1} and the discounted value of successor state $\gamma v(S_{t+1})$. The Bellman equation for MRPs is base on this idea of decomposition of the value function (Puterman, 2014). Indeed, the value function must satisfy:

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] = \tag{2.1}$$

$$= r_s + \gamma \sum_{s' \in \mathcal{S}} p_{ss'} v(s') \ . \tag{2.2}$$

The Bellman equation can also be stated using matrices as follows:

$$\boldsymbol{v} = \boldsymbol{r} + \gamma P \boldsymbol{v} \ , \tag{2.3}$$

where $\boldsymbol{v}$ is a column vector with one entry per state[1]. Since (2.3) is a linear equation, we can solve it directly with $\boldsymbol{v} = (\mathbf{I} - \gamma P)^{-1}\boldsymbol{r}$. However, the best achieved computational complexity of inverting a matrix is $O(n^{2.37})$ (Coppersmith & Winograd, 1990), so the closed form solution is not efficient for large state spaces. There exist many iterative solutions for large MRPs such as dynamic programming and Monte-Carlo evaluation (Sutton & Barto, 2018).

2.2.3 Markov Decision Process

A Markov decision process (MDP) is a Markov reward process with decisions. It is an environment in which all states are Markov. An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ in which $\mathcal{A}$ is a finite set of actions. The transition probability and reward function also depend on action taken from each state:

$$p^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \tag{2.4}$$

$$r^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.5}$$

The solution to an MDP is a policy. A policy π is a distribution over actions conditioned on states $\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$. A policy in MDPs fully defines the behavior of an agent and it depends only on the current state instead of the full history. Every Markov decision process with a policy π can be converted to a Markov process $\langle \mathcal{S}, P^\pi \rangle$ if we only sample the state sequence $(S_1, S_2, \ldots)$ that follows the policy π. The same rule applies for an MRP $\langle \mathcal{S}, P^\pi, R^\pi, \gamma \rangle$ when we only sample the state and reward sequence $(S_1, R_2, S_2, R_3, \ldots)$ where:

$$p^\pi_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a|s) p^a_{ss'} \tag{2.6}$$

$$r^\pi_s = \sum_{a \in \mathcal{A}} \pi(a|s) r^a_{ss'} \tag{2.7}$$

In an MDP environment, value function can be defined as state-value function $v^\pi(s)$ and action-value funcion $q^\pi(s, a)$. The state-value function of an MDP is the expected return starting from

[1]For vectors, we set the variable in boldface.

state s, and then following policy π, $v^{\pi}(s) = \mathbb{E}^{\pi}[G_t|S_t = s]$. The action-value function $q^{\pi}(s,a) = \mathbb{E}^{\pi}[G_t|S_t = s, A_t = a]$.

Bellman Equation

Both state-value function and the action-value function can be decomposed into immediate reward and discounted value of the next state (Sutton & Barto, 2018):

$$v^{\pi}(s) = \mathbb{E}^{\pi}[R_{t+1} + \gamma v^{\pi}(S_{t+1})|S_t = s] \tag{2.8}$$

$$q^{\pi}(s,a) = \mathbb{E}^{\pi}[R_{t+1} + \gamma q^{\pi}(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. \tag{2.9}$$

We can express Bellman expectation equation in a concise form using induced MRP with $\boldsymbol{v}^{\pi} = \boldsymbol{r}^{\pi} + \gamma P^{\pi}\boldsymbol{v}^{\pi}$, therefore the direct solution would be $\boldsymbol{v}^{\pi} = (\mathbf{I} - \gamma P^{\pi})^{-1}\boldsymbol{r}^{\pi}$.

Bellman Optimality Equation

The objective of an MDP is to find an optimal policy, π^*, such that $v_{\pi^*}(s) = v^*(s) \geq v^{\pi}(s)$ for all $s \in \mathcal{S}$ and for all policies Π. The optimal state-value function $v^*(s)$ is the maximum value function over all policies:

$$v^*(s) = \max_{\pi} v^{\pi}(s).$$

The optimal action-value function $q^*(s,a)$ is the maximum action-value function over all policies.

$$q^*(s,a) = \max_{\pi} q^{\pi}(s,a).$$

An MDP is considered solved when we know the optimal value function.

2.3 Value Function Approximation using Linear Architectures

A value function in contexts RL maps every state to a real number that represents the value of input state. The value function measures the usefulness of being in that state. In small MDPs where the number of states is not so large, we can store the value of each state in a table; the agent can look it up while traversing the environment. Such representation of value function is also known as the tabular representation of the value function.

With a large state space, we cannot rely on tabular solution methods since the number of states can be so large that makes storing the value function as a table impossible. The difficulty with large state spaces is not only the memory storage but also the generalization (Sutton & Barto, 2018). Generalization concerns with how the experience of a subset of data can be generalized to a much larger state space. In RL, the generalization can be made using function approximation. With function approximation, we can collect samples for a subset of states and then generalize from them to build an approximation of the entire function.

Although with function approximation, we can generalize the state space, it is hard to manage, control, and understand. Some problems may arise with function approximation such as:

- *Non-stationarity:* learning a value function based on a subset of the data is useful only if we assume that the dynamics of the environment remain stationary. In other words, an optimal learned policy remains optimal if the parameters of the environment such as transition probabilities and reward function remain unchanging. However, generalization might fail if the world's parameters are non-stationary (Da Silva, Basso, Bazzan, & Engel, 2006).
- *Dynamic learning:* Neural nets and statistical learning approaches all assume a static training set over which many passes are made. In RL, learning is performed by collecting data via interaction with environments so learning a value function requires to be able to deal with incrementally acquired data (Bertsekas & Tsitsiklis, 1996).
- *Bootstrapping:* Semi-gradient (a.k.a bootstrapping) approaches do not converge as robustly as gradient methods, but they work reliably in the linear case (Sutton & Barto, 2018).
- *Delayed target:* The feedback signal in an RL setting does not always reflects the quality of

recent activity of the agent. Therefore, the difference between good and bad actions might remain undetected if we rely on limited samples.

In reinforcement learning, function approximation techniques are applied to estimate the value function. The idea is to approximate a value function from simulated data from a known policy. This value function is represented by parameters that we learn through sampling from experience and it is usually denoted as $\hat{v}^{\pi}(s, \boldsymbol{w})$ or in the case of state-action value function we have $\hat{q}^{\pi} = (s, a; \boldsymbol{w})$. In a linear function approximation, the vector $\boldsymbol{w} \in \mathbb{R}^k$ represent the features' weights. If we approximate the value function using a multilayer artificial neural network, $\boldsymbol{w}$ stands for the vector of connection of layers. Typically, the number of parameters k is smaller than the number of states $\mathcal{S}$, so each parameter corresponds to many more states at the same time. Whenever a single state is updated, it affects many other states. Therefore, value function approximation theory raises the idea of generalization in learning and make approximation methods more powerful but harder to understand (Sutton & Barto, 2018). In RL systems it is necessary to consider that learning occurs while interacting with the environment, so the agent should be able to learn from incrementally collected data. The agent also needs to account for functions that can handle non-stationary target function.

Function approximation methods are applied where the value function cannot be computed precisely. Most Markov Decision Process (MDP) in real-world have continuous or high-dimensional state spaces, which demand function approximation to work efficiently. These methods reduce the state space of the problem to a lower dimension. In the following section, we describe one of the linear methods, which are the most common approach for value function approximation and are easy to analyze theoretically.

2.3.1 Linear Value Function Approximation

In this section, we summarize the background on linear value function approximation. Value function approximation becomes necessary in MDPs with large state spaces. The value function $\boldsymbol{v}^{\pi}$

can then be approximated by a linear combination of features $\phi_1, \ldots, \phi_k \in \mathbb{R}^{|\mathcal{S}|}$, which are vectors over states. Using the vector notation, an approximate value function $\hat{\boldsymbol{v}}^\pi$ can be expressed as:

$$\hat{\boldsymbol{v}}^\pi = \Phi \boldsymbol{w} \ ,$$

for some vector $\boldsymbol{w}$ of scalar weights that quantify the importance of features. It is well-known that the value function $\boldsymbol{v}^\pi$ for a policy π must satisfy the Bellman optimality condition (e.g., (Puterman, 2014)):

$$\boldsymbol{v}^\pi = \boldsymbol{r}^\pi + \gamma P^\pi \boldsymbol{v}^\pi \ , \tag{2.10}$$

where P^π and $\boldsymbol{r}^\pi$ are the matrix of transition probabilities and the vector of rewards, respectively, for the policy π. The feature matrix Φ is of dimensions $|\mathcal{S}| \times k$; $(k \ll |\mathcal{S}|)$ the columns of this matrix are the features ϕ_i. $\Phi = \{\phi_1, \ldots, \phi_k\}$ in this equation is a set of linearly independent basis functions of the state. The weights vector $\boldsymbol{w} = \{w_1, \ldots, w_k\}$ is a set of scalars that need to be estimated through the process of approximation. The vector $\boldsymbol{w}$ quantify the importance of features. Here, Φ is the feature matrix of dimensions $|\mathcal{S}| \times k$; the columns of this matrix are the features ϕ_i. Because of its simplicity and interpretability, linear function approximation techniques are one of the most common techniques for RL.

One of the advantages of basis functions is mitigating the curse of dimensionality of these practical problems (Keller, Mannor, & Precup, 2006). The produced basis functions Φ still describe the high dimensional state space but with some information loss. Numerous algorithms such as linear TD(λ), LSTD, and LSPE for computing linear value function approximation have been proposed (Sutton & Barto, 2018; Lagoudakis & Parr, 2003; Szepesvári, 2010). We focus on fixed-point methods that compute the *unique* vector of weights $\boldsymbol{w}_\Phi^\pi$ that satisfy the projected Bellman equation (2.10):

$$\boldsymbol{w}_\Phi^\pi = \Phi^+ (\boldsymbol{r}^\pi + \gamma P^\pi \Phi \boldsymbol{w}_\Phi^\pi) \ , \tag{2.11}$$

where Φ^+ denotes the Moore-Penrose pseudo-inverse of Φ (e.g., Golub and Van Loan (2013)). This equation follows by applying the orthogonal projection operator $\Phi(\Phi^\top \Phi)^{-1} \Phi^\top$ to both sides of

(2.10).

