Batch Reinforcement Learning

Sascha Lange, Thomas Gabel, Martin Riedmiller

Note: This is a preprint version of the chapter on "Batch Reinforcement Learning" as part of the book "Reinforcement Learning: State-of-the-Art" (2012) compiled by Marco Wiering and Martijn van Otterlo. Cite this chapter as:

S. Lange, T. Gabel, M. Riedmiller (2012) Batch Reinforcement Learning. In: M. Wiering, M. van Otterlo (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, Berlin, Heidelberg

1 Introduction

Batch reinforcement learning is a subfield of dynamic programming (DP) based reinforcement learning (RL) that has vastly grown in importance during the last years. Historically, the term 'batch RL' is used to describe a reinforcement learning setting, where the complete amount of learning experience—usually a set of transitions sampled from the system—is a priori given and fixed. The task of the learning system then is to derive a solution—usually an optimal policy—out of this given batch of samples.

In the following, we will relax this assumption of an a priori fixed set of training experience. The crucial benefit of batch algorithms lies in the way they handle a batch of transitions and get the best out of it rather than in the fact, that this set is fixed. From this perspective, batch RL algorithms are characterized by two basic ingredients: all observed transitions are stored and updates occur synchronously on the whole batch of transitions ('fitting'). In particular, this allows the definition of 'growing batch' methods, that are allowed to extend the set of sample experience, in order to incrementally improve their solution. From the interaction perspective, the growing batch approach reduces the difference between batch methods and pure online learning methods.

The benefits coming with the batch idea, stability and data-efficiency of the learning process, account for the large interest in batch algorithms. Whereas basic algo-

rithms like Q-learning usually need many interactions until convergence to good policies, thus often rendering a direct application to real applications impossible, methods including ideas from batch reinforcement learning usually converge in a fraction of the time. Recently, a number of successful examples of applying ideas originating from batch RL to learning in the interaction with real-world systems have been published (see sections 1.5.2 and 1.6).

In this chapter, we will first define the batch reinforcement learning problem and its variants, that is the problem space treated by batch RL methods. We will then give a brief historical recap of the development of the central ideas that, in retrospect, build the foundation of all modern batch RL algorithms. On the basis of the problem definition and the introduced ideas, we will present the most important algorithms in batch RL. We will discuss their theoretical properties and also name some variations that have a high relevance for practical applications. This includes a treatment of Neural Fitted Q-Iteration (NFQ) and some of its applications, as it has proven a powerful tool for learning on real systems. With the application of batch methods to visual learning of control policies and to solving distributed scheduling problems we will briefly discuss further ongoing research.

1.1 The Batch Reinforcement Learning Problem

Batch reinforcement learning historically was defined as the class of algorithms developed for solving a particular learning problem, namely the batch reinforcement learning problem.

1.1.1 The Batch Learning Problem

As in the general reinforcement learning problem defined by Sutton and Barto [38], the task in the batch learning problem is to find a policy that maximizes the sum of expected rewards in the familiar agent-environment loop. But differing from the general case, in the batch learning problem the agent itself is not allowed to interact with the system during learning. Instead of observing a state s, trying an action a and changing its policy according to the subsequent following state s' and reward r, the learner only receives a set $\mathcal{F} = \{(s_t, a_t, r_{t+1}, s_{t+1}) | t = 1, \ldots, p\}$ of p transitions (s, a, r, s') sampled from the environment.

In the most general case of this batch reinforcement learning problem, the learner cannot make any assumptions on the sampling procedure of the transitions. They may be sampled by an arbitrary—even pure random—policy, they are not necessarily sampled uniformly from the sate-action space $S \times A$, they must not even be sampled along connected trajectories. Using only this information, the learner has to come up with a policy that is then used by the agent to interact with the environment. During this application phase, the policy is fixed and not further improved as new observations come in. Since the learner itself is not allowed to interact with

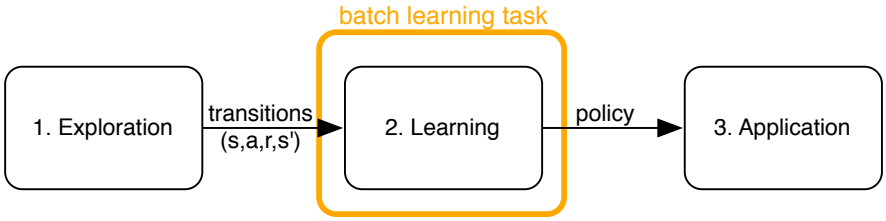

Fig. 1 The three distinct phases of the batch reinforcement learning process: 1: Collecting transitions with an arbitrary sampling strategy. 2: Application of (batch) Reinforcement Learning algorithms in order to learn the best possible policy from the set of transitions. 3: Application of the learned policy. Exploration is not part of the batch learning task. During the application phase, that isn't part of the learning task either, policies stay fixed and are not improved further.

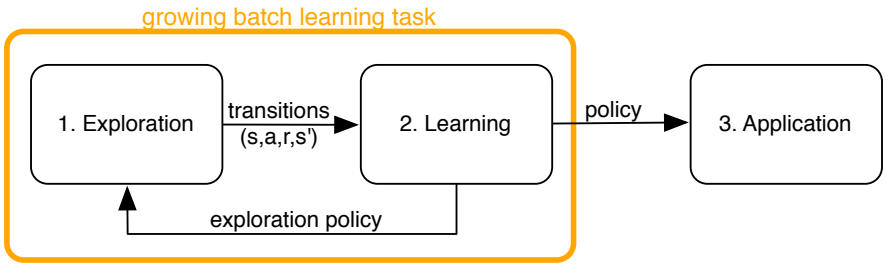

Fig. 2 The growing batch reinforcement learning process has the same three phases as the 'pure' batch learning process depicted in figure 1. But differing from the pure batch process, the growing batch learning process alternates for several times between the exploration and the learning phase, thus incrementally 'grows' the batch of stored transitions using intermediate policies.

the environment, and the given set of transitions usually is finite, the learner cannot be expected to always come up with an optimal policy. The objective therefore has been changed from learning an optimal policy as in the general reinforcement learning case to deriving the best possible policy from the given data.

The sharp separation of the whole procedure into three phases—exploring the environment and collecting state transitions and rewards, learning a policy and application of the learned policy—their sequential nature and the data passed at the interfaces is further clarified in figure 1. Obviously, treatment of the exploration–exploitation dilemma is not subject of algorithms solving such a pure batch learning problem, as the exploration is not part of the learning task at all.

1.1.2 The Growing Batch Learning Problem

Although this batch reinforcement learning problem historically has been the start of the development of batch reinforcement learning algorithms, modern batch RL algorithms are seldomly used in this 'pure' batch learning problem. In practice, exploration has an important impact on the quality of the policies that can be learned.

Obviously, the distribution of transitions in the provided batch must resemble the 'true' transition probabilities of the system in order to allow the derivation of good policies. The easiest way to achieve this is to sample the training examples from the system itself by interacting with it. But when sampling from the real system, another aspect becomes important: the covering of the state space by the transitions used for learning. If 'important regions'— e.g. states close to the goal state— are not covered by any or not enough samples, then it is obviously not possible to learn a good policy from the data, since important information is missing. This is a real problem, as in practice, an completely 'uninformed' policy—e.g. a pure random policy—is often not able to achieve a good covering of the state space, especially, in case of attractive starting states and hard to reach desirable states. Often, it is necessary to already have a rough idea of a good policy in order to be able to explore interesting regions that are not in the direct vicinity of the starting states.

This is the main reason, why in practice a third variant of the reinforcement learning problem inbetween pure online problem and the pure batch problem became popular. We will refer to it as the 'growing batch' learning problem, since the main idea is alternating between phases of exploration, where set of training examples is grown by interacting with the system, and phases of learning, where the whole batch of observations is used (see fig. 2). In the literature, this growing batch approach can be found in several different flavors; the number of alternations between episodes of exploration and episodes of learning can be in the whole range of being as close to the pure batch approach as using only two iterations [34] to recalculating the policy after every few interactions—e.g. after finishing one episode in a shortest-path problem [19, 24]. In practice, the growing batch approach is the modeling of choice, when applying batch reinforcement learning algorithms to real systems. Since the growing batch approach from the interaction perspective is very similar to the 'pure' online approach—the agent improves its policy *while* interacting with the system—the interaction perspective with the distinction between 'online' and 'offline' isn't that useful anymore for identifying batch RL algorithms. When talking about 'batch' RL algorithms now, it's more important to look at the algorithms and search for typical properties of the specific update rules.

