Freie Universität Berlin
Fachbereich Mathematik und Informatik

Institut für Mathematik

Masterarbeit

Model-Based Option Selection for Efficient Transfer of Expert Knowledge in Reinforcement Learning

Florian Dorner

Matrikelnummer:	4855587
Betreuung:	PD Dr. Pavel Gurevich
Zweitgutachten:	Prof. Dr. Nicolas Perkowski
Abgabe:	January 16, 2020

Freie Universität Berlin

Selbstständigkeitserklärung

Name:	(Nur Block- oder Maschinenschrift verwenden.)
Vorname:	
geb.am:	
Matr.Nr.:	

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende ______________________________ selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: ___________________ Unterschrift: _____________________________

Contents

Abstract

Providing reinforcement learning (RL) agents with useful inductive biases, whether they are obtained from other RL agent's experiences in similar domains or from human experts, can be a challenging task. In this work, we present an approach to partial solve this problem in domains that allow for a natural decomposition of the state space with a bottleneck. This is done by fixing a set of policies on one part S_{fix} of the state space and learning a conditional model of these policy's consequences. After having been learned once, these models can then be used to train an agent on the other part of the state space S_{learn} to act optimally under the implicit assumption that it will always choose the best conditional policy when entering S_{fix}. Since the learned models are independent of S_{learn}, this allows for a spatial modularization of RL: A variety of different environments can contain states with the same dynamics and rewards as S_{fix} and the learned models can be reused for all of them. Access to the modelled consequences of the policies allows an agent to be trained without having to actually enter S_{fix}. This can be especially useful if operating in S_{fix} is expensive or risky. For example, S_{fix} could correspond to the agent driving a car on a busy street. The developed ideas can be framed as a special case of the options framework with a focus on spatial restrictions on options that allow for smaller representations of the corresponding models as well as the learned Q-function.

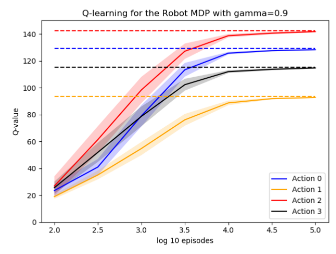
1 Introduction

Humans often seem to learn in a somewhat modular way: after learning how to perform a new skill, we are able to perform it in a variety of circumstance and are able to learn more complicated behavior that builds upon that skill. Oftentimes, we are even able to learn higher level skills without having to actually perform all of the necessary subroutines: If Alice is already quite good at typesetting in LaTeX, she could learn to compose computer science papers while letting Bob do all the typesetting and still be reasonably confident, that she would be able to do the combined process of composing and typesetting herself for the next paper.

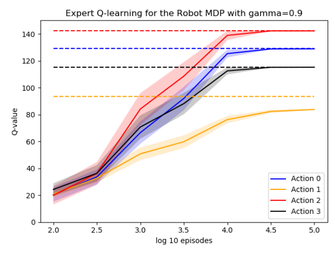
Most reinforcement learning systems, on the other hand, learn in a very monolithic way: even if a task can easily be broken down into subtasks by a human, it is not obvious how to efficiently incorporate this information into the reinforcement learning framework. As long as the subtasks are completely independent such that the optimal strategy for subtask A does not depend on subtask B, it is possible to first learn the optimal policy for task A and then fix a reinforcement learning agent's behavior to the optimal policy for task A while letting it learn the optimal policy on B. However, this strategy has two disadvantages: at first, this requires simulating or actually performing task A a lot of times, even after having learned how to do it. If A is a lengthy task, that might waste a lot of resources on simulation. And it gets even worse if learning is to be performed in the real world instead of a simulation and performing A wastes resources directly (like moving a robotic arm from A to B), comes with some unavoidable risk (like walking next to a cliff), or even both (like driving a car). The second problem is that the optimality of behavior can be highly context dependent: if there are two ways to achieve task A, that deplete different types of resources, the optimal choice depends on the future availability and relevance of those resources. And that might depend on how well the agent has learned to acquire the different resources and what other tasks she is planning to do. So apart from mastering both ways to achieve the task, it is also necessary to decide which of them to use. Just fixing a policy for A clearly does not achieve that!