In order to construct good features, it is essential to be able to determine their quality in terms of whether they can express a good approximate value function. The standard bound on the performance loss of a policy computed using, for example, approximate policy iteration can be bounded as a function of the Bellman error (e.g., Williams and Baird (1993)). To motivate FFS, we use the following result that shows that the Bellman error can be decomposed into the error in 1) the compressed rewards, and in 2) the compressed transition probabilities.

Theorem 1 ((Song et al., 2016))**.** *Given a policy π and features Φ, the Bellman error of a value function $v = \Phi \boldsymbol{w}_\Phi^\pi$ satisfies:*

$$\mathrm{BE}_\Phi = \underbrace{(\boldsymbol{r}^\pi - \Phi \boldsymbol{r}_\Phi^\pi)}_{\Delta_r^\pi} + \gamma \underbrace{(P^\pi \Phi - \Phi P_\Phi^\pi)}_{\Delta_P^\pi} \boldsymbol{w}_\Phi^\pi \ .$$

We seek to construct a basis that minimizes both $\|\Delta_r^\pi\|_2$ and $\|\Delta_P^\pi\|_2$. These terms can be used to bound the L_2 norm of Bellman error as:

$$\begin{aligned} \|\,\mathrm{BE}_\Phi\,\|_2 &\le \|\Delta_r^\pi\|_2 + \gamma \|\Delta_P^\pi\|_2 \|\boldsymbol{w}_\Phi^\pi\|_2 \le \\ &\le \|\Delta_r^\pi\|_2 + \gamma \|\Delta_P^\pi\|_F \|\boldsymbol{w}_\Phi^\pi\|_2 \ , \end{aligned} \tag{2.12}$$

where the second inequality follows from $\|X\|_2 \le \|X\|_F$.

The Bellman error (BE) decomposition in (2.12) has two main limitations. The first limitation is that it is expressed in terms of the L_2 norm rather than L_∞ norm, which is needed for standard Bellman residual bounds (Williams & Baird, 1993). This can be addressed, in part, by using the weighted L_2 norm bounds (Munos, 2007). The second limitation of (2.12) is that it depends on $\|\boldsymbol{w}_\Phi^\pi\|_2$ besides the terms $\|\Delta_r^\pi\|_2, \|\Delta_P^\pi\|_2$ that we focus on. Since $\|\boldsymbol{w}_\Phi^\pi\|_2$ can be problem-dependent, the theoretical analysis of its impact on the approximation error is beyond the scope of this work.

2.3.2 Linear Models Approximation

In the linear model approximation, we aim to approximate the transition model using linearly independent features Φ. Such features are required to approximate next state feature values $\Phi(s'|s)$ and reward function $\boldsymbol{r}$ given the current state. The idea is similar to linear value function approximation approach. Parr et al. (2008) shows that the same set of features that predict a reliable value function can be implemented for approximating the linear model.

Given a basis function $\Phi = \{\phi_1, \dots, \phi_k\}$ for a linear value function or a linear model, a feature vector $\Phi(s) = [\phi_1(s), \dots, \phi_k(s)]^\top$ can be defined with $\phi_i(s)$ representing value of feature i in state s. Meanwhile, $\Phi(s'|s)$ is defined as the random vector of next features (Parr et al., 2008):

$$\Phi(s'|s) = [\phi_1(s'), \dots, \phi_k(s')]^\top \ , \tag{2.13}$$

where $s' \sim P(s'|s)$. The idea is to generate a matrix $P_\Phi \in \mathbb{R}^{k \times k}$ that predicts the expected next feature vector:

$$P_\Phi^\top \Phi(s) \approx \mathbb{E}_{s' \sim P(s'|s)}\{\Phi(s'|s)\} \tag{2.14}$$

And minimizes the expected feature prediction error:

$$P_\Phi = \arg\min_{P_k} \sum_s \left\| P_k^\top \Phi(s) - \mathbb{E}\{\Phi(s'|s)\} \right\|_2^2 \tag{2.15}$$

One way to solve this optimization problem is to find the least-squares solution to the system $\Phi P_\Phi \approx P\Phi$. Given the policy π, the solution to the system would be:

$$P_\Phi^\pi = (\Phi^\top \Phi)^{-1} \Phi^\top P^\pi \Phi = \Phi^+ P^\pi \Phi, \tag{2.16}$$

We can also implement a standard least-squares projection into span(Φ) to compute an approximate reward predictor:

$$\boldsymbol{r}_{\Phi}^{\pi} = (\Phi^{\top}\Phi)^{-1}\Phi^{\top}\boldsymbol{r}^{\pi} = \Phi^{+}\boldsymbol{r}^{\pi} \ . \tag{2.17}$$

with corresponding approximate reward $\hat{\boldsymbol{r}} = \Phi\boldsymbol{r}_{\Phi}^{\pi}$.

2.4 Policy Iteration and Approximate Policy Iteration

Policy iteration (Howard, 1960) also known as actor-critic architectures (Barto, Sutton, & Anderson, 1983; Sutton & Barto, 1984) is a process of identifying the optimal policy for any given MDP. Policy iteration is an iterative method in the space of deterministic policies; it finds the optimal policy by performing a series of monotonically improving policies. Every iteration consists of two stages:

- Stage one: policy evaluation (also recognized as the critic) calculates the state-action value function of given policy. Policy evaluation solves the linear system of the Bellman equations.
- Stage two: policy improvement (which is also recognized as the actor) describes the improved greedy policy. The resulting policy is a deterministic policy which is at least as good as the previous policy.

These two stages are repeated until the calculated policy is the same as the previous one. In this phase, we claim that the iteration has converged to the optimal policy. We can guarantee the convergence of policy iteration to the optimal policy only in the event of a tabular representation of the value function (exact solution), and tabular representation of policy (Lagoudakis & Parr, 2003). Such precise representations and approaches are unrealistic for large state and action spaces. Therefore, approximation methods are applied to mitigate the curse of dimensionality. Approximations in the policy-iteration structure can be presented at two points:

- In the value function representation: A general parametric function approximation replaces the tabular representation of the original value function. $\hat{q}(s, a; \boldsymbol{w})$

- In the policy representation: The tabular representation of the policy is substituted by a parametric representation $\hat{\pi}(s;\theta)$

In each case, merely the parameters of the representation must be saved, and we require a much smaller memory to store the parameters than the tabular case. The critical factor for a good approximate algorithm is the determination of the parametric approximation structure and the decision of the projection method (Lagoudakis & Parr, 2003).

2.4.1 Least-Squares Temporal Difference (LSTD)

LSTD is a temporal difference algorithm based on the theory of linear least-squares function approximation, and it is introduced by Bradtke and Barto (1996). The idea is that a linear architecture approximates the state value function and its representation Φ is consist of a compact description of the essential functions and a set of parameters $\boldsymbol{w}_\Phi$:

$$\boldsymbol{w}_\Phi = (\mathbf{I} - \gamma(\Phi^\top\Phi)^{-1}\Phi^\top P\Phi)^{-1}(\Phi^\top\Phi)^{-1}\Phi^\top \boldsymbol{r} \tag{2.18}$$

$$= (\Phi^\top\Phi - \gamma\Phi^\top P\Phi)^{-1}\Phi^\top \boldsymbol{r}\ . \tag{2.19}$$

LSTD is sample-efficient and converges quicker than TD(λ) (Sutton, 1988). LSTD is useful for prediction problems when the agent is learning the value function of a fixed policy. It does not have any use in control problems because LSTD learns the state-value function of a fixed policy and can not be applied in action selection without learning a model of the underlying MDP.

2.4.2 Least-Squares Policy Iteration

It is not practical to use LSTD directly as policy evaluation for a policy iteration algorithm. Koller and Parr (2000) present an instance of LSTD-style function approximation and policy iteration where the policy changes between two poor performances in an MDP with a small number of states. Least-Squares Policy Iteration (LSPI) suggested by Lagoudakis and Parr (2003) is a modified algorithm for policy iteration which takes advantage of the benefits of LSTD and extends them to control problems. LSPI mixes the policy-search effectiveness of policy iteration with the data efficiency of LSTDQ.

For instance, in the LSPI algorithm, the linear approximation for state-action value function (q-function) is adopted from LSTD and extended to LSTDQ, which performs the weighting of the features. However, LSTDQ approximates q-functions applying a different feature set for each possible action.

LSPI learns the state-action value function which allows for action selection without a model. It is sample-efficient, off-policy algorithm that reuses samples in each iteration. Besides, it does not require any parameter tuning since learning parameters, such as the learning rate, are omitted.

In the following chapter, we review the most relevant techniques for feature selection which have been successfully applied for value function approximation.

CHAPTER 3

Feature Selection Methods

In the previous chapter, we discussed Least-Squares Policy Iteration (LSPI) for linear Value Function Approximation (VFA) which employs previously chosen sets of features. LSPI tends to evaluate the approximated function, analyze the features, tweaking them, and re-running the approximator until the approximation is consistent. This approach by itself is not satisfying, as this iterative process of trying different feature sets until success depends on human intuition, dismantling the concept of autonomous artificial intelligent agents. Feature selection methods attempt to automate this process in a way that maintain the low computational cost of linear approximation (Petrik et al., 2010).

A significant limitation of linear value function approximation is that it requires good features, which is, features that can represent the state space properly and be able approximate the optimal value function well. This is often a difficult goal to achieve since good priori estimates of the optimal value functions are rarely available. Considerable effort has therefore been dedicated to methods that can automatically construct useful features for linear value function approximation. Examples of methods that construct features, or select good features from a large set, include proto-value functions (Mahadevan, 2005), diffusion wavelets (Mahadevan & Maggioni, 2006), Fourier Basis (Konidaris, Osentoski, & Thomas, 2011), Krylov bases (Petrik, 2007), BEBF (Parr et al., 2007), L_1-regularized TD (Kolter & Ng, 2009), and approximate linear programs (Petrik et al., 2010). In this chapter, we briefly describe some of the most common methods in feature selection techniques for reinforcement learning problems.

3.1 Proto-value Functions

Mahadevan (2005) introduced Proto-Value Functions (PVFs), which is a specific instantiation of a general scheme for basis functions by diagonalizing symmetric diffusion operators. PVFs are created by employing the eigenvectors of the graph Laplacian on an undirected graph (N, E) is made of state transitions to develop a basis for linear value function approximation.