1.2 Foundation of Batch RL Algorithms

Model-free online learning methods like Q-learning are conceptually appealing and have been very successful when applied to problems with small, discrete state spaces. But when it comes to applying them to more realistic systems with larger and possible continuous state spaces, these algorithms came up against limiting factors. In this respect, there can be identified three independent problems:

1. the 'exploration overhead' causing slow learning in practice
2. inefficiencies due to the stochastic approximation
3. stability issues when using function approximation

A common factor in modern batch RL algorithms is that typically, these algorithms address all three issues and come up with specific solutions to each of them. In the following we will discuss these problems in more detail and present the proposed solutions (in historical order) that now form the defining ideas behind modern batch reinforcement learning.

The Idea of Experience Replay for Addressing the Exploration Overhead

In order to explain the problem referred to as 'exploration overhead', let us for a moment consider the common Q-update rule for model-free online learning as given by

$$Q'(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \right] \tag{1}$$

(see chapter **??**). In pure online Q-learning the agent practically alternates between learning and exploring with every single time step: in state s the agent selects and executes an action a, and then, when observing the subsequent state s' and reward r, it immediately updates the policy according to (1), afterwards forgetting the 'experienced' state transition tuple (s,a,r,s'). It then returns to exploring with the updated policy $Q'(s,a)$. Although this approach is guaranteed to converge in the limit, there is a severe performance problem with these 'local' updates along actually experienced transitions. When for example updating the q-value of state-action pair (s_t,a_t) in time step t this may influence the values (s_{t-1},a) for all $a \in A$ of a preceding state s_{t-1}. But this change will not back-propagate immediately to all the involved preceding states, as the states preceding s_t are only updated the next time they are eventually visited. And states preceding those preceding states s_{t-1} even need another trial afterwards to be adapted to the new value of s_{t-1}. The performance problem with this is that these interactions actually are not needed to collect more information from the system in the first place, but mainly are needed to spread already available information through the whole state space in reverse order along trajectories. In practice, many interactions in Q-learning are of this 'spreading' type of interaction. This becomes even worse, when considering that those updates in model-free Q-learning are 'physical' in the sense of needing real interaction with the system and seldomly can be sampled in ordered sweeps over the whole state space or at least according to a uniform distribution.

In order to overcome this performance issue, Lin introduced the idea of 'experience replay' [27]. Although the intention was to solve the 'exploration overhead' problem in online learning, in retrospect, we might identify it as a basic but nevertheless first technique for addressing the growing batch problem. The experience replay idea is to speed up convergence by not only using observed state transitions—the experience—once, but replaying them repeatedly to the agent as if they were new observations collected while interacting with the system. Practically, one can store only a few to as many as all transitions observed so far and use them to up-

date the Q-function for every single stored transition according to (1) after every interaction with the system. The stored transitions are exactly the information that is missing in basic online Q-learning to be able to back-propagate information from updated states to preceding states without further interaction. The transitions collected when using experience replay resemble the connection between individual states and make more efficient use of this information, spreading information along these connections and ideally speeding up convergence.

The Idea of 'Fitting' to Address Stability Issues

In online RL it is common to use 'asynchronous' updates in the sense, that the value function is updated immediately after each observation locally, for that particular state, leaving all other states untouched. In the discrete case, this means updating one single Q-value for a state action pair (s,a) in Q-learning, 'over-writing the' value of the starting state of the transition immediately. Subsequent updates would use this updated value for their own update. This idea was also used with function approximating, first doing a DP-update like e.g.

$$\bar{q}_{s,a} = r + \gamma \max_{a' \in A} f(s',a') \tag{2}$$

recalculating the approximated value of the present state action pair $\bar{q}_{s,a}$ by e.g adding immediate reward and approximated value of the subsequent state and then immediately 'storing' the new value in the function approximator in the sense of moving the value of the approximation slightly towards the value of the new estimate $\bar{q}_{s,a}$ for the state-action pair (s,a):

$$f'(s,a) \leftarrow (1-\alpha)\, f(s,a) + \alpha\, \bar{q}_{s,a}\ . \tag{3}$$

Please note, that (2) and (3) are just a re-arranged form of equation (1), using an arbitrary function approximation scheme for calculation an approximation f' of the 'true' Q-value function Q'.

Baird, Gordon and others have shown examples, where particular combinations of Q-learning and similar updates with function approximators behave instable or even lead to sure divergence [2, 15]. Stable behavior could be proven only for particular instances of combinations of approximation scheme and update rule or under particular circumstances and assumptions on the system and reward structure [36]. In practice it needed a lot of experience by the engineer, in order to get a particular learning algorithm to work on a system. The observed stability issues are related to the interdependency of errors made in function approximation and deviations of the estimated value function from the optimal value function. Whereas the DP-update (2) tries to gradually decrease the difference between $Q(s,a)$ and the optimal Q-function $Q^*(s,a)$, storing the updated value in the function approximator in step (3) might (re-)introduce a possibly larger error. This approximation error again influences all subsequent DP-updates and may work against the contraction or even pre-

vent it. The problem becomes even worse, when global function approximators like e.g. multi-layer perceptrons are used; improving a single Q-value of a state-action pair might impair all other approximations through the whole state space.

In this situation Gordon [13] came up with the compelling idea of modifying the update scheme slightly in order to separate the dynamic programming step from the function approximation step. The idea is to first apply a DP-update to all members of a set of so-called 'supports' —points distributed throughout the state space— calculate new 'target values' for all supports like in (2), and then use supervised learning to train ('fit') a function approximator on all these new target values replacing the local updates of (3) [13]. In this sense, the estimated Q-function is updated 'synchronously', updating it at all supports at the same time. Although Gordon introduced this fitting idea in the setting of model-based value iteration, this idea became the foundation and perhaps even the starting point of all modern batch algorithms.

Replacing Inefficient Stochastic Approximation

Gordon discussed the possibility of transfering the fitting idea to model-free—sample-based—approaches like e.g. Q-learing, but did not find a solution and saw several convergence issues when estimating the values at the supports from samples in their surrounding. Ormoneit and Sen finally came up with the solution of how to transfer his idea to the sample-based case. In their work [30] they proposed to not use arbitrarily selected supports in the state space to approximate value functions, but to use the sampled transitions directly to approximate the value function—either at the starting or the ending states of the transitions with the help of a kernel-based approximator. Their idea is to calculate an estimate of the value of each explored state action pair under the actually observed transition (reward plus expected value of subsequent state) and then estimate the values of the subsequent states by averaging over the values of nearby transitions. Technically, the trick was to replace the exact DP-operator that was inroduced by Gordon with a non-exact random operator (see section 1.3.1 for a formal definition). This random operator does not use the exact models in the DP-step but only estimates the exact DP-operation by using a random sampling of transitions drawn from the unknown transition models. Lagoudakis and Parr independently came up with a similar idea [21, 22].

As a side effect, calculating the costs of particular transitions and estimating the values of states (and actions) by averaging over data, solved another performance problem that is connected to the stochastic approximation in regular Q-learning. Whereas in model-based value iteration the value of a state according to the current values of it's possible subsequent states can be updated in a single step using the transition model, in Q-learning—due to the replacement of the model by stochastic approximation—such an update needs many visits to the state in question. Moreover, since we are using one learning rate for the whole state space, this learning rate cannot be adjusted in a way that is optimal for all states depending on the number of visits, thus reaching the optimal convergence rate in practice is impossible. The algorithm of Ormoneit and Sen does not rely on stochastic approximation but

implicitly estimates the transition model by averaging over the observed transitions that—if collected in interaction with the system—actually form a random sampling from the true distributions.

1.3 Algorithms

In their work, Ormoneit and Sen proposed a 'unified' kernel-based algorithm that actually could be seen as a general framework used by several later algorithms. Their 'kernel-based approximate dynamic programming' (KADP) brought together the ideas of experience replay (storing and re-using experience), fitting (separation of DP-operator and approximation) and kernel-based self-approximation (sample-based).