The idea explored in this thesis is to compress learned subtasks, such that an agent can learn as if they were performed, without having to actually perform them. In the simplest case, this can be imagined as an expert telling the agent how good the consequences of performing a subtask in a predefined way would be for her in expectation. This solves the first problem by allowing the agent to learn how to perform surrounding tasks without having to repeat the already learned subtask. The second problem can be alleviated in a similar way: this time, there are multiple experts and each knows a different way of solving the subtask. Again, they tell the agent how good the consequences of their course of action would be (taking into account the agent's general skill level) and the agent is assumed to follow the advice of the expert who predicts the best consequences. For example, one expert might tell an agent that it could cross a river by walking to a far away bridge and that this takes half an hour, while the other one suggests taking a nearer, but shaky bridge. If the agent cannot swim, taking the shaky bridge carries the risk of falling into the water and getting carried away by the current, such that the expected time for crossing is over two hours. On the other hand, an agent that is able to swim against the current only loses a few minutes by falling into the water, making the shaky bridge the superior option. Thus an agent that knows how to swim will follow the second expert's advice, while an agent who doesn't will cross the further bridge. This way, the agent can learn to perform subtasks contextually and as will be shown later, this leads to globally optimal behavior as long the way of performing the subtask needed for that is known to one of the experts; even if none of the experts knows that her specific way of doing the task is part of the global optimum.

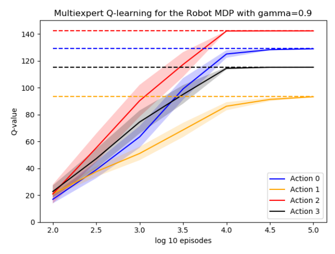
On a more formal level, we notice that for a policy fixed on some part of the state space, the values of states in this part are a linear function of the values of states outside. Even better, they only depend on the values of those states outside which are directly reachable from the inside. This means that this linear function is the same for subsets of the state space with the same dynamics ("an expert's domain of expertise") for different MDPs. Once we have learned this linear function, we are able to reuse it for a lot of MDPs that contain the same subproblems: Instead of trying to learn on the subproblem, we treat transitions to it as terminal and give a reward that is equal to the linear function applied to our current estimates of values for states outside the subproblem. This way we can get to correct value estimates for our higher level task, either for a fixed or the optimal policy (given the fixed policy for the subproblem), without having to simulate the subproblem ever again. We can also learn the linear function for multiple policies for the same subtask and give the maximum of the outputs of those functions as a reward, instead. This corresponds to taking into account the suggestions of multiple experts and choosing the contextually best one: which of the linear functions is maximal depends on the values of the outside states, which are subject to change with learning. In this case, we are even able to recover the globally optimal policy, as long as its restriction to the subproblem was one of our policies for the subproblem, even if we have no idea which of them. As it turns out, the learned linear function is a special case of an option model as defined in [12] and can be learned by a modified version of SARSA, while the resulting learning algorithm is an amalgam of a single-step and a two/multi-step Bellman update of an action-and-option value function.



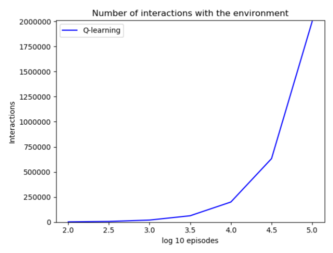
(a) Q-values (Q-learning)



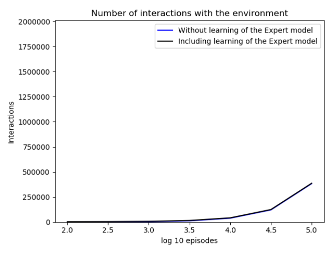
(b) Q-values (one expert)



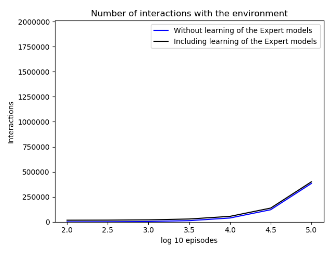
(c) Q-values (multiexpert)



(d) Interactions (Q-learning)



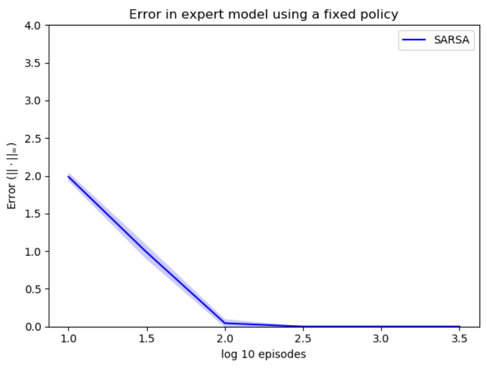
(e) Interactions (one expert)