The states of the MDP express nodes of an undirected weighted graph given a fixed policy. This graph can be defined by a symmetric adjacency matrix W, where each element of the matrix denotes the weight between the nodes. A diagonal matrix D of the node degrees is determined as $D_{xx} = \sum_{y \in N} W_{xy}$. The benefit of the normalized Laplacian is that it is symmetric, and therefore its eigenvectors are orthogonal (Petrik, 2007). Proto-value functions presenting a compressed representation of the powers of transition matrices and they span the entire space of possible value functions for a given state space.

3.2 Diffusion Wavelets

Mahadevan and Maggioni (2006) introduce two methods for automatically constructing basis functions on state spaces that can be represented as graphs or manifolds. The first way is to use the eigenfunctions of the Laplacian similar to Proto-value functions; another approach is based on diffusion wavelets, which generalize classical wavelets to graphs.

The extension of the Laplacian approach with Diffusion Wavelet Transforms (DWT) is a compact multi-level representation of Markov diffusion processes on manifolds and graphs. Diffusion wavelets are used to perform a fast multi scale analysis of functions on a manifold or graph, which are generalizing wavelet analysis and associated with signal processing techniques such as compression.

The diffusion wavelets create a low-order approximation of the inverse of a matrix. One of the disadvantages of applying the wavelets for VFA is the massive computational cost required to build the inverse approximation. However, once the inverse is constructed, it can be reused for different rewards as long as the transition matrix is fixed (Petrik, 2007).

3.3 Eigenbasis of the Transition Matrix

Eigenvalues of a square matrix $A \in \mathbb{R}^{n \times n}$ play an important role in dynamic problems such as $d\boldsymbol{u}/dt = A\boldsymbol{u}$. Almost all vectors $\boldsymbol{x} \in \mathbb{R}^n$ will have a new direction after multiplication by A. There are some exceptions which are known as eigenvectors. Eigenvectors $\boldsymbol{x}$ stay in the same direction after such multiplication by A. The result of $A\boldsymbol{x}$ is a vector λ times the original $\boldsymbol{x}$ so we can write $A\boldsymbol{x} = \lambda\boldsymbol{x}$. The number λ is an eigenvalue of A. The eigenvalue λ can tell whether the eigenvector is expanded or shrunk or reversed or remains unchanged (Strang, 2009). To compute eigenvalues we need to set the determinant of $A - \lambda\mathbf{I}$ to zero. Solving $det(A - \lambda\mathbf{I}) = 0$ result in eigenvalues, however to find the eigenvectors we need to find all vectors in the Null-space of $A - \lambda I$. For each eigenvector λ, any vector $\boldsymbol{x}$ that is a solution for $(A - \lambda\mathbf{I})\boldsymbol{x} = 0$ is an eigenvector for A.

The matrix A can be transformed to a diagonal matrix Λ when we use the eigenvectors properly. Suppose A is a n by n matrix with n linear eigenvectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. We can put all eigenvector into the columns of an eigenvector matrix S.

A matrix A can be diagonalized as $S^{-1}AS = \Lambda$ where Λ is a diagonal matrix filled with eigenvalues of A. This form of matrix factorization is called *diagonalization*, and can be shown in different forms as well:

$$AS = S\Lambda \qquad \text{or} \qquad A = S\Lambda S^{-1} \tag{3.1}$$

The matrix S is invertible because we assumed the eigenvectors of A to be linearly independent. Without independent eigenvector, diagonalization is impossible. Also, some matrices do not have enough eigenvectors, so they cannot be diagonalized either. In general, any matrix that does not contain repeated eigenvalue can be diagonalized. It worth mentioning that invertibility and diagonalizability are two different issues. A matrix is invertible when no eigenvalue is equal to zero ($\lambda \neq 0$). However, diagonalizability is only concerned with the available eigenvectors. Enough independent eigenvector from different λ satisfies diagonalizability.

Difference-equation is an equation that recursively represents a sequence. Once the primary terms are given, any further term of the sequence is determined as a function of the previous terms. A typical difference-equation is $\boldsymbol{u}_{k+1} = A\boldsymbol{u}_k$. At each step, the current state $\boldsymbol{u}_k$ multiplies by A. The solution for such difference equation is $\boldsymbol{u}_k = A^k\boldsymbol{u}_0$. Diagonalization of matrix A provides a quick way to calculate A^k. Assume S is the matrix of eigenvectors of A. According to equation (3.1) matrix A can be written as $A = S\Lambda S^{-1}$. This form of factorization is suitable for computing the powers of A. The reason is that every time S^{-1} multiplies S we get $\mathbf{I}$:

$$A^k\boldsymbol{u}_0 = (S\Lambda S^{-1})\cdots(S\Lambda S^{-1})\boldsymbol{u}_0 = S\Lambda^k S^{-1}\boldsymbol{u}_0 \tag{3.2}$$

We can split $S\Lambda^k S^{-1}\boldsymbol{u}_0$ as follows. First, we can write $\boldsymbol{u}_0$ as a combination of eigenvectors $c_1\boldsymbol{x}_1 + \cdots + c_n\boldsymbol{x}_n$. Then $\boldsymbol{c} = S^{-1}\boldsymbol{u}_0$. Then we need to multiply each eigenvector $\boldsymbol{x}_i$ by λ_i^k so we have $\Lambda^k S^{-1}\boldsymbol{u}_0$. And finally we add up the $c_i\lambda_i^k\boldsymbol{x}_i$ to get the solution $\boldsymbol{u}_k = A^k\boldsymbol{u}_0$. The solution for $\boldsymbol{u}_{k+1} = A\boldsymbol{u}_k$ would be:

$$\boldsymbol{u}_k = A^k\boldsymbol{u}_0 = c_1\lambda_1^k\boldsymbol{x}_1 + \cdots + c_n\lambda_n^k\boldsymbol{x}_n \tag{3.3}$$

If A is a Markov matrix, all entries of A are positive, and all rows sum up to 1. These properties guarantee that the largest eigenvalue of A is $\lambda = 1$ (Strang, 2009). Considering that P is the transition matrix (Markov matrix) for a fixed Markov policy, we can represent the value function as (Puterman, 2014):

$$\boldsymbol{v} = (\mathbf{I} - \gamma P)^{-1}\boldsymbol{r} = \sum_{i=0}^{\infty}(\gamma P)^i\boldsymbol{r}. \tag{3.4}$$

Let assume the transition matrix is diagonalizable and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are n linearly independent orthonormal eigenvectors of P with corresponding $\lambda_1 \ldots \lambda_n$ eigenvalues. We can show the reward vector as a mixture of eigenvectors similar to $\boldsymbol{u}_0$ in the deference equation problem.

$$\boldsymbol{r} = c_1\boldsymbol{x}_1 + \cdots + c_n\boldsymbol{x}_n = \sum_{j=1}^{n} c_j\boldsymbol{x}_j \tag{3.5}$$

For $\boldsymbol{c} = S^{-1}\boldsymbol{r}$, and using $P^i\boldsymbol{x}_j = \lambda_j^i\boldsymbol{x}_j$ we have (Petrik, 2007):

$$\boldsymbol{v} = \sum_{j=1}^{n}\sum_{i=1}^{\infty}(\gamma\lambda_j)^i c_j\boldsymbol{x}_j = \sum_{j=1}^{n}\frac{1}{1-\gamma\lambda_j}c_j\boldsymbol{x}_j = \sum_{j=1}^{n} dj\boldsymbol{x}_j \tag{3.6}$$

Considering a subset U of eigenvectors $\boldsymbol{x}_j$ as a basis for linear value function $\boldsymbol{v}$ the error bound on the approximation would be:

$$\|\boldsymbol{v} - \hat{\boldsymbol{v}}\|_2 \leq \sum_{j\notin U}|d_j|. \tag{3.7}$$

The value function can be approximated by using that $\boldsymbol{x}_j$'s with greatest $|d_j|$.

3.4 Krylov Subspace method

Alexei Krylov, a Russian applied mathematician introduce the concept of Krylov subspace in 1931. Krylov subspace $\mathcal{K}_n(A, \boldsymbol{b})$ is the subspace spanned by $b, Ab, \ldots, A^{n-1}b$. Petrik (2007) proposed a value function approximation based on Krylov subspaces, and denoted these vectors as $y_i = P^i\boldsymbol{r}$ for $i \in [0, n-1]$. The Krylov basis constructs a basis that minimizes the Bellman error $\|\boldsymbol{v} - \hat{\boldsymbol{v}}\|$ in each iteration. The major issue with Krylov vectors is that they are far from orthogonal. The Arnoldi iteration creates an orthonormal basis $q_1, q_2, \ldots$ for the same space using the Gram-Schmidt technique. Arnoldi iteration orthogonalize each new vector against the previous one (see algorithm 1). The Krylov subspace spanned by $\boldsymbol{r}, P\boldsymbol{r}, \ldots, P^{n-1}\boldsymbol{r}$ then will have much better basis with $q_1, \ldots, q_n$ (Strang, 2009).

Algorithm 1: Arnoldi Iteration

```
Input: Transition matrix $P$, rewards $\boldsymbol{r}$, and number of features $k$
1  $\boldsymbol{q}_1 = \boldsymbol{r}/\|\boldsymbol{r}\|$ ;
2  for $n = 1$ to $k - 1$ do
3      $\boldsymbol{v} = Pq_n$ ;
4      for $j = 1$ to $n$ do
5          $h_{jn} = \boldsymbol{q}_j^\top v$ ;
6          $\boldsymbol{v} = \boldsymbol{v} - h_{jn}\boldsymbol{q}_j$ ;
7      end
8      $h_{n+1,n} = \|\boldsymbol{v}\|$ ;
9      $\boldsymbol{q}_{n+1} = \boldsymbol{v}/h_{n+1,n}$ ;
10 end
11 return Approximation features: $\Phi = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k\}$.
```

3.5 Bellman Error Basis Function

Bellman Error Basis Function (BEBF) is a well-known method for expanding already given basis functions for linear value function (Parr et al., 2007). The idea behind BEBF is taken from Bellman-error-based methods (Keller et al., 2006) in which the new feature ϕ_{k+1} is obtained from bellman error of approximated value function with already calculated set of features $\Phi \in \mathbb{R}^{|\mathcal{S}|\times k}$.