1.3.1 Kernel-Based Approximate Dynamic Programming

Ormoneit's kernel-based approximative dynamic programming tries to solve an approximated version of the 'exact' Bellman-equation $V = HV$ that is expressed in $\hat{V} = \hat{H}\hat{V}$. It not only uses an approximation $\hat{V}$ of the 'true' state value function V—as Gordons fitted value iteration did—but also uses an approximative version $\hat{H}$ of the exact DP-operator H itself. The KADP algorithm to solve this equation works as follows.

Starting from an arbitrary initial approximation $\hat{V}^0$ of the state-value function, each iteration i of the KADP-algorithm consists of solving the equation $\hat{V}^{i+1} = H_{max}\hat{H}^a_{dp}\hat{V}^i$ for a given set $\mathscr{F} = \{(s_t, a_t, r_{t+1}, s_{t+1}) | t = 1, \ldots, p\}$ of p transitions (s,a,r,s'). In this equation, the approximate DP-operator $\hat{H} = H_{max}\hat{H}^a_{dp}$ has been split into an exact part H_{max} maximizing over actions and an approximate random operator $\hat{H}^a_{dp}$ approximating the 'true' (model-based) DP-step for individual actions from the observed transitions. The first half of this equation is calculated according to the sample-based dp-update

$$\hat{Q}^{i+1}_a(\sigma) := \hat{H}^a_{dp}\hat{V}^i(\sigma) = \sum_{(s,a,r,s') \in \mathscr{F}_a} k(s,\sigma)\left[r + \gamma \hat{V}^i(s')\right] . \tag{4}$$

This equation calculates a weighted average of the well-known Q-updates $r + \gamma\hat{V}^i(s') = r + \gamma \max_{a' \in A} \hat{Q}^i(s', a')$—please note the similarity to equation (2)—along all transitions $(s,a,r,s') \in \mathscr{F}_a$, where $\mathscr{F}_a \subset \mathscr{F}$ is the subset of $\mathscr{F}$ that contains only the transitions $(s,a,r,s') \in \mathscr{F}$ that used the particular action a. The second half of the equation applies the maximizing-operator H_{max} to the approximated q-functions $\hat{Q}^{i+1}_a$:

$$\hat{V}^{i+1}(s) = H_{max}\hat{Q}^{i+1}_a(s) = \max_{a \in A} \hat{Q}^{i+1}_a(s) . \tag{5}$$

Please note, this algorithm uses an individual approximation $\hat{Q}_a^{i+1} : S \mapsto R$ for each action $a \in A$ in order to approximate the Q-function $Q_a^{i+1} : S \times A \mapsto R$. Furthermore, a little bit counter-intuitively, in a practical implementation, this last equation is actually evaluated and stored for all ending states s' of all the transitions $(s,a,r,s') \in \mathscr{F}$—not the starting states s. This decision is explained by noting that in the right-hand side of equation (4) we only query the present estimate of the value function at the ending states of the transitions, never its starting states (see [30, section 4]). This becomes more obvious when looking at figure 3.

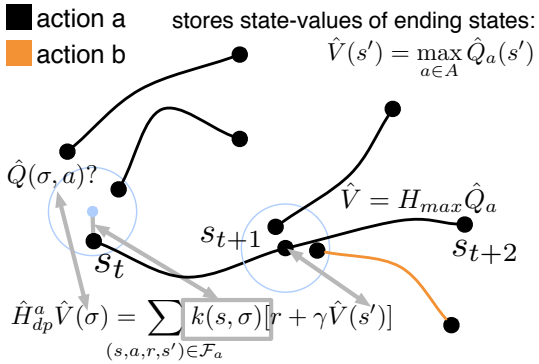

Fig. 3 Visualization of the kernel-based approximation in KADP. For computing the q-value $\hat{Q}(\sigma,a) = \hat{H}^a_{dp}(\sigma)$ of (σ,a) at an arbitrary state $\sigma \in S$ KADP uses the starting states s of nearby transitions (s,a,r,s') only for calculating the weighting factors $k(s,\sigma)$ but depends on the state values $\hat{V}(s')$ of ending states s' (in the depicted example $s = s_t$ and $s' = s_{t+1}$).

Using weighting-kernels that adhere to the 'averager' restriction from Gordon, thus using weights that sum up to one

$$\sum_{(s,a,r,s') \in \mathscr{F}_a} k(s,\sigma) = 1 \quad \forall \sigma \in S$$

and are non-negative

$$k(s,\sigma) \geq 0 \quad \forall \sigma \in S \;\; \forall (s,a,r,s') \in \mathscr{F}_a \,,$$

iterating over (4-5), explicitly storing the values $\hat{V}^i(s')$ of $\hat{V}^i$ at the end-points s' of the transitions (s,a,r,s') will finally converge to a unique solution.

The decision to store and iterate over state-values instead of state-action values results in the need for applying another DP-step in order to derive a policy from the result of the KADP algorithm:

$$\pi^{i+1}(\sigma) = \arg\max_{a \in A} \hat{H}^a_{dp}\hat{V}^i(\sigma) \tag{6}$$

$$= \arg\max_{a \in A} \sum_{(s,a,r,s') \in \mathscr{F}_a} k(s,\sigma)\left[r + \gamma \hat{V}^i(s')\right] \,. \tag{7}$$

This might be a small problem from an efficiency point of view—as you have more complex computations as in q-learning and need to remember all transitions in the application phase—but is not a theoretical problem, as applying the DP-operator to the fixed point of $\hat{V} = \hat{H}\hat{V}$ does not change anything but results in the same unique fixed point.

1.3.2 Fitted Q Iteration

Perhaps the most popular algorithm in batch RL is Damien Ernst's 'Fitted Q Iteration' [8]. It can be seen as the 'Q-Learning of batch RL', as it actually is a straightforward transfer of the basic Q-learning update-rule to the batch case. Given a fixed set $\mathscr{F} = \{(s_t, a_t, r_{t+1}, s_{t+1}) | t = 1, \ldots, p\}$ of p transitions (s,a,r,s') and an initial Q-value $\bar{q}^0$—Ernst used $\bar{q}^0 = 0$ [8]—the algorithm starts by initializing an initial approximation $\hat{Q}^0$ of the Q-function Q^0 with $\hat{Q}^0(s,a) = \bar{q}^0$ for all $(s,a) \in S \times A$. It then iterates over the following two steps:

1. Start with an empty set P^{i+1} of patterns $(s,a;\bar{q}^{i+1}_{s,a})$. For each transition $(s,a,r,s') \in \mathscr{F}$ calculate a new target Q-value $\bar{q}^{i+1}_{s,a}$ according to

$$\bar{q}^{i+1}_{s,a} = r + \gamma \max_{a' \in A} \hat{Q}^i(s',a') \tag{8}$$

   (similar to the update (2)) and add a corresponding pattern $(s,a;\bar{q}^{i+1}_{s,a})$ to the pattern set; thus $P \leftarrow P \cup \{(s,a;\bar{q}^{i+1}_{s,a})\}$.
2. Use supervised learning to train a function approximator on the pattern set P^{i+1}. The resulting function $\hat{Q}^{i+1}$ then is an approximation of the Q-function Q^{i+1} after $i+1$ steps of dynamic programming.

Originally, Ernst proposed randomized trees for approximating the value function. After fixing their structure, these trees can be also represented as kernel-based averagers, thus reducing step 2 to

$$\hat{Q}^{i+1}_a(\sigma) = \sum_{(s,a;\bar{q}^{i+1}_{s,a}) \in P^{i+1}_a} k(s,\sigma)\bar{q}^{i+1}_{s,a} \ , \tag{9}$$

with the weights $k(\cdot,\sigma)$ being determined by the structure of the tree. This variant of FQI, constructs an individual approximation $\hat{Q}^{i+1}_a$ for each discrete action $a \in A$ that together form the approximation $\hat{Q}^{i+1}(s,a) = \hat{Q}^{i+1}_a(s)$ [8, section 3.4]. Besides this variant of FQI, Ernst also proposed a variant with continuous actions. We may refer the interested reader to [8] for a detailed description of this.