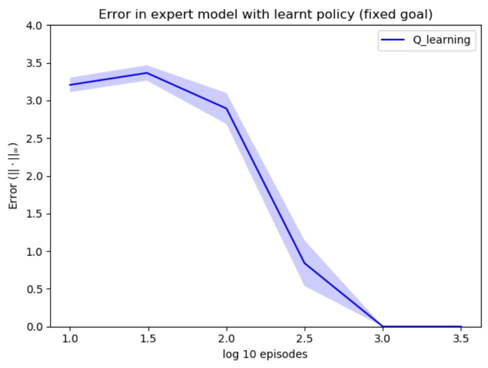
(f) Interactions (multiexpert)

Figure 1: Q-values for the initial state using no, one and multiple experts. The dashed lines represent the correct Q^*-values. a) Q-learning seems to converge to the correct value for all actions b) Expert-Q-learning converges for every action but one. c) Multi-Expert-Q-learning converges for all actions d) Q-learning needs over 2 million samples from the environment to converge. e) Expert-Q-learning needs less than five hundred thousand interactions to converge for three of the actions. Learning the model of the expert policy's consequences requires comparably few actions. f) For Multi-Expert-Q-learning, learning the models takes slightly longer, but this still only slightly increases the total amount of interactions.

Figure 1 shows the advantages of the approach: In a simple MDP where a robot has to keep his fuel in balance while conducting different tasks (see section 5), outsourcing the refueling to a locally optimal expert (and using a prelearned model for the consequences of the expert's behavior) reduces the number of necessary interactions with the environment considerably while learning the correct Q-values for most actions in the initial state. Adding different expert policies leads to faster convergence and only needs slightly more interactions. This shows that with the correct decomposition, the approach can even be competitive when the models for the expert policies' effects needs to be learned from scratch. As the models describe expected returns given a combination of the fixed policy on S_{fix} and learned behavior on S_{learn}, they can be derived from looking at Q-values on S_{fix} for different Q-values on S_{learn}. This allows us to also learn the models, while learning how to carry out an expert policy that is implicitly specified by rewards for reaching certain states from S_{fix}:



(a) Model error (fixed policy)



(b) Model error (fixed goal)

Figure 2: Model errors for Q-learning conditioned on a goal and SARSA used to learn the model of expert behavior's consequences. a) The SARSA-based approach with the expert policy converges to the correct model very quickly. b) The approach based on Q-learning with a reward for reaching the same end state as the expert policy takes longer to converge, but the amount of necessary episodes is still negligible compared with learning a solution for the whole MDP.

Figure 2 shows that learning an implicitly specified expert policy plus model for the policy's consequences can be viable as well. A modified Q-learning agent that receives a reward at the expert's usual end state is able to learn the correct model. While this approach obviously needs more interactions with the environment than the approach for which the expert policy is already known, the overhead is still small compared to the total learning time of the Q-learning agent in figure 1, as the correct model is learned after around 1000 episodes.

After reviewing some preliminaries on conditional expectations, Markov Decision Processes and Dynamic Programming in section 2, we will discuss how to use spatial decomposition combined with a set of fixed policies on one part of the state space S_{fix} to speed up dynamic programming methods in section 3. This will lead to the first main result, theorem 3.2.1 : an adapted dynamic programming method converges to the Q-values for the optimal policy on the rest of the state space, given a fixed policy on S_{fix}, without having to keep Q-values for states in S_{fix}. The second main result, theorem 3.3.1 generalizes this to multiple fixed policies, of which the best one given the current Q-values is implicitly chosen by the algorithm. In section 4, we review a general result by Szepesvári [3] about approximating dynamic programming with samples from an MDP. This result is then used to show that the sampling-based algorithms derived from the adapted dynamic programming approach, Expert-Q-learning (theorem 4.4.1) and Multi-Expert-Q-learning (theorem 4.5.1) converge without needing samples from S_{fix}, as long as a correct model for the consequences of the fixed policies is available. Section 5 contains experiments, section 6 a survey of related previous work and the thesis concludes with a brief discussion of promising avenues for future work in section 7.