Let $\hat{\boldsymbol{v}} = \Phi\boldsymbol{w}$ be the approximated value function. We can obtain $\phi_{k+1} = T\hat{\boldsymbol{v}} - \hat{\boldsymbol{v}}$ where T is the Bellman operator. ϕ_{k+1} is orthogonal to the span of Φ and is a Bellman Error Basis Function (BEBF) for constructing a new design matrix $\Phi' = [\Phi, \phi_{k+1}]$ when we concatenate the column vector to the previews set of features. The Bellman error is an intuitively interesting approach to expand the basis because it is pointing towards the optimal value function $\boldsymbol{v}^*$ (Parr et al., 2008).

3.6 L_1-regularized TD

The Least-Squares Temporal Difference (LSTD) algorithms (see section 2.4.1) are employed for learning the parameters of the linear value function. The naive implementation of LSTD only uses the trajectories generated by the simulator. However, when the number of features is enormous, LSTD can overfit to data and become computationally expensive. Kolter and Ng (2009) propose an L_1-regularization method that provides sparse solutions and behaves as a feature selection process.

Given n samples of (s, a, r, s') from system trajectories we can form $n \times k$ feature matrices Φ and Φ' and the reward vector $\boldsymbol{r}$. LSTD computes the fixed point of the following equation:

$$\boldsymbol{w} = \hat{f}(\boldsymbol{w}) = \arg\min_{u \in \mathbb{R}^k} \left\| \Phi u - (\boldsymbol{r} + \gamma \Phi' \boldsymbol{w}) \right\|_D^2 \tag{3.8}$$

The solution follows analytically by solving the linear system:

$$\boldsymbol{w} = (\Phi^\top (\Phi - \gamma \Phi'))^{-1} \Phi^\top \boldsymbol{r} = A^{-1} b \tag{3.9}$$

To limit overfitting, particularly when the number of samples are smaller than the number of features ($m < k$), Kolter and Ng (2009) add a regularization penalty function to the equation (3.8).

$$\boldsymbol{w} = \hat{f}(\boldsymbol{w}) = \arg\min_{u \in \mathbb{R}^k} \frac{1}{2} \left\| \Phi u - (\boldsymbol{r} + \gamma \Phi' \boldsymbol{w}) \right\|_D^2 + \beta \|u\|_1 \tag{3.10}$$

Because the equation (3.10) is not convex and non-differentiable, the optimal solution $\boldsymbol{w}$ can be constructed incrementally. it is desirable to apply an algorithm known as Least Angle Regression (LARS) (Efron, Hastie, Johnstone, Tibshirani, et al., 2004) and updating one element of $\boldsymbol{w}$ at a time until we approach the exact solution of the optimization problem. Because L_1 regularization is known to provide sparse solutions, it can provide for more efficient implementation and be beneficial in finding more meaningful of feature selection.

3.7 Approximate Linear Programming with L_1 Regularization

For large MDPs, one way to approximate value function is employing Approximate Linear Programming (ALP) (Schweitzer & Seidmann, 1985; de Farias & Van Roy, 2005). Given a set of features Φ and a linear space $\mathcal{M} \subseteq \mathbb{R}^{|\mathcal{S}|}$, where $\mathcal{M} = \text{span}\{\Phi\}$ and its first column is $\mathbf{1}$ we can formulate the problem as the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{v} \in \mathcal{M}} \quad & \sum_{s \in \mathcal{S}} \rho(s) v(s) \\ \text{subject to} \quad & r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') v(s') \leq v(s) \end{aligned} \tag{3.11}$$

where ρ is the distribution over the initial states. For the sake of simplicity, one can reformulate the optimization problem in (3.11) by using an ALP constraint matrix $\mathcal{L}$. The constraint matrix has the property of $\mathcal{L}\boldsymbol{v} \leq \boldsymbol{v}$ that is equivalent to the set $\{L_a \boldsymbol{v} \leq \boldsymbol{v} : \forall a \in \mathcal{A}\}$. L_a denotes the Bellman update.

$$\begin{aligned} \min_{\boldsymbol{v}} \quad & \boldsymbol{\rho}^\top \boldsymbol{v} \\ \text{subject to} \quad & \mathcal{L}\boldsymbol{v} \leq \boldsymbol{v} \\ & \boldsymbol{v} \in \mathcal{M} \end{aligned} \tag{3.12}$$

As we mentioned in the previous section, the approximate dynamic programming approach requires a small set of appropriate features to find a reliable solution. However, when the number of features is enormous, the solution may cause to over-fit when the number of samples is limited. ALP approach is not an exception. Petrik et al. (2010) suggest an L_1 regularization technique for approximate linear programming (RALP) to overcome such problems. L_1 regularization can pick the appropriate feature automatically and also improve the algorithm performance by limiting the number of selected features. The RALP for basis Φ and L_1 constraint ψ is defined as follow:

$$\begin{aligned} \min_{\boldsymbol{v}} \quad & \boldsymbol{\rho}^\top \Phi \boldsymbol{w} \\ \text{subject to} \quad & \boldsymbol{\mathcal{L}} \Phi \boldsymbol{w} \leq \Phi \boldsymbol{w} \\ & \|\boldsymbol{w}\|_{1,e} \leq \psi \ , \end{aligned} \tag{3.13}$$

where $\|\boldsymbol{w}\|_{1,e}$ is the weighted L_1 norm for the value function parameters. RALP with sampled constraints guarantees that the solution is bounded and also presents worst-case error bounds on the value function (Petrik et al., 2010).

In the next chapter, we introduce a new feature construction algorithm that can extract a basis for a linear value function from the raw input data.

CHAPTER 4

FFS: Fast Feature Selection

In this chapter, we describe the proposed method for selecting features from a low-rank approximation of the transition probabilities. To simplify the exposition, we first introduce the method for the tabular case and then extend it to the batch RL setting with many raw features in section 4.2.

4.1 FFS for Tabular Solution

Algorithm 2: TFFS: Tabular Fast low-rank Feature Selection

```
  Input: Transition matrix $P$, rewards $\boldsymbol{r}$, and number of features $k$
1 Compute SVD decomposition of $P$: $P = U\Sigma V^\top$ ;
2 Assuming decreasing singular values in $\Sigma$, select the first $k-1$ columns of $U$:
    $U_1 \leftarrow [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{k-1}]$ ;
3 return Approximation features: $\Phi = [U_1, \boldsymbol{r}]$.
```

The Tabular Fast Feature Selection algorithm is summarized in Algorithm 2. Informally, the algorithm selects the top $k-1$ singular vectors and the reward function for the features. Our error bounds show that including the reward function as one of the features is critical. It is not surprising that when the matrix P is of a rank at most k then using the first k singular vectors will result in no approximation error. However, such low-rank matrices are rare in practice. We now show that it is sufficient that the transition matrix P is close to a low-rank matrix for TFFS to achieve small approximation errors. In order to bound the error, let the SVD decomposition of

P be $\mathrm{SVD}(P) = U\Sigma V^\top$, where

$$U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}, \qquad \Sigma = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix}, \qquad V = \begin{bmatrix} V_1 & V_2 \end{bmatrix} .$$

That implies that the transition probability matrix can be expressed as:

$$P = U_1\Sigma_1 V_1^\top + U_2\Sigma_2 V_2^\top .$$

Let matrix U_1 have k columns and let the singular values be ordered decreasingly. Then, algorithm 2 generates $\Phi = U_1$. The following theorem bounds the error regarding the largest singular value for a vector not included in the features.

Theorem 2. *Assuming k features Φ computed by Algorithm 2, the error terms in 1 are upper bounded as:*

$$\begin{aligned} \|\Delta_P\|_2 &\le \|\Sigma_2\|_2, \\ \|\Delta_r\|_2 &= 0 . \end{aligned}$$

Proof. From the definition of Δ_P and P_Φ we get the following equality:

$$\Delta_P = U\Sigma V^\top U_1 - U_1(U_1^\top U_1)^{-1}U_1^\top U\Sigma V^\top U_1 .$$

Recall that singular vectors are orthonormal which implies that $(U_1^\top U_1)^{-1} = \mathbf{I}$ and $U_1^\top U = \begin{bmatrix} \mathbf{I}_1 & 0 \end{bmatrix}$. Substituting these terms into the equality above, we get:

$$\begin{aligned} \|\Delta_P\|_2 &= \left\|(U\Sigma V^\top - U_1\Sigma_1 V_1^\top)U_1\right\|_2 \\ &\le \left\|U\Sigma V^\top - U_1\Sigma_1 V_1^\top\right\|_2 \|U_1\|_2 . \end{aligned}$$

Simple algebraic manipulation shows that $\left\|U\Sigma V^\top - U_1\Sigma_1 V_1^\top\right\|_2 = \|\Sigma_2\|_2$ and $\|U_1\|_2 = 1$ because U is an unitary matrix. This establishes the inequality for Δ_P; the result for Δ_r follows directly from the properties of orthogonal projection since $\boldsymbol{r}$ itself is included in the features. $\square$

Theorem 2 implies that if we choose Φ in a way that the singular values in Σ_2 are zero (when the transition matrix is low rank), Δ_P would be zero. That means that for a matrix of rank k there is no approximation error because $\|\Delta_P\|_2 = 0$. More broadly, when the rank of the matrix is greater than k, the error is minimized by choosing the singular vectors with the greatest singular values. That means that TFFS chooses features Φ that minimize the error bound in Theorem 2.

4.1.1 Contributions

In this section, we present TFFS, an exact solution algorithm for fast low-rank feature selection. Theorem 2 provides an upper error bound on Bellman error for the TFFS's approximation features.

4.2 FFS for Approximate Solution

Using TFFS in batch RL is impractical since the transition matrix and reward vector are usually too large and are not available directly. The values must instead be estimated from samples and the raw features.