From a theoretical stand-point, fitted Q-iteration nevertheless is based on Ormoneit and Sen's theoretical framework. The similarity between fitted Q-iteration and KADP becomes obvious, when re-arranging equations (8-9):

$$\hat{Q}^{i+1}(\sigma,a) = \hat{Q}^{i+1}_a(\sigma) = \sum_{(s,a;\bar{q}) \in P^{i+1}_a} k(s,\sigma)\bar{q}^{i+1}_{s,a} \tag{10}$$

$$= \sum_{(s,a,r,s') \in \mathcal{F}_a} k(s,\sigma) \left[r + \gamma \max_{a' \in A} \hat{Q}^i_{a'}(s') \right] . \qquad (11)$$

Equation (10) is the original averaging step (9) in FQI for discrete actions. By inserting FQI's DP-step (8) immediately follows (11). This result (11) is practically identical to the update used in KADP, as can be seen by inserting (5) into (4):

$$\hat{Q}^{i+1}_a(\sigma) = \sum_{(s,a,r,s') \in \mathcal{F}_a} k(s,\sigma) \left[r + \gamma \hat{V}^i(s') \right] \qquad (12)$$

$$= \sum_{(s,a,r,s') \in \mathcal{F}_a} k(s,\sigma) \left[r + \gamma \max_{a \in A} \hat{Q}^i_a(s') \right] . \qquad (13)$$

Besides the optional treatment of continuous actions, another difference between FQI and KADP is in the splitting of operators and in the choice of explicitly represented values—where KADP explicitly represents and uses state-values in the DP-step (4), FQI explicitly represents the Q-function and calculates state-values $\hat{V}^i(s) = max_{a \in A} \hat{Q}^i(s,a)$ on the fly by maximizing over actions in its DP-step (8). Although 'lazy-learning' with kernel-based averaging, as proposed as the standard in KADP, is also allowed in FQI, Ernst assumes the usage of a trained averager or other parametric function approximator for explicitly storing the Q-function. The 'structure' of this approximator is neither necessarily related to the starting or ending points of transitions and since it represents the Q-function explicitly, deriving a policy in FQI is rather simple:

$$\pi^i(s) = \arg\max_{a \in A} \hat{Q}^i(s,a) \qquad (14)$$

where $\hat{Q}^i$ is realized explicitly, either by a single function approximator (continuous actions) or by a set of function approximators $\hat{Q}^i_a$ for the actions $a \in A$. These subtle differences in otherwise equivalent algorithms make FQI more intuitive, and more similar to the online approach of Q-learning, what might explain this algorithm's greater popularity.

1.3.3 Least-Squares Policy Iteration

Least-squares policy iteration (LSPI, [22]) is another early representative of a batch mode reinforcement learning algorithm. In contrast to the other algorithms reviewed in this section, LSPI explicitly embeds the task of solving control problems into the framework of policy iteration [38], thus alternating between policy evaluation and policy improvement steps. However, LSPI never stores a policy explicitly. Instead it works solely on the basis of a state-action value function Q from which a greedy policy is to be derived via $\pi(s) = \arg\max_{a \in A} Q(s,a)$. For the purpose of representing state-action value functions, LSPI employs a parametric linear approximation architecture with a fixed set of k pre-defined basis functions $\phi_i : S \times A$ and a weight vector $w = (w_1, \ldots, w_k)^T$. Hence, any approximated state-action value function $\hat{Q}$ in

the scope of LSPI takes the form

$$\hat{Q}(s,a,;w) = \sum_{j=1}^{k} \phi_j(s,a) w_j = \Phi w^T. \tag{15}$$

Its policy evaluation step employs a least-squares temporal difference learning algorithm for the state-action value function (LSQ [21], later called LSTDQ [22]). This algorithm takes as input the current policy π_m—as pointed out above, represented by a set of weights that determine a value function $\hat{Q}$ from which π_m is to be derived greedily—as well as a finite set $\mathscr{F}$ of transitions (s,a,r,s'). From these inputs, LSTDQ derives analytically the state-action value function $\hat{Q}^{\pi_m}$ for the given policy under the state distribution determined by the transition set. Clearly, the derived value function returned $\hat{Q}^{\pi_m}$ by LSTDQ is, again, fully described by a weight vector w^{π_m} given the above-mentioned linear architecture used.

Generally, the searched for value function is a fixed point of the $\hat{H}_\pi$ operator

$$(\hat{H}_\pi Q)(s,a) = r(s,\pi(s)) + \gamma \sum_{s' \in S} p(s,\pi(s),s') \max_{b \in A} Q(s,b), \tag{16}$$

i.e. $\hat{H}_{\pi_m} Q^\pi = Q^{\pi_m}$, thus a good approximation $\hat{Q}^{\pi_m}$ should comply to $\hat{H}_{\pi_m} \hat{Q}^{\pi_m} \approx \hat{Q}^{\pi_m} = \Phi (w^{\pi_m})^T$. Practically, LSTDQ aims at finding a vector w^π such that the approximation of the result of applying the $\hat{H}_{\pi_m}$ operator to $\hat{Q}_{\pi_m}$ is as near as possible to the true result (in an L_2 norm minimizing manner). For this, LSTDQ employs an orthogonal projection and sets

$$\hat{Q}^{\pi_m} = (\Phi(\Phi^T\Phi)^{-1}\Phi^T)\hat{H}\hat{Q}^{\pi_m}. \tag{17}$$

With the compactly written version of Equation 16, $\hat{H}_{\pi_m} Q^{\pi_m} = \mathscr{R} + \gamma \mathscr{P} \Pi_{\pi_m} Q^{\pi_m}$, Equation 17 can be rearranged to

$$w^{\pi_m} = (\Phi^T(\Phi - \gamma \mathscr{P} \Pi_{\pi_m} \Phi))^{-1} \Phi^T \mathscr{R} \tag{18}$$

where Π_{π_m} is a stochastic matrix of size $|S| \times |S||A|$ describing policy π_m: $\Pi_{\pi_m}(s,(s',a')) = \pi_m(s',a')$. Importantly, in this equation LSTDQ approximates the model of the system on the basis of the sample set $\mathscr{F}$ given, i.e. P is a stochastic matrix of size $|S||A| \times |S|$ that contains transition probabilities as observed within the transition set according to $P((s,a),s') = \sum_{(s,a,\cdot,s') \in \mathscr{F}} 1 / \sum_{(s,a,\cdot,\cdot) \in \mathscr{F}} 1 \approx Pr(s,a,s')$ and $\mathscr{R}$ is a vector of size $|S||A|$ that summarizes the rewards contained in $\mathscr{F}$.

After having determined the state-action value function $\hat{Q}^{\pi_m}$ for the current policy π_m, a greedy (improved) policy π_{m+1} can be derived as usual by letting

$$\pi_{m+1}(s) = \arg\max_{a \in A} \hat{Q}^{\pi}_m(s,a) = \arg\max_{a \in A} \phi(s,a)(w^{\pi_m})^T \tag{19}$$

Since LSPI never stores policies explicitly, but implicitly by the set of basis functions and corresponding weights—and thus, in fact, by a state-action value function—

the policy improvement step of LSPI merely consists of overwriting the old weight vector w by the current weight vector found by a call to LSTDQ.

It is standing to reason to compare a single iteration of the fitted Q iteration algorithm with a single iteration (one policy evaluation and improvement step) of LSPI. The main difference between LSPI and value function-centered fitted Q iteration algorithms is that, in a single iteration, LSPI determines an approximation of the state-action value function Q^{π_m} for the current policy and batch of experience. LSPI can do this analytically, i.e. without iterating the $\hat{H}_{\pi_m}$ operator, because of the properties of its linear function approximation architecture. By contrast, FQI algorithms rely on a set of target values for the supervised fitting of a function approximator which are based on a single dynamic programming update step, i.e. on a single application of the $\hat{H}$ operator. Consequently, if we interpret fitted Q iteration algorithms from a policy iteration perspective, then this class of algorithms implements a batch variant of optimistic policy iteration, whereas LSPI realized standard (non-optimistic) policy iteration.