Algorithm 3: FFS: Fast low-rank Feature Selection from raw features

```
Input: Sampled raw features $A$, next state of raw feature $A'$ , rewards $\boldsymbol{r}$, and number of
       features $k$
1 Estimate compressed transition probabilities $P_A = A^+ A'$ as in LSTD ;
2 Compute SVD decomposition of $P_A$: $P_A = U \Sigma V^\top$ ;
3 Compute compressed reward vector: $\boldsymbol{r}_A = A^+ \boldsymbol{r}$ ;
4 Assuming decreasing singular values in $\Sigma$, select the first $k-1$ columns of $U$:
    $U_1 \leftarrow [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{k-1}]$ ;
5 return Approximation features: $\widehat{\Phi} = [U_1, \boldsymbol{r}_A]$.
```

As described in the introduction, we assume that the domain samples include a potentially large number of low-information raw features. We use A to denote the $n \times l$ *matrix of raw features.* As with Φ, each row corresponds to one state, and each column corresponds to one *raw* feature. The compressed transition matrix is denoted as $P_A = A^+PA$ and compressed rewards are denoted as $\boldsymbol{r}_A = A^+\boldsymbol{r}$ and are computed as in equations (2.16) and (2.17). To emphasize that the matrix P is not available, we use $A' = PA$ to denote the expected value of features after one step. Using this notation, the compressed transition probabilities can be expressed as $P_A = A^+A'$.

Algorithm 3 describes the FFS method that uses raw features. Similarly to TFFS, the algorithm computes an SVD of the transition matrix. Note that the features $\widehat{\Phi}$ are linear combinations of the raw features. To get the actual state features, it is sufficient to compute $A\widehat{\Phi}$ where $\widehat{\Phi}$ is the output of Algorithm 3. The matrix $\widehat{\Phi}$ represents features for P_A and is of a dimension $l \times k$ where l is the number of raw features in A. We omit the details for how the values are estimated from samples, as this is well known, and refer the interested reader to Lagoudakis and Parr (2003) and Johns, Petrik, and Mahadevan (2009).

Using raw features to compress transition probabilities and rewards is simple and practical, but it is also essential to understand the consequences of relying on these raw features. Because FFS computes features that are a linear combination of the raw features, they cannot express more complex value functions. FFS thus introduces additional error—akin to bias—but reduces sampling error—akin to variance. The following theorem shows that the errors due to our approximation and using raw features merely add up with no additional interactions.

Theorem 3. *Assume that the raw features A for P and computed features $\widehat{\Phi}$ for P_A are normalized, such that $\|A\|_2 = \left\|\widehat{\Phi}\right\|_2 = 1$. Then:*

$$\|\Delta_P^{A\widehat{\Phi}}\|_2 \le \|\Delta_P^A\|_2 + \|\Delta_{P_A}^{\widehat{\Phi}}\|_2 \ ,$$

$$\|\Delta_r^{A\widehat{\Phi}}\|_2 \le \|\Delta_r^A\|_2 + \|\Delta_{r_A}^{\widehat{\Phi}}\|_2 \ ,$$

where the superscript of Δ indicates the feature matrix for which the error is computed; for example $\Delta_{P_A}^{\widehat{\Phi}} = P_A\widehat{\Phi} - \widehat{\Phi}(P_A)_{\widehat{\Phi}}$.

Proof. We show the result only for Δ_P; the result for Δ_r follows similarly. From the definition of

Δ,

$$\left\|\Delta_P^{A\widehat{\Phi}}\right\|_2 = \left\|PA\widehat{\Phi} - A\widehat{\Phi}P_{A\widehat{\Phi}}\right\|_2 .$$

Now, by adding a zero $(AP_A\widehat{\Phi} - AP_A\widehat{\Phi})$ and applying the triangle inequality, we get:

$$\begin{aligned}\left\|\Delta_P^{A\widehat{\Phi}}\right\|_2 &= \left\|PA\widehat{\Phi} - AP_A\widehat{\Phi} + AP_A\widehat{\Phi} - A\widehat{\Phi}P_{A\widehat{\Phi}}\right\|_2 \leq \\ &\leq \left\|PA\widehat{\Phi} - AP_A\widehat{\Phi}\right\|_2 + \left\|AP_A\widehat{\Phi} - A\widehat{\Phi}P_{A\widehat{\Phi}}\right\|_2 .\end{aligned}$$

Given $(A\widehat{\Phi})^+ = \widehat{\Phi}^+A^+$ and the property of the compressed transition matrix in equation (2.16) we can show:

$$\begin{aligned}(P_A)_{\widehat{\Phi}} - P_{A\widehat{\Phi}} &= \widehat{\Phi}^+P_A\widehat{\Phi} - (A\widehat{\Phi})^+PA\widehat{\Phi} \\ &= \widehat{\Phi}^+(P_A - A^+PA)\widehat{\Phi} = 0\end{aligned}$$

$$\begin{aligned}\left\|\Delta_P^{A\widehat{\Phi}}\right\|_2 \leq \|PA - AP_A\|_2\left\|\widehat{\Phi}\right\|_2 &+ \\ + \left\|P_A\widehat{\Phi} - \widehat{\Phi}(P_A)_{\widehat{\Phi}}\right\|_2\|A\|_2 &.\end{aligned}$$

The theorem then follows directly from algebraic manipulation and the fact that the features are normalized. □

Note that the normalization of features required in Theorem 3 can be achieved by multiplying all features by an appropriate constant, which is an operation that does not affect the approximate value function. Scaling features does, however, affect the magnitude of $\boldsymbol{w}_\Phi$, which, as we discuss above, is problem-specific and largely independent of the feature selection method used.

Perhaps one of the most attractive attributes of FFS is its simplicity and low computational complexity. Selecting the essential features only requires computing the singular value decomposition—for which many efficient methods exist—and augmenting the result with the reward function. As we show next, this simple approach is well-motivated by bounds on approximation errors.

We described FFS in terms of singular value decomposition and showed that when the (compressed) transition probability matrix has a low rank, the approximation error is likely to be small. Next, we describe the relationship between FFS and other feature selection methods in more detail.

4.2.1 Contributions

In this section, we introduce a fast low-rank feature selection algorithm that uses raw feature inputs to generate approximation features for linear value function. We also show in Theorem 3 that using raw features and feature approximation simultaneously does not result in additional interaction error.

4.3 Related Feature Selection Methods

In this section, we describe similarities and differences between FFS and related feature construction or selection methods.

Perhaps the best-known method for feature construction is the technique of *proto-value functions* (Mahadevan & Maggioni, 2006, 2007). Proto-value functions are closely related to spectral approximations (Petrik, 2007). This approximation uses the eigenvector decomposition of the transition matrix $P = S\Lambda S^{-1}$, where S is a matrix with eigenvectors as its columns and Λ is a diagonal matrix with eigenvalues that are sorted from the largest to the smallest. The first k columns of S are then used as the approximation features. As with our FFS method, it is beneficial to augment these features with the reward vector. We will refer to this method as EIG+R in the numerical results. Surprisingly, unlike with FFS, using the top k eigenvectors does not guarantee zero Bellman residual even if the rank of P is less than k.

Using the Krylov subspace is another feature selection approach (Petrik, 2007) which has also been referred to as BEBF (Parr et al., 2007, 2008). The Krylov subspace $\mathcal{K}$ is spanned by the images of $\boldsymbol{r}$ under the first k powers of P (starting from $P^0 = \mathbf{I}$):

$$\mathcal{K}_k(P, \boldsymbol{r}) = \operatorname{span}\{\boldsymbol{r}, P\boldsymbol{r}, \ldots, P^{k-1}\boldsymbol{r}\} \ .$$

Petrik (2007) shows that when k is equal to the degree of the minimal polynomial, the approximation error is zero. Krylov methods are more likely to work in different problem settings than either EIG+R or FFS and can be easily combined with them.

Algorithm 4: LFD: Linear Feature Discovery for a fixed policy π (Song et al. 2016).

```
1 $D_0 \leftarrow \text{random}(k, l)$;
2 $i \leftarrow 1$;
3 **while** *Not Converged* **do**
4     $E_i \leftarrow A^+ A' D_{i-1}$ ;
5     $D_i \leftarrow (A E_i)^+ A'$;
6     $i \leftarrow i + 1$ ;
7 **end**
8 **return** $E_k$ // Same role as $\widehat{\Phi}$ in FFS.
```

Finally, Linear Feature Discovery (LFD) (Song et al., 2016) is a recent feature selection method that is closely related to FFS. Algorithm 4 depicts a simplified version of the LFD algorithm, which does not consider the reward vector and approximates the value function instead of the q-function for a fixed policy π. Recall that A is the matrix of raw features and $A' = P^\pi$.

LFD is motivated by the theory of *predictive optimal feature encoding.* A low-rank encoder E^π is *predictively optimal* if there exist decoders D_s^π and D_r^π such that:

$$A E^\pi D_s^\pi = P^\pi A \ , \qquad\qquad A E^\pi D_r^\pi = r^\pi \ .$$

When an encoder and decoder are predictively optimal, then the Bellman error is zero (Song et al., 2016). Unfortunately, it is almost impossible to find problems in practice in which a predictively optimal controller exists. No bounds on the Bellman error are known when a controller is merely close to predictively optimal. This is in contrast with the bounds in Theorems 2 and 3 that hold for FFS.

Although LFD appears to be quite different from FFS, our numerical experiments show that it computes solutions that are similar to the solutions of FFS. We argue that LFD can be interpreted as a coordinate descent method for computing the following low-rank approximation problem:

$$\min_{E\in\mathbb{R}^{l\times k}, D\in\mathbb{R}^{k\times l}} \|AED - A'\|_F^2 \ . \tag{4.1}$$

This is because the iterative updates of E_i and D_i in Algorithm 4 are identical to solving the following optimization problems:

$$E_i \leftarrow \arg\min_{E\in\mathbb{R}^{l\times k}} \|AED_{i-1} - A'\|_F^2$$

$$D_i \leftarrow \arg\min_{D\in\mathbb{R}^{k\times l}} \|AE_iD - A'\|_F^2$$

The equivalence follows directly from the orthogonal projection representation of linear regression. This kind of coordinate descent is a very common heuristic for computing low-rank matrix completions (Hastie, Mazumder, Lee, & Zadeh, 2015). Unfortunately, the optimization problem in equation (4.1) is *non-convex* and coordinate descent, like LFD, may only converge to a local optimum, if at all. Simple algebraic manipulation reveals that any set of k singular vectors represents a local minimum of LFD. Finally, we are not aware of any method that can solve equation (4.1) optimally.

Similarly to LFD, FFS solves the following optimization problem:

$$\min_{E\in\mathbb{R}^{l\times k}, D\in\mathbb{R}^{k\times l}} \|ED - A^+A'\|_F^2 \ . \tag{4.2}$$

This fact follows readily from the SVD decomposition of A^+A' and the fact that the Frobenius norm is equal to the L_2 norm of the the singular values (Hastie, Tibshirani, & Friedman, 2009; Golub & Van Loan, 2013).

Note that when the using tabular features ($A = \mathbf{I}$) the optimization problems in equation (4.1) and equation (4.2) are identical. For any other raw features, there are two reasons for preferring equation (4.2) over equation (4.1). First, FFS is much easier to solve both in theory and in practice. Second, as Theorem 3 shows, the approximation error of FFS is simply additive to the error inherent to the raw features. No such property is known for LFD. In the next chapter, we compare the two methods numerically.

CHAPTER 5

Empirical Evaluation

In this section, we empirically evaluate the quality of features generated by FFS both with and without using raw features. We focus on a comparison with LFD (Song et al., 2016) which was empirically shown to outperform radial basis functions (RBFs) (Lagoudakis & Parr, 2003), random projections (Ghavamzadeh, Lazaric, Maillard, & Munos, 2010), and other methods.

We first compare the quality of solutions on a range of synthetic randomly-generated problems. The goal is to ensure that the methods behave similarly regardless of the number of samples, or the type of raw features that are used. Then, we use an image-based version of the cart-pole benchmark, used previously by Song et al. (2016), to evaluate FFS in more complex settings. This problem is used to evaluate both the solution quality and the computational complexity of the methods.

5.1 Synthetic Problems

To compare FFS to other common approaches in feature selection, we start with small policy evaluation problems. Since the policy is fixed throughout these experiments, we omit all references to it. The data matrix $A \in \mathbb{R}^{n \times l}$ only contains the states where n denotes the number of states and l the length of each *raw* feature, with $\Phi \in \mathbb{R}^{n \times k}$ using k features.

The synthetic problems that we use throughout this section have 100 states. The rewards $\boldsymbol{r} \in \mathbb{R}^{100}$ are generated uniformly randomly from the interval of $[-500, 500)$. The stochastic transition probabilities $P \in [0,1)^{100 \times 100}$ are generated from the uniform Dirichlet distribution. To ensure that the rank of P is at most 40, we compute P as a product $P = XY$, where X and Y are small-dimensional. The discount factor we use is $\gamma = 0.95$.

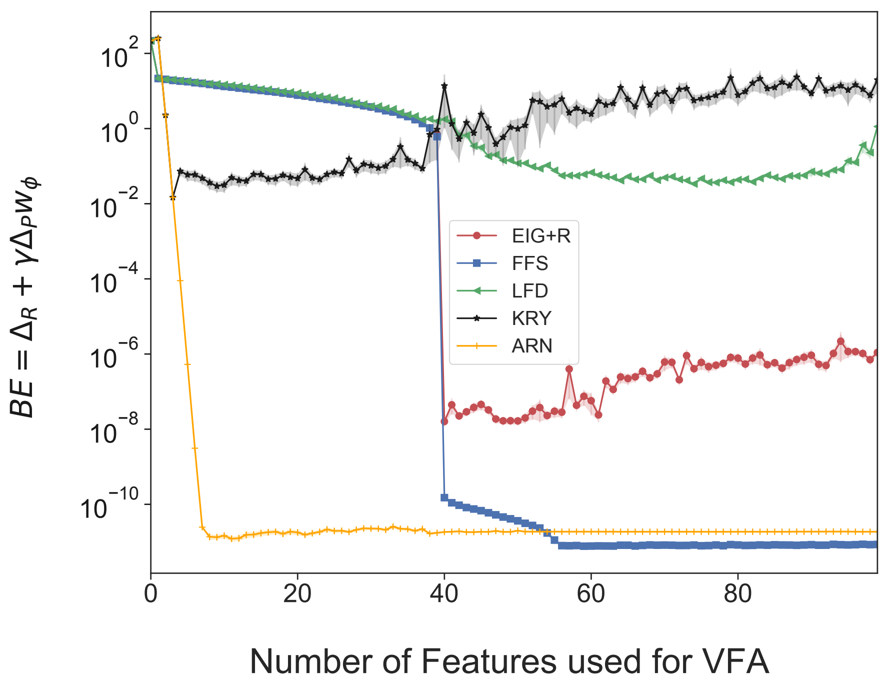

Figure 5-1: Bellman error for the exact solution. The transition matrix is 100×100 and has a low rank with $\text{rank}(P) = 40$. The Input matrix is $A = \mathbf{I}$ an identity matrix.

We now proceed by evaluating FFS for both tabular and image-based features. For the sake of consistency, we use FFS to refer to both FFS in a tabular case and FFS when raw features are available. To evaluate the quality of the value function approximation, we compute the Bellman residual of the fixed-point value function, which is a standard metric used for this purpose. Recall that the Bellman error can be expressed as

$$\text{BE} = \Delta_{\boldsymbol{r}} + \gamma \Delta_P \boldsymbol{w}_\Phi,$$

where $\boldsymbol{w}_\Phi$ is the value-function given in equation (2.18). All results we report in this section are an average of 100 repetitions of the experiments. All error plots show the L_2 norm of the Bellman error in logarithmic scale.

Case 1: Tabular raw features. In this case, the true transition probabilities P and the reward function $\boldsymbol{r}$ are known, and the raw features are an identity matrix: $A = \mathbf{I}$. Therefore all computations are made concerning the precise representations of the underlying MDP.

This is the simplest setting, under which SVD simply reduces to a direct low-rank approximation of the transition probabilities. That is, the SVD optimization problem reduces to:

$$\min_{U_1 \in \mathbb{R}^{n\times k}} \min_{\Sigma_1 V_1^\top \in \mathbb{R}^{k\times n}} \|U_1 \Sigma_1 V_1^\top - P\|_F^2 \ .$$

Similarly, the constructed features will be $\Phi = U_1$. In case of TFFS, we can simply add the reward vector to feature's set $\Phi = [U_1, \boldsymbol{r}]$. EIG+R and KRY are implemented as described in (Petrik, 2007; Parr et al., 2008). In case of EIG+R approach, we use the eigenvectors of P as basis functions, and then $\boldsymbol{r}$ is included. For Krylov basis we calculate $\Phi = \mathcal{K}_k(P, \boldsymbol{r})$. We also include the result of Arnoldi iteration (ARN) using Algorithm 1.

Figure 5-1 depicts the Bellman error for the exact solution when the number of features used for value function varies from 1 to 100. Note that the Bellman error of FFS is zero for $k \geq 40$. This is because the rank of P is 40, and according to Theorem 2 the first 40 features obtained by FFS are sufficient to get $\|\mathrm{BE}\|_2 = 0$. This experiment shows FFS is robust and generally outperforms other methods. The only exception is the Krylov method which is more effective when few features are used but is not numerically stable with more features. However, Arnoldi iteration does not suffer from this problem. The Krylov method could be combined relatively easily with FFS to get the best of both bases.

Case 2: Image-based raw features. In this case, the raw features A are not tabular but instead simulate an image representation of states. So the Markov dynamics are experienced only via samples and the functions are represented using an approximation scheme. The matrix A is created by randomly allocated zeros and ones similar to the structure of a binary image. We use LSTD to compute the approximate value function, as described in section 2.3.1.

The SVD optimization problem now changes as described in section 4.2. The constructed features will be $\Phi = A\widehat{\Phi}$ and for FFS we include the reward predictor vector $[P_A, \boldsymbol{r}_A]$ in the optimization problem. In the case of the EIG+R method, we multiply the eigenvectors of P_A and $\boldsymbol{r}_A$ with the raw features. The Krylov basis is constructed as: $\Phi = A\,\mathcal{K}_k(P_A, \boldsymbol{r}_A)$ where $\mathcal{K}_k$ is the k-th order Krylov operator.

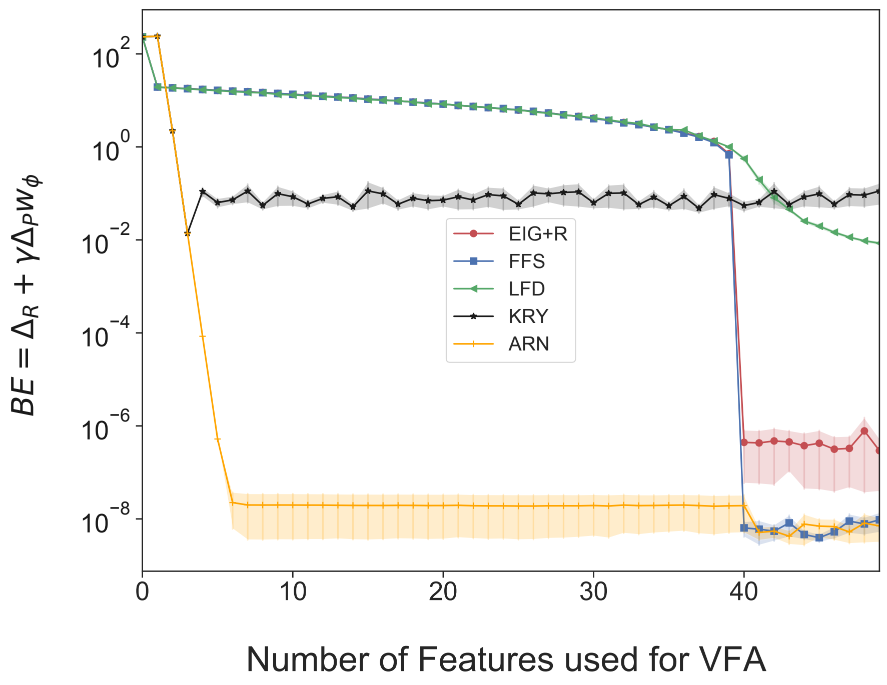

Figure 5-2: Bellman error for the approximate solution. The transition matrix is 100×100 and has a low rank with $\text{rank}(P) = 40$. The Input matrix is $A =$ random binary matrix.

Figure 5-2 compares the Bellman error for the approximate solution. FFS outperforms other methods but Arnoldi iteration (ARN). LFD is unstable when the number of features exceeds the rank of P, and sometimes it is not possible to obtain the pseudo-inverse of matrix AE.