1.3.4 Identifying Batch Algorithms

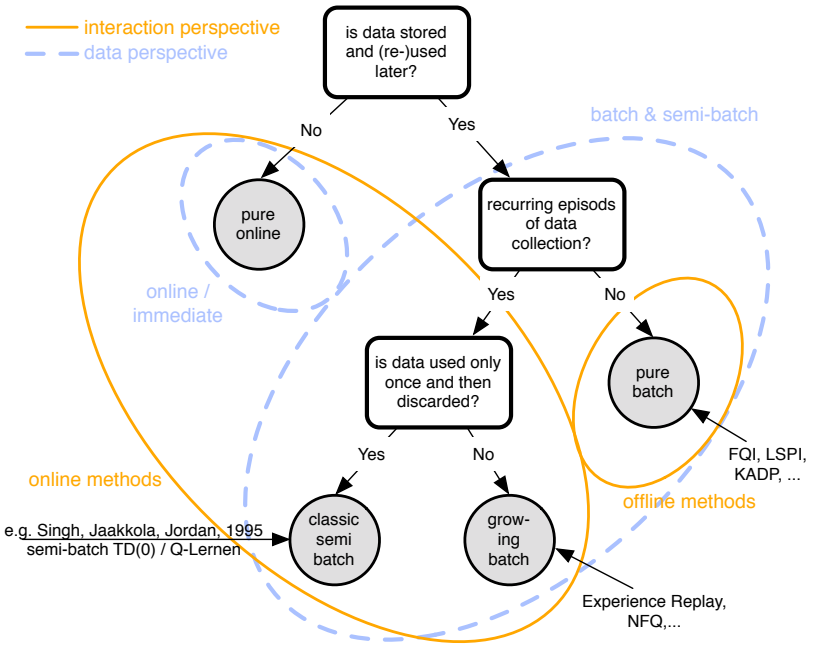

Fig. 4 Classification of batch vs. non-batch algorithms. With the interaction perspective and the data-usage perspective there are at least two different perspectives to define the category borders.

Whereas the algorithms described here, could be seen as the foundation of modern batch reinforcement learning, several other algorithms have been referred to as 'batch' or 'semi-batch' algorithms in the past. Furthermore, the borders between 'online', 'offline', 'semi-batch' and 'batch' can not be drawn distinctly—there are at least two different perspectives to look at the problem. Figure 4 proposes an ordering of online, semi-batch, growing batch and batch reinforcement learning algorithms. At the one side of the tree we have pure online algorithms like classic Q-learning and on the opposite side of the tree we have pure batch algorithms that work completely 'offline' on a fixed set of transitions. Inbetween these extremal positions are a number of other algorithms, that could be classified as either online or (semi-)batch algorithms, depending on the perspective. For example, the growing batch approach could be either classified as an online method—it interacts with the system like an online method and incrementally improves its policy as new experience becomes available—but at the same time from a data usage perspective could be seen as a batch-algorithm, since it stores all experience and uses 'batch methods' to learn from these observations. Whereas FQI like KADP and LSPI has been proposed by Ernst as a pure batch algorithm working on a fixed set of samples, it can easily be adapted to the growing batch setting, as e.g. shown in [19]. This holds for every 'pure' batch approach. On the other hand, NFQ (see section 1.5.1) that has been introduced in a growing-batch setting, can also be adapted to the pure batch setting straight-forward. Another class is formed by the 'semi-batch' algorithms that have been introduced mainly for formal reasons in the 90's [37]. These algorithms make an aggregate update for several transitions—so its not pure online learning with immediate updates—but they do not store and reuse the experience after making this update—so its not a full batch approach either.

1.4 Theory of Batch RL

The compelling feature of the batch RL approach is that it grants stable behavior for q-learning like update rules and a whole class of function approximators (averagers) in a broad number of systems, independent of a particular modeling or specific reward function. There are two aspects to discuss: a) stability in the sense of guaranteed convergence to a solution and b) quality in the sense of the distance of this solution to the true optimal value function.

Gordon introduced the important notion of the 'averager' [13, 14] and proved convergence of his model-based fitted value iteration for this class of function approximation schemes by showing their non-expansive properties (in maximum norm) and then relying on the classical contraction argument [4] for MDPs with discounted rewards. For non-discounted problems he identified a more restrictive class of compatible function approximators and proved convergence for the 'self-weighted' averagers [14, section 4]. Ormoneit and Sen extended these proofs to the model-free case. Their kernel-based approximators are equivalent to the 'averagers' introduced by Gordon [30]. Approximated values must be a weighted average of the

samples, where all weights are positive and sum up to one (see section 1.3.1). These requirements grant the non-expansive property in maximum norm. Ormoneit and Sen showed that their random dynamic programming operator using kernel-based approximation contracts the approximated function in maximum norm for any given set of samples and, thus, converges to a unique fixed point in the space of possible approximated functions. The proof has been carried out explicitly for MDPs with discounted rewards [30] and for average-cost problems [28, 29].

The quality of the solution found by the algorithms is another aspect. Gordon gave an absolute upper bound on the distance of the fixed point of his fitted value iteration to the optimal value function [14]. This bound depends mainly on the expressiveness of the function approximator and its 'compatibility' with the optimal value function to approximate. Besides the function approximator, in model-free batch reinforcement learning the random sampling of the transitions obviously is another aspect that influences the quality of the solution. Therefore, for KADP there is no absolute upper bound limiting the distance of the approximate solution given a particular function approximator. Ormoneit and Sen instead proved the stochastic consistency of their algorithm what actually could be seen as an even stronger statement. Continuously increasing the size of samples in the limit guarantees stochastic convergence to the optimal value function under certain assumptions [30]. These assumptions [30, appendix A] besides other constraints on the sampling of transitions include smoothness constraints on the reward function (needs to be a Lipschitz continuous function of s, a and s') and the kernel. A particular kernel that fulfills these constraints and is used throughout their experiments [28] is derived from the 'mother kernel'

$$k_{\mathscr{F}_a,b}(s,\sigma) = \phi^+\left(\frac{||s-\sigma||}{b}\right) \Bigg/ \sum_{(s_i,a_i,r_i,s_i')\in\mathscr{F}_a} \phi^+\left(\frac{||s_i-\sigma||}{b}\right) \tag{20}$$

with ϕ^+ being a univariate Gaussian function. The parameter b controls the 'bandwidth' of the kernel, that is its region of influence or simply its 'resolution'. Relying on such a kernel, the main idea of their consistency proof is to first define an 'admissible' reduction rate for the parameter b in dependence of the growing number of samples and then proof the stochastic convergence of the series of approximations $\hat{V}^k$ under this reduction rate to the optimal value function. Reducing the bandwidth parameter b can be interpreted as increasing the resolution of the approximator. When reducing b, the expected deviation of the implicitly estimated transition model from the true transition probabilities—in the limit—vanishes to zero as more and more samples become available. It is important to note that increasing the resolution of the approximator only is guaranteed to improve the approximation for smooth reward functions and will not necessarily help for, e.g., approximating step functions—thus the Lipschitz constraint on the reward function.

Besides these results that are limited to the usage of averagers within the batch RL algorithms, there are promising new theoretical analysis of Antos, Munos and Szepesvari [1], that presently do not cover more general function approximators but may lead to helpful results for non-averagers in the future.

1.5 Batch RL in Practice

1.5.1 Neural Fitted Q-Iteration (NFQ)

The ability to approximate functions with high accuracy and to generalize well from few training examples, makes neural networks, in particular multi-layer perceptrons an attractive candidate to represent value functions. However, in the classical online reinforcement learning setting, the current update has often unforeseeable influence on the efforts taken so far. In contrast, batch RL changes the situation dramatically: by updating the value function simultaneously at all transitions seen so far, the effect of destroying previous efforts can be overcome. This was the driving idea behind the proposition of Neural Fitted Q-Iteration, NFQ [31]. As a second important consequence, the simultaneous update at all training instances makes the application of batch supervised learning algorithms possible. In particular, within the NFQ framework, the adaptive supervised learning algorithm Rprop [32] is used as the core of the fitting step.