It is worth noting that this section deals with very small MDPs with only about 100 states. It is expected to see a more significant gap in Bellman error of these methods when dealing with large MDPs with enormous and high-dimensional state spaces. In the next section, we compare LFD and FFS with Krylov subspace approach using a more significant and more challenging benchmark problem.

5.1.1 Contribution

We present an experimental analysis over the Bellman residual of feature selection with FFS and other relevant approaches, using small MDPs that are generated randomly.

5.2 Cart-Pole

These experiments evaluate the similarity between the Linear Feature Discovery (LFD) approach and the Fast Feature Selection (FFS) method on a modified version of cart-pole, which is a standard reinforcement learning benchmark problem. The controller must learn a good policy by merely observing the *image* of the cart-pole without direct observations of the angle and angular velocity of the pole. This problem is large enough that the computational time plays an important role, so we also compare the computational complexity of the three methods.

Note that this is a control benchmark, rather than value approximation for a fixed policy. Since the goal of RL is to optimize a policy, results on policy optimization are often more meaningful than just obtaining a small Bellman residual which is not sufficient to guarantee that a good policy will be computed (Johns et al., 2009).

To obtain training data, we collect the specified number of trajectories with the starting angle and angular velocity sampled uniformly on $[-0.1, 0.1]$. The cart position and velocity are set to zero at each episode.

The algorithm was given three consecutive, rendered, gray-scale images of the cart-pole. Each image is down sampled to 39×50 pixels, so the raw state is a $39 \times 50 \times 3 = 5850-$dimensional vector. We chose three frames to preserve the Markov property of states without manipulating the cart-pole simulator in OpenAI Gym. We used $k = 150$ features for all methods.

We follow a setup analogous to Song et al. (2016) by implementing least-squares policy iteration (Lagoudakis & Parr, 2003) to obtain the policy. The training data sets are produced by running the cart for $[10, 25, 50, 100, 150, 200, 400, 600]$ episodes with a random policy. We then run policy iteration to iterate up to 50 times or until there is no change in the $A' = P^{\pi} A$ matrix.

The state of the pole in the classic cart-pole problem is described by its angle and angular velocity. However, in the image-based implementation, the agent does not observe this information. Song et al. (2016) chose two successive frames to show the state of the pole. To preserve the Markovian property of the state, they had to modify the simulator and force the angular velocity to match the change in angle per time step $\dot{\theta} = (\theta' - \theta)/\delta t$. We, instead, use the standard simulator from OpenAI Gym and choose the last three consecutive frames rather than two. Three consecutive frames are sufficient to infer θ and $\dot{\theta}$ and construct a proper Markov state. Intriguingly, no linear feature construction methods work well in the original problem definition when using only the last two frames.

The performance of the learned policy is reported for 100 repetitions to obtain the average number of balancing steps. Figure 5-3 displays the average number of steps during which the pole kept its balance using the same training data sets. For each episode, a maximum of 200 steps was allowed to run. This result shows that on the larger training sets the policies obtained from FFS and KRY are quite similar. It worth noting that in our experiment both ARN and KRY produce the same quality controller. The results show that FFS and KRY outperform LFD significantly.

Figure 5-4 depicts the average running time of KRY, LFD and FFS for obtaining the value function with $k = 150$. The computation time of FFS grows very slowly as the number of training episodes increases; at 600 training episodes, the maximum number of episodes tested, FFS is 6 times faster than LFD. Therefore, LFD would likely be impractical in large problems with many training episodes.

Both FFS and LFD implementations use randomized SVD in all computations including the computation of pseudo-inverses. The result is usually very close to truncated singular value decomposition. Randomized SVD is fast on large matrices on which we need to extract only a small number of singular vectors. It reduces the time to compute k top singular values for an $m \times n$ matrix from $O(mnk)$ to $O(mn \log(k))$ (Halko, Martinsson, & Tropp, 2011).

In comparison to black box methods such as neural networks, linear value functions are more interpretable: their behavior is more transparent from an analysis standpoint and feature engineering standpoint. It is comparatively simple to gain some insight into the reasons for which a particular choice of features succeeds or fails. When the features are normalized, the magnitude of each parameter is related to the importance of the corresponding feature in the approximation (Lagoudakis & Parr, 2003).

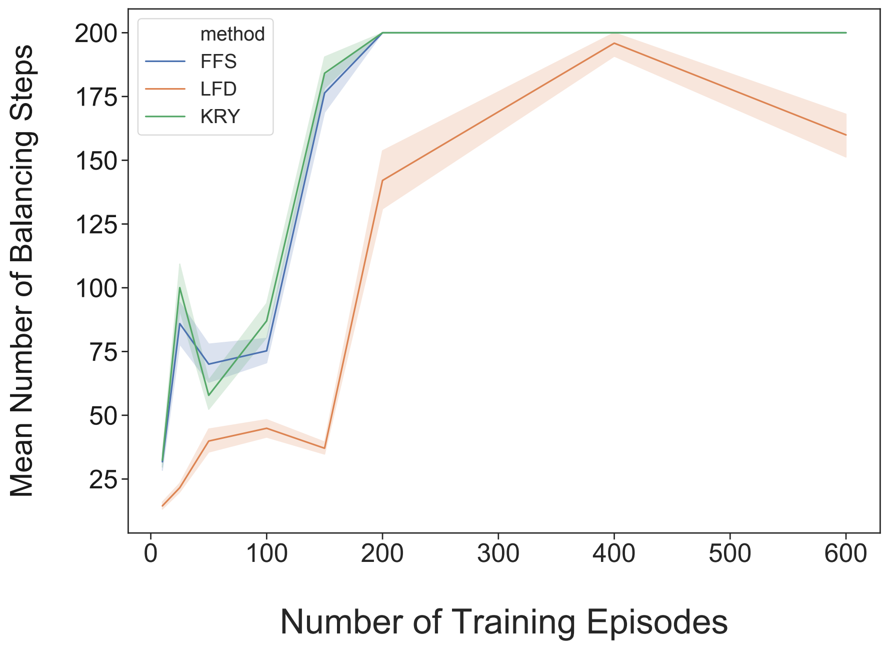

Figure 5-3: Average number of balancing steps with $k = 150$.

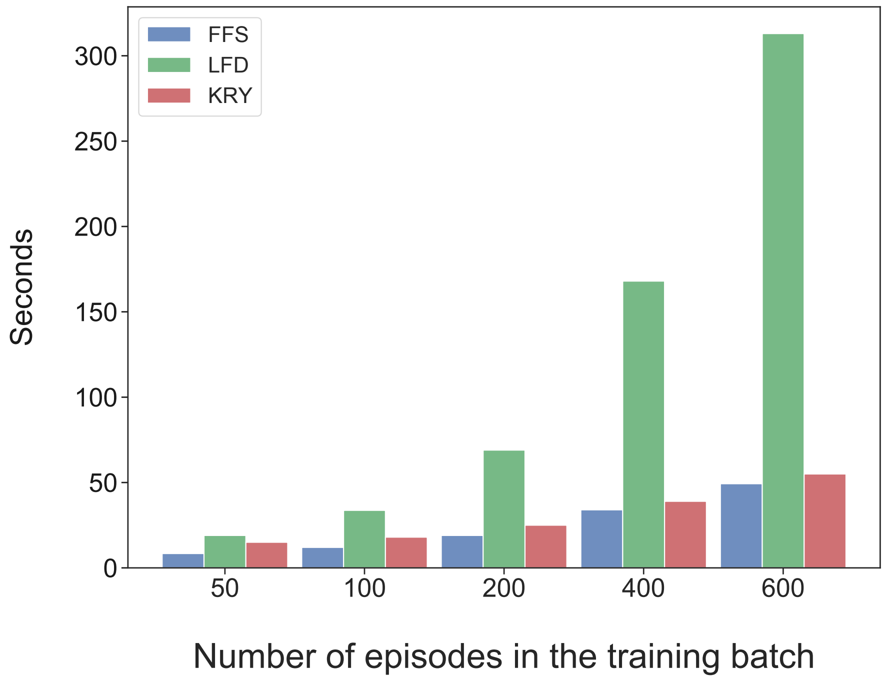

Figure 5-4: Mean running time for estimating the q-function with $k = 150$.

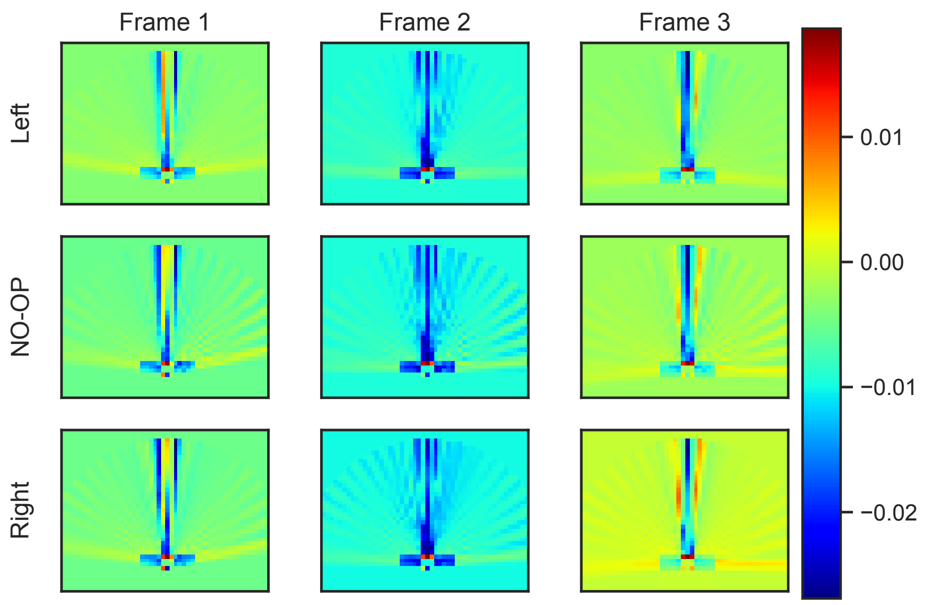

Figure 5-5: Value function in jet color-map.