The implementation of the batch RL framework using neural networks is mostly straight-forward, as algorithm 5 shows. However, there are some additional tricks and techniques that help to overcome some of the problems that occur, when approximating (Q-)value functions by multi-layer perceptrons:

- scaling input and target values is crucial for success, and should always be done when using neural networks. A sensible scaling can be easily realized, since all training patterns are known at the beginning of training.
- adding artificial training patterns (also called 'hint-to-goal'-heuristic in [31]). Since the neural network generalizes from collected experiences, it can be observed that the network output tends to increase to its maximum value, if no or too few goal-state experiences with zero path costs are included in the pattern set. A simple method to overcome this problem, is to build additional artificial (i.e. not observed) patterns within the goal region with target value zero, that literally 'clamp' the neural network output in that region to 0. For many problems this method is highly effective and can be easily applied. No extra knowledge is used to apply this method, when the target region is known, which is typically the case.
- the 'Qmin-heuristic': normalizing' the 'Q' target values [17]. A second method to work against the effect of increasing output values is to do a normalization step by subtracting the lowest target value from all target values. This results in a pattern set that has a target value of 0 for at least one training pattern. This method has the advantage, that no additional knowledge about states in the target regions need to be known in advance.
- using a smooth immediate cost-function [17]. Since multi-layer perceptrons do basically a smooth mapping from input to outputs, it is reasonable to also use a smooth immediate cost function. As an example, consider the immediate cost function that gives constant positive costs outside the target region and 0 costs inside the target region. This leads to a minimum time control behavior, which is

favorable in many applications. However, the path costs accordingly have crispy jumps, which are kind of difficult to represent by a neural network. Replacing this immediate cost function by a smoothed version, the main characteristic of the policy induced by the crisp immediate cost function is widely preserved while the value function approximation is much smoother. For more details, please see [17].

NFQ_main() {
input: a set of transition samples D; output: Q-value function Q_N
i=0
init_MLP() $\rightarrow Q_0$;
DO {
generate_pattern_set $P = \{(input_t; target_t), t = 1, \ldots, \#D\}$ where:
$input_t = s_t, a_t,$
$target_t = c(s_t, a_t, s'_t) + \gamma \min_{a' \in A} Q_i(s'_t, a')$
add_artificial_patterns(P)
normalize_target_values(P)
scale_pattern_values(P)
Rprop_training(P) $\rightarrow Q_{i+1}$
$i \leftarrow i+1$
} WHILE ($i < N$)

Fig. 5 Main loop of NFQ.

1.5.2 NFQ in Control Applications

Fig. 6 Brainstormers MidSize league robot. The difficulty of dribbling lies in the fact, that by the rules at most one third of the ball might be covered by the robot. Not loosing the ball while turning therefore requires a sophisticated control of the robot motion.

Applying reinforcement learning to the control of technical processes is particularly appealing, since it promises to autonomously learn optimal or near optimal controllers even in the presence of noise or nonlinearities without even knowing process behavior in advance. The introduction of batch RL has contributed to a major breakthrough in this domain, since due to its data efficiency, it is now possible to learn complex control behavior from scratch by directly interacting with the real system. Some recent examples of NFQ in real-world applications are learning to swing-up and balance a real cart-pole system, time optimal position control of pneumatic devices or learning to accurately steer a real car within less than half an hour of driving [35].

The following briefly describes learning of a neural dribble controller for a RoboCup MidSize League robot (for more details, see also [33]). The autonomous robot (figure 6) uses a camera as its main sensor and has an omnidirectional drive. The control interval is 33 ms. Each motor command consists of three values denoting v_y^{target} (target forward speed relative to the coordinate system of the robot), v_x^{target} (target lateral speed) and v_θ^{target} (target rotation speed).

Dribbling means to keep the ball in front of the robot, while turning to a given target. Since by the rules of the MidSize league only one-third of the ball might be covered by a dribbling device this is quite challenging: the dribbling behavior must carefully control the robot such that the ball is not running away from the robot when the robot changes direction.

The learning problem is modelled as a stochastic shortest path problem with both a terminal goal state and terminal failure states. Intermediate steps are punished by constant costs of 0.01. NFQ is used as the core learning algorithm. The computation of the target value for the batch training set thus becomes:

$$Q^{target}(s,a) := \begin{cases} 1.0 & , \quad \text{if } s' \in S^- \\ 0.01 & , \quad \text{if } s' \in S^+ \\ 0.01 + \min_b \tilde{Q}(s',b) & , \quad else \end{cases} \tag{21}$$

where S^- denotes the states, where the ball is lost, and S^+ denotes the states, where the robot has the ball and heads towards the target. State information contains speed of the robot in relative x and y direction, rotation speed, x and y ball position relative to the robot and finally the heading direction relative to the given target direction. A failure state $s \in S^-$ is encountered, if the ball relative x coordinate is larger than 50 mm or less than -50 mm, or the relative y coordinate exceeds 100 mm. A success state is reached whenever the absolute difference between the heading angle and the target angle is less than 5 degrees.

The robot is controlled by a three-dimensional action vector, denoting target translational and rotational speeds. Overall, 5 different action triples are used, $U = \{(2.0, 0.0, 2.0), (2.5, 0.0, 1.5), (2.5, 1.5, 1.5), (3.0, 1.0, 1.0), (3.0, -1.0, 1.0)\}$, where each triple denotes $(v_x^{target}, v_y^{target}, v_\theta^{target})$.

Input to the Neural Fitted Q Iteration method is a set of transition triples of the form (state, action, successor state). A common procedure to sample these transitions is to alternatively train the Q-function and then sample new transitions episodewise by greedily exploiting the current Q-function. However, on the real robot, this

means, that between each data collection phase one has to wait until the new Q-function has been trained. This can be annoying, since putting the ball back on the play-field requires human interaction. Therefore, a batch-sampling method is used, which collects data over multiple trials without relearning.

The value function is represented by a multi-layer perceptron with 9 input units (6 state variables and 3 action variables), 2 hidden layers of 20 neurons each and 1 output neuron. After each batch of 12 trials, 10 NFQ iterations are performed. Learning the target values was done in 300 epochs of supervised batch learning, using the Rprop learning method with standard parameters. After learning was finished, the new controller was used to control the robot during the next data collection phase. After 11 batches (= 132 trials), a very good controller was learned. The complete learning procedure took about 1 and a half hour, including the time used for offline updating the neural Q function. The actual interaction time with the real robot was about 30 minutes, including preparation phases.

The neural dribbling skill performed significantly better than the previously used hand-coded and hand-tuned dribbling routine, particularly in terms of space and time needed to turn to the desired target direction (see figure 7). The neural dribbling skill is successfully used in the Brainstormers competition team since 2007. Using it, the Brainstormers won the RoboCup world championship 2007 in Atlanta and became third at the world championship in 2008 in Suzhou, China.

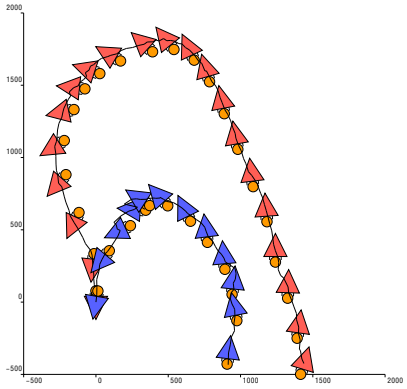

Fig. 7 Comparison of hand-coded (red) and neural dribbling behavior (blue) when requested to make a U-turn. The data was collected on the real robot. When the robot gets the ball, it typically has an initial speed of about 1.5 to 2 m/s in forward direction. The position of the robots are displayed every 120 ms. The U-turn performed by the neural dribbling controller is much sharper and faster.

1.5.3 Batch RL for Learning in Multi-Agent Systems

While the previous two sections have pointed to the advantages of combining the data-efficiency of batch-mode RL with neural network-based function approximation schemes, this section elaborates on the benefits of batch methods for cooperative multi-agent reinforcement learning. Assuming independently learning agents, it is obvious that single transition tuples some agent experiences are strongly affected by the actions taken concurrently by other agents. This dependency of single transitions on external factors, i.e. on other agents' policies, gives rise to another argument for batch training: While a single transition tuple contains probably too little information for doing a reliable update, a rather comprehensive batch of experience may contain sufficient information to apply value function-based RL in a multi-agent context.

The framework of decentralized Markov decision processes (DEC-(PO)MDP, [3]) is frequently used to address environments populated with independent agents that have access to local state information only and thus do not know about the full, global state. The agents are independent of one another both in terms of acting as well as learning. Solving problems of this type optimally is generally intractable which is why a meaningful goal is to find approximate joint policies for the ensemble of agents using model-free reinforcement learning. To this end, a local state-action value function $Q_k : S_k \times A_k$ is defined for each agent k that it successively computes and improves, and that it uses to choose its local actions.

In a straightforward approach, a batch RL algorithm—in the following, the focus is put on the use of NFQ—might be run independently by each of the learning agents, thus disregarding the possible existence of other agents and making no attempts to enforce coordination across them. This approach can be interpreted as an 'averaging projection' where the state-action values of of state-action pairs where agents behave cooperatively are mingled with the values of those where they do not cooperate. As a consequence, the agents' local Q_k functions underestimate the optimal joint Q function. The following briefly describes a batch RL-based approach to sidestep that problem and points to a practical application where the resulting multi-agent learning procedure has been employed successfully (for more details, see also [11]).