Figure 5-5 shows the learned coefficients of q-function for three actions (left, right and no-op) using color codes. The q-values are obtained by the inner product of raw features (3-frames of cart-pole) and these coefficients. They are computed by the FFS method from 400 training episodes with random policy. In this experiment, the raw images, taken from the cart-pole environment in OpenAI Gym toolkit, are preprocessed, converted to gray-scale and normalized. Therefore, the pole in the raw images is in black, and the value of black pixels are close to zero. Other areas in the raw features are in white, so these pixel values are closer to one. It is interesting to see how the linear value function captures the dynamics of the pole (the cart is stationary). If the pole is imbalanced, the value function is smaller since the blue area in figure 5-5 represents negative scalars.

Figure 5-6 depicts the singular values of the compressed transition matrix $P_A = A^+ A'$ obtained from $[200, 400, 600]$ episodes training datasets. P_A is a 17550×17550 matrix considering three actions. The result shows in this example that the assumption behind FFS is not a strong assumption because the compressed transition probabilities matrix is low-rank.

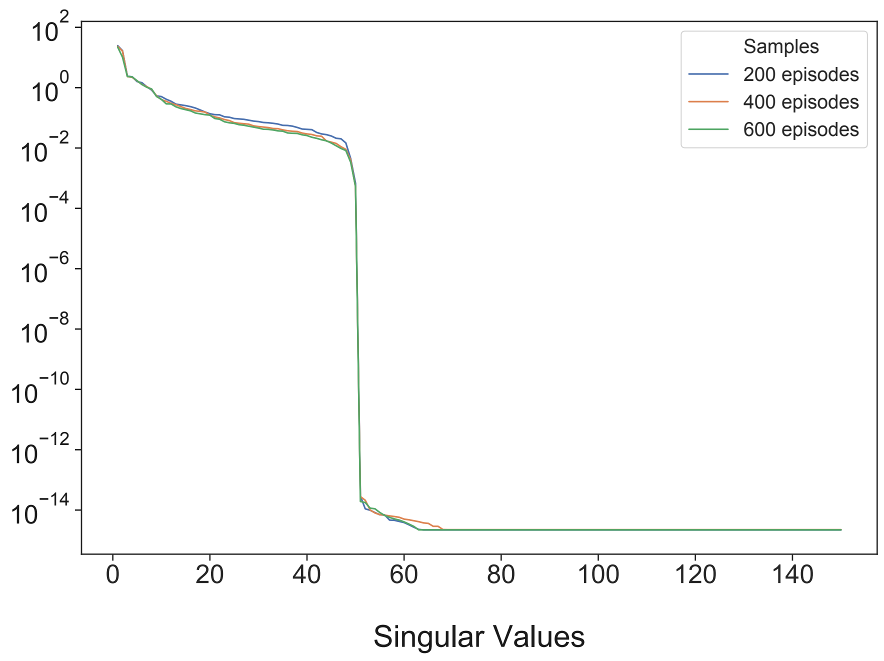

Figure 5-6: Singular values of P_A. The transition matrix is computed from datasets with 200, 400, and 600 and 800 episode samples. $y-$axis is in log scale.

5.2.1 Contribution

We compare the performance of our proposed method FFS with LFD and Krylov subspace approach to control a modified version of cart-pole which is a more challenging benchmark problem. In our setting the input data is made of raw images instead of angle and angular velocity of the pole. We also deliver some additional insights into the linear value function by visualizing the features obtained by FFS.

CHAPTER 6

Conclusion, Discussion and Future Directions

6.1 Future Work

It is important to study how common it is for reinforcement learning to exhibit the low-rank property exploited by FFS. This is most likely to vary depending on the domain, problem size, and amount of batch data. We suspect that in most applications, a combination of several methods, such as FFS and BEBF, is likely to yield the best and most robust results. Finally, it would be interesting to study the impact of FFS on finite-sample bounds and robustness in RL.

It is unlikely that linear feature selection techniques can be competitive with modern deep reinforcement learning. Linear feature selection is nevertheless essential. First, it can be used in conjunction with deep learning methods where the raw features are taken from the neural net. Second, linear features can be used to gain additional insights into the given reinforcement learning problem. These are issues that we are planning to address in future work.

6.2 Conclusion

We propose FFS, a new feature construction technique that computes a low-rank approximation of the transition probabilities. We believe that FFS is a promising method for feature selection in batch reinforcement learning. It is very simple to implement, fast to run, and relatively easy to analyze. A particular strength of FFS is that it is easy to judge its effectiveness by singular values of features not included in the approximation (Theorem 2).

FFS, as a result, addresses the central limitation of LFD and makes the algorithm more practical when solving large-scale problems. FFS also has better theoretical properties than LFD, and it is easier to implement and analyze. In our formulation, the low-rank approximation can be derived directly from an SVD of a compressed transition probabilities matrix without the need to solve a complex optimization problem.

Our empirical results show that FFS computes value functions that are better than LFD and also, in case of computational complexity, faster. Even in the cart-pole problem, which is a relatively simple benchmark problem, we observed up to almost six fold speedup in feature computation time. Also, while LFD needs to know the number of features to select in advance, the SVD-based approaches can select the number of features on the fly based on the decay of singular values.

Bibliography

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike daptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, pp. 834–846.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming* (1st edition). Athena Scientific.

Bradtke, S. J., & Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine learning*, *22*(1), 33–57.

Cheng, B., Asamov, T., & Powell, W. B. (2018). Low-rank value function approximation for co-optimization of battery storage. *IEEE Transactions on Smart Grid*, *9*(6), 6590–6598.

Coppersmith, D., & Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, *9*(3), 251–280.

Da Silva, B. C., Basso, E. W., Bazzan, A. L., & Engel, P. M. (2006). Dealing with non-stationary environments using context detection. In *Proceedings of the 23rd international conference on Machine learning*, pp. 217–224. ACM.

de Farias, D. P., & Van Roy, B. (2005). A linear program for bellman error minimization with performance guarantees. *Advances in Neural Information Processing Systems (NIPS)*, *17*.

Efron, B., Hastie, T., Johnstone, I., Tibshirani, R., et al. (2004). Least angle regression. *The Annals of statistics*, *32*(2), 407–499.

Ghavamzadeh, M., Lazaric, A., Maillard, O., & Munos, R. (2010). LSTD with random projections. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 721–729.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations*.

Halko, N., Martinsson, P.-G., & Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, *53*(2), 217–288.

Hastie, T., Mazumder, R., Lee, J. D., & Zadeh, R. (2015). Matrix completion and low-rank svd via fast alternating least squares. *The Journal of Machine Learning Research*, *16*(1), 3367–3402.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd edition).

Howard, R. A. (1960). *Dynamic programming and Markov processes.* MIT Press, Cambridge, Massachusetts.

Johns, J., Petrik, M., & Mahadevan, S. (2009). Hybrid least-squares algorithms for approximate policy evaluation. *Machine Learning*, *76*(2), 243–256.

Keller, P. W., Mannor, S., & Precup, D. (2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pp. 449–456. ACM.

Koller, D., & Parr, R. (2000). Policy iteration for factored mdps. In *Conference on Uncertainty in Artificial Intelligence*, pp. 326–334. Morgan Kaufmann Publishers Inc.

Kolter, J. Z., & Ng, A. Y. (2009). Regularization and feature selection in least-squares temporal difference learning. In *International Conference on Machine Learning (ICML)*, pp. 521–528.

Konidaris, G., Osentoski, S., & Thomas, P. S. (2011). Value function approximation in reinforcement learning using the fourier basis. In *Association for the Advancement of Artificial Intelligence (AAAI)*, Vol. 6, p. 7.

Lagoudakis, M. G., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research (JMLR)*, *4*(Dec), 1107–1149.

Lange, S., Gabel, T., & Riedmiller, M. (2012). Batch reinforcement learning. In *Reinforcement Learning*, pp. 45–73.

Le, L., Kumaraswamy, R., & White, M. (2017). Learning sparse representations in reinforcement learning with sparse coding. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2067–2073.

Li, S., Zhu, J., & Miao, C. (2015). A generative word embedding model and its low rank positive semidefinite solution. *Computing Research Repository (CoRR).*

Mahadevan, S. (2005). Proto-value functions: Developmental reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, pp. 553–560. ACM.

Mahadevan, S., & Maggioni, M. (2006). Value function approximation with diffusion wavelets and laplacian eigenfunctions. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 843–850. MIT Press.

Mahadevan, S., & Maggioni, M. (2007). Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research (JMLR)*, 2169–2231.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602.*

Munos, R. (2007). Performance bounds in l_p-norm for approximate value iteration. *SIAM Journal on Control and Optimization*, *46*(2), 541–561.

Murphy, K. P. (2012). *Machine Learning: A probabilistic perspective.* The MIT Press.

Oh, J., Guo, X., Lee, H., Lewis, R. L., & Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2863–2871.

Ong, H. Y. (2015). Value function approximation via low rank models. *Computing Research Repository (CoRR).*

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., & Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *International Conference on Machine Learning (ICML).*

Parr, R., Painter-Wakefield, C., Li, L., & Littman, M. (2007). Analyzing feature generation for value-function approximation. In *International Conference on Machine Learning (ICML*, pp. 737–744.

Petrik, M. (2007). An analysis of Laplacian methods for value function approximation in MDPs. In *International Joint Conference on Artificial Intelligence*, Vol. 35, pp. 2574–2579.

Petrik, M., Taylor, G., Parr, R., & Zilberstein, S. (2010). Feature selection using regularization in approximate linear programs for markov decision processes. In *International Conference on Machine Learning (ICML)*, pp. 871–878.

Puterman, M. L. (2014). *Markov decision processes: Discrete stochastic dynamic programming.* John Wiley & Sons.

Rendle, S., Freudenthaler, C., & Schmidt-Thieme, L. (2010). Factorizing personalized markov chains for next-basket recommendation. In *International Conference on World Wide Web (WWW).*

Schweitzer, P. J., & Seidmann, A. (1985). Generalized polynomial approximations In Markovian decision processes. *Journal of Mathematical Analysis and Applications*, *110*, 568–582.

Song, Z., Parr, R. E., Liao, X., & Carin, L. (2016). Linear feature encoding for reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 4224–4232.

Strang, G. (2009). *Introduction to linear algebra* (Fourth Edition edition).

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

Sutton, R. S., & Barto, A. G. (1984). Temporal credit assignment in reinforcement learning..

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

Szepesvári, C. (2010). *Algorithms for reinforcement learning*. Morgan & Claypool Publishers.

Williams, R. J., & Baird, L. C. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep., College of Computer Science, Northeastern University.