For a better estimation of the Q_k values, the inter-agent coordination mechanism introduced in [26] can be used and integrated into the framework of fitted Q iteration. Here, the basic idea is that each agent always optimistically assumes that all other agents behave optimally (though they often will not, e.g. due to exploration). Updates to the value function and policy learned are only done when an agent is certain that a superior joint action has been executed. The performance of that coordination scheme quickly degrades in the presence of noise, which is why determinism in the DEC-MDP's state transitions must be assumed during the phase of collecting transitions. However, this assumption can be dropped, when applying the policies learned.

For the multi-agent case, step 1 of FQI (cf. Equation 8) is modified: Each agent k collects its own transition set $\mathcal{F}_k$ with local transitions (s_k, a_k, r_k, s'_k). It then cre-

ates a reduced (so-called 'optimistic') training pattern set $\mathscr{O}_k$ such that $|\mathscr{O}_k| \leq |P_k|$. Given a deterministic environment and the ability to reset the system to a specific initial state during data collection, the probability that agent k enters some s_k more than once is larger than zero. Hence, if a certain action $a_k \in A_k$ has been taken multiple times in s_k, it may—because of differing local actions selected by other agents—have yielded very different rewards and local successor states for k. Instead of considering all tuples from $\mathscr{F}_k$, only those are used for creating $\mathscr{O}_k$ that have resulted in maximal expected rewards. This means, we assume that all other agents take their best possible local action, which are, when combined with a_k, most suitable for the current global state. Accordingly, the optimistic target Q-values $q^{i+1}_{s_k,a_k}$ for a given local state-action pair (s_k,a_k) of agent k is computed according to

$$q^{i+1}_{s_k,a_k} := \max_{\substack{(s,a,r,s')\in\mathscr{F}_k,\\ s=s_k,a=a_k}} \left(r+\gamma \max_{b\in A_k} \hat{Q}^i_k(s',b) \right). \tag{22}$$

Consequently, $\mathscr{O}_k$ realizes a partitioning of $\mathscr{F}_k$ with respect to identical values of s_k and a_k, and $q^{i+1}_{s_k,a_k}$ is the maximal sum of the immediate rewards and discounted expected rewards over all tuples $(s_k,a_k,\cdot,\cdot) \in \mathscr{F}_k$.

There are many applications that fit to the problem class outlined, including production planning and resource allocation problems [12]. One particular class of problems that can be cast as a decentralized MDPs are job-shop scheduling problems [6]. Here, the goal is to allocate a specified number of jobs (also called tasks) to a limited number resources (also called machines) in such a manner that some specific objective is optimized. Taking a batch reinforcement learning approach for scheduling, a learning agent is attached to each of the resources, receives local information about the set of jobs waiting for processing—features characterizing the jobs which are inputs to the neural networks used for value function approximation—, and must decide which job to dispatch next. The agents interact repeatedly with the scheduling plant and build up their own batches of transitions which are then used to derive an improved dispatching policy by learning agent-specific state-action value functions $\hat{Q}_k$ using the NFQ adaptation outlined above. For well-established benchmark problems, the learned dispatching policies yield competitive performance, and, moreover, exhibit good generalization capabilities, meaning that they can be immediately applied to modified factory layouts [10].

1.5.4 Deep Fitted Q-Iteration

Present reinforcement learning algorithms in general are still limited to solving tasks with state spaces of rather low dimensionality. For example, learning policies directly from high-dimensional visual input—e.g. raw images as captured by a camera—is still far from being possible. Usually in such a task, the engineer provides a method for extracting the relevant information from the high-dimensional inputs and for encoding it in a low-dimensional feature space of a feasible size. The learning algorithm is then applied to this manually constructed feature space.

Here, the introduction of batch reinforcement learning provides new opportunities for dealing with high-dimensional state spaces directly. Consider a set of transitions $\mathscr{F} = \{(s_t, a_t, r_{t+1}, s_{t+1}) | t = 1, \ldots, p\}$ where the states s are elements of a high-dimensional state space $s \in R^n$. The idea is, to use an appropriate unsupervised learning method for learning a feature extraction mapping from the data automatically. Ideally, the learned mapping $\phi : R^n \mapsto R^m$ with $m \ll n$ should encode all the relevant information contained in a state s in the resulting feature vectors $z = \phi(s)$. The interesting thing now is, that by relying on the batch RL methods, we can combine learning of feature spaces and learning a policy within a stable and data-efficient algorithm, learning both at the same time. Within the growing batch approach, when starting a new learning phase, we would first use the data $\mathscr{F}$ to learn a new feature extraction mapping $\phi : R^n \mapsto R^m$ and then learn a policy in this feature space. This is done by first mapping all samples from the state space to the feature space, constructing a pattern set $\mathscr{F}_\phi \{(\phi(s), a, r, \phi(s')) | (s, a, r, s') \in \mathscr{F}\}$in the feature space and then applying a batch algorithm such as FQI. Since all experiences are stored in the growing batch approach, it is possible to change the mapping after each episode of exploration and improve it with the newly available data, and then immediately calculate a new approximation of the value function from the mapped transitions. Actually, using the set of transitions $\mathscr{F}$ it is possible to directly 'translate' an approximation $\hat{Q}^\phi$ that was calculated for the feature extraction ϕ to a new feature extraction ϕ' without loosing any information. We would simply apply one step of FQI with a slightly modified calculation of the target values:

$$\bar{q}_{\phi'(s),a} = r + \gamma \max_{a' \in A} \hat{Q}^\phi_{a'}(\phi(s')) \; . \tag{23}$$

When calculating the new target value for $\bar{q}_{\phi'(s),a}$ for the new feature vector $\phi'(s)$ of state s, this update looks up the expected reward from the subsequent state s' in the old approximation $\hat{Q}^\phi$ using the feature vector $\phi(s')$ in the old feature space. These target values are then used to calculate a new approximation $\hat{Q}^{\phi'}$ for the new feature space.

We already have implemented this idea in a new algorithm named 'Deep Fitted Q-Iteration' (DFQ) [24, 25]. DFQ uses a deep auto-encoder neural network [18] with up to millions of weights for unsupervised learning of low-dimensional feature spaces from high dimensional visual inputs. Training of these neural networks is embedded in a growing batch reinforcement learning algorithm derived from Fitted Q-Iteration, thus enabling learning of feasible feature spaces and useful control policies at the same time (see figure 8). By relying to kernel-based averagers for approximating the value function in the automatically constructed feature spaces, DFQ inherits the stable learning behavior from the batch methods. Extending the theoretical results of Ormoneit and Sen, the inner loop of DFQ could be shown to converge to a unique solution for any given set of samples of any MDP with discounted rewards [23].

The DFQ algorithm has been successfully applied to learning visual control policies in a grid-world benchmark problem—using synthesized [25] as well as screen-

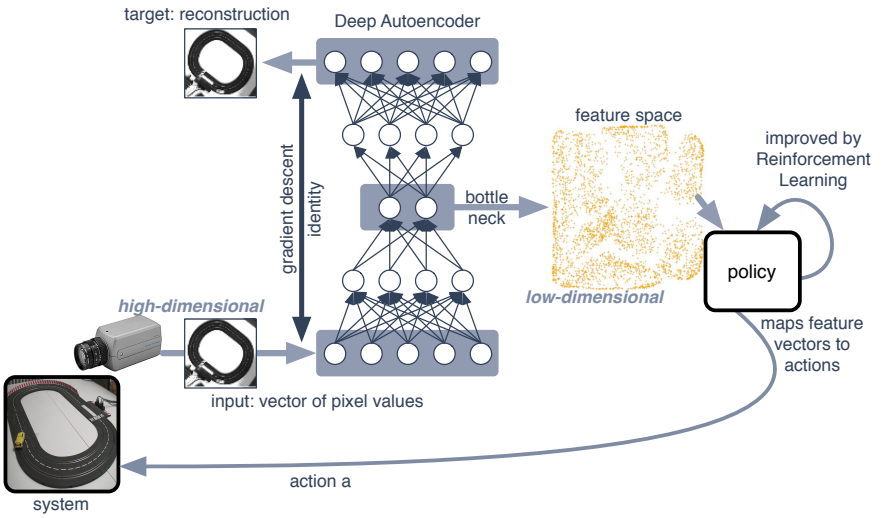

Fig. 8 Schematic drawing of Deep Fitted Q-Iteration. Raw visual inputs from the system are feed into a deep auto-encoder neural network that learns to extract a low-dimensional encoding of the relevant information in its bottle-neck layer. The resulting feature vectors are then used to learn policies with the help of batch updates.

captured images [24]—and to controlling a slot-car racer just on the basis of the raw image data captured by a top-mounted camera [23].

1.6 Applications/ Further References

The following table provides a quick overview of recent applications of batch RL methods.

1.7 Summary

This chapter has reviewed both the historical roots and early algorithms as well as contemporary approaches and applications of batch-mode reinforcement learning. Research activity in this field has grown substantially in the recent years, primarily due to the main merits of the batch approach, namely its efficient use of collected data as well as the stability of the learning process caused by the separation of the dynamic programming and value function approximation steps. Besides this, various practical implementations and applications for real-world learning tasks have contributed to the increased interest in batch RL approaches.

Domain	Description	Method	Reference
Technical process control, real world	Slot car racing	NFQ	[20]
Technical process control, real world	Dribbling soccer robot	NFQ	[33]
Technical process control, real world	Steering an autonomous car	NFQ	[35]
Technical process control, simulation	Nonlinear control benchmarks	NFQ, NFQCA	[17]
Technical process control, simulation	Pole swing up and balancing	Batch RL, Gaussian Processes	[7]
Technical process control, simulation	Mobile wheeled pendulum	NFQ, FQI (Extra Trees)	[5]
Technical process control, simulation	Semi-active suspension control	Tree based batch RL	[40]
Technical process control, simulation	Control of a power system, comparison to MPC	FQI (Extra Trees)	[9]
Portfolio management, simulation	Managing financial transactions	KADP	[28]
Benchmarking, simulation	Mountain car, acrobot	FQI, CMAC	[39]
Multi agent systems, simulation	Decentralized scheduling policies	NFQ	[10]
Multi agent systems, simulation	Keepaway soccer	FQI (NN, CMAC)	[19]
Medical applications	Treatment of Epilepsy	Tree based batch RL	[16]

Table 1 Overview of applications of batch RL methods

References

1. A. Antos, R. Munos, and C. Szepesvari. Fitted Q-iteration in continuous action-space MDPs. *Advances in neural information processing systems*, 20:9–16, 2008.
2. L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proc. of the 12th International Conference on Machine Learning*, pages 30–37, 1995.
3. D. Bernstein, D. Givan, N. Immerman, and S. Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.
4. D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic programming*. Belmont, MA: Athena Scientific, 1996.
5. Andrea Bonarini, Claudio Caccia, Alessandro Lazaric, and Marcello Restelli. Batch reinforcement learning for controlling a mobile wheeled pendulum robot. In *IFIP AI*, pages 151–160, 2008.
6. P. Brucker and S. Knust. *Complex Scheduling*. Springer, Berlin, Germany, 2005.
7. Marc P. Deisenroth, Carl E. Rasmussen, and Jan Peters. Gaussian Process Dynamic Programming. *Neurocomputing*, 72(7–9):1508–1524, March 2009.
8. D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(1):503–556, 2005.
9. Damien Ernst, Mevludin Glavic, Florin Capitanescu, and Louis Wehenkel. Reinforcement learning versus model predictive control: a comparison on a power system problem. *Trans. Sys. Man Cyber. Part B*, 39(2):517–529, 2009.

10. T. Gabel and M. Riedmiller. Adaptive Reactive Job-Shop Scheduling with Reinforcement Learning Agents. *International Journal of Information Technology and Intelligent Computing*, 24(4), 2008.
11. T. Gabel and M. Riedmiller. Evaluation of Batch-Mode Reinforcement Learning Methods for Solving DEC-MDPs with Changing Action Sets. In *Proceedings of the 8th European Workshop on Reinforcement Learning (EWRL 2008)*, pages 82–95, Lille, France, 2008. Springer.
12. T. Gabel and M. Riedmiller. Reinforcement Learning for DEC-MDPs with Changing Action Sets and Partially Ordered Dependencies. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 1333–1336, Estoril, Portugal, 2008. IFAAMAS.
13. G. J. Gordon. Stable Function Approximation in Dynamic Programming. In *Proc. of the Twelfth International Conference on Machine Learning*, pages 261–268, Tahoe City, USA, 1995. Morgan Kaufmann.
14. G.J. Gordon. Stable function approximation in dynamic programming. Technical report, CMU-CS-95-103, CMU School of Computer Science, Pittsburgh, PA, 1995.
15. G.J. Gordon. Chattering in SARSA (λ). Technical report, 1996.
16. Arthur Guez, Robert D. Vincent, Massimo Avoli, and Joelle Pineau. Adaptive treatment of epilepsy via batch-mode reinforcement learning. In *AAAI*, pages 1671–1678, 2008.
17. R. Hafner and M. Riedmiller. Reinforcement Learning in Feedback Control. 2011. submitted for publication.
18. G.E. Hinton and R.R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.
19. S. Kalyanakrishnan and P. Stone. Batch reinforcement learning in a complex domain. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 650–657, New York, NY, USA, May 2007. ACM.
20. T. Kietzmann and M. Riedmiller. The Neuro Slot Car Racer: Reinforcement Learning in a Real World Setting. In *Proceedings of the Int. Conference on Machine Learning Applications (ICMLA09)*, Miami, Florida, Dec 2009. Springer.
21. M. Lagoudakis and R. Parr. Model-Free Least-Squares Policy Iteration. In *Proceedings of Neural Information Processing Systems (NIPS2001)*, pages 1547–1554, Vancouver, Canada, 2001.
22. M. Lagoudakis and R. Parr. Least-Squares Policy Iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.
23. S. Lange. Tiefes Reinforcement Lernen auf Basis visueller Wahrnehmungen. Dissertation, Universität Osnabrück, 2010.
24. S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain, 2010.
25. S. Lange and M. Riedmiller. Deep learning of visual control policies. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2010)*, Brugge, Belgium, 2010.
26. M. Lauer and M. Riedmiller. An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, pages 535–542, Stanford, USA, 2000. Morgan Kaufmann.
27. L. Lin. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8(3):293–321, 1992.
28. D. Ormoneit and P. Glynn. Kernel-based reinforcement learning in average-cost problems: An application to optimal portfolio choice. *Advances in neural information processing systems*, pages 1068–1074, 2001.
29. D. Ormoneit and P. Glynn. Kernel-based reinforcement learning in average-cost problems. *IEEE Transactions on Automatic Control*, 47(10):1624–1636, 2002.
30. D. Ormoneit and Ś. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49(2):161–178, 2002.
31. M. Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning*, pages 317–328, Porto, Portugal, 2005. Springer.

32. M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In H. Ruspini, editor, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, pages 586 – 591, San Francisco, 1993.
33. M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement Learning for Robot Soccer. *Autonomous Robots*, 27(1):55–74, 2009.
34. M. Riedmiller, R. Hafner, S. Lange, and M. Lauer. Learning to dribble on a real robot by success and failure. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 2207–2208, 2008.
35. M. Riedmiller, M. Montemerlo, and H. Dahlkamp. Learning to Drive in 20 Minutes. In *Proceedings of the FBIT 2007 conference.*, Jeju, Korea, 2007. Springer. Best Paper Award.
36. R. Schoknecht and A. Merke. Convergent combinations of reinforcement learning with linear function approximation. *Advances in Neural Information Processing Systems*, pages 1611–1618, 2003.
37. S. Singh, T. Jaakkola, and M.I. Jordan. Reinforcement learning with soft state aggregation. *Advances in neural information processing systems*, pages 361–368, 1995.
38. R. Sutton and A. Barto. *Reinforcement Learning. An Introduction*. MIT Press/A Bradford Book, Cambridge, USA, 1998.
39. S. Timmer and M. Riedmiller. Fitted Q Iteration with CMACs. In *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 07)*, Honolulu, USA, 2007.
40. S. Tognetti, S.M. Savaresi, C. Spelta, and M. Restelli. Batch reinforcement learning for semi-active suspension control. pages 582 –587, jul. 2009.