2 Preliminaries

2.1 Conditional expectations

This brief reminder of the definition of conditional expectations as well as their elementary properties follows [8], but most graduate level textbooks on probability theory should include the same results:

Definition 2.1.1. *Let (Ω, Σ, P) be a probability space and $H \subset \Sigma$ be a sub sigma-algebra. For a real random variable $Z(\omega)$ with $\mathbb{E}[|Z|] < \infty$, a conditional expectation of Z given H, $\mathbb{E}[Z|H]$ is a random variable Y on Ω such that Y is H-measurable and for all sets $h \in H$ we have $\mathbb{E}[Y\chi_h] = \mathbb{E}[Z\chi_h]$ with χ_A denoting the characteristic function of a set A.*

The notation $\mathbb{E}[Z|H]$ is admissible, since all such random variables are pairwise equal with probability one. For another random variable Y and the respectivly generated $\sigma-$algebra $\sigma(Y)$, we use the abbreviation $\mathbb{E}[Z|Y] := \mathbb{E}[Z|\sigma(Y)]$

Theorem 2.1.2. *For random variables Y, Z with $\mathbb{E}[|Z|] < \infty$ and $\mathbb{E}[|Y|] < \infty$ on Ω and sub-σ-algebras $G \subset H \subset \Sigma$, we have that*

1) $\mathbb{E}[\mathbb{E}[Z|H]] = \mathbb{E}[Z]$

2) *If $Z = c \in \mathbb{R}$ for some constant c with probability one, $\mathbb{E}[Z|H] = c$.*

3) *For $a, b \in \mathbb{R}$ we have that $\mathbb{E}[aY + bZ|H] = a\mathbb{E}[Y|H] + b\mathbb{E}[Z|H]$ with probability one.*

4) *If $Z \leq Y$ with probability one, we have $\mathbb{E}[Z|H] \leq \mathbb{E}[Y|H]$ with probability one.*

5) $\mathbb{E}[\mathbb{E}[Z|H]|G] = \mathbb{E}[Z|G]$ *with probability one*

6) *If $\mathbb{E}[|YZ|] < \infty$ and Y is H-measurable we have with probability one $\mathbb{E}[YZ|H] = Y\mathbb{E}[Z|H]$. In particular we have that: $\mathbb{E}[Y|H] = Y$ for H-measurable and integratable Y.*

7) *If $P(y|x)$ is the conditional measure of a random variable $y \in \mathbb{R}$ given a random variable $x \in \mathbb{R}^n$ in the sense that*

$$\int_G P(y \in H|x) \; dP(x) = P(y \in H \cap x \in G)$$

for all Borel sets $G \subset \mathbb{R}^n, H \subset \mathbb{R}$, we have that

$$\int_{-\infty}^{\infty} y \; dP(y|x) = \mathbb{E}[y|x]$$

with probability one.

In particular, 7) works whenever y is sampled from $P(y|x)$ after x was sampled from some other distribution.

2.2 Markov Decision Processes

Markov Decision Processes (and their generalizations) are a very common formalization of decision making in complex environments. The basic premise is that actions affect the environment's next state in a way that depends on the current state. Some actions correspond to good decisions and are rewarded, while bad actions lead to punishment. Rewards and punishment are often purely consequentialist (depend only on the next state), but they are also allowed to depend on the state-action combination, independently of the next state. This allows for modeling deontological decision contexts, as well. For example, an agent could be punished for lying under most circumstances, independent of the actual consequences of that lie. The following exposition is mostly based on [1].

Definition 2.2.1. *A Markov Decision Process (MDP) is the tuple $((S, \Sigma, \mu), A, p, r, p_0)$ for a measure space (S, Σ, μ), an arbitrary set A, $p : S \times S \times A \to R$, such that $p(\cdot, s, a)$ is a probability density on S for all $s \in S, a \in A$, $r : S \times A \times S \to \mathbb{R}$, as well as a probability density on S, p_0. An MDP is called finite, if S and A are finite sets.*

We will only consider finite MDPS and abbreviate them as a tuple (S, A, p, r, p_0) implicitly assuming the counting measure on S.

S is called the state space and represent all possible states of the environment. The action space A represents the possible actions an agent can take in the environment. The transition probabilities $p(s', s, a) = p(s'|s, a)$ define how likely it is for the environment state to transition to s', if the agent takes action a in state s. The reward function $r(s', a, s)$ encodes the reward that the agent receives for a transition to s' after performing action a in state s. p_0 defines the distribution of start states s. Sometimes stochastic rewards appear in the literature as well. In this case, $r(s', a, s)$ would have to be redefined to be a distribution instead of a real number. However, this complicates some arguments as well as notation without bringing too much of a benefit: In the MDP-framework we assume that all relevant information about the environment is encoded in the states, such that there does not seem to be any ready interpretation for the source of stochasticity in rewards. Still, the presented results should also hold for stochastic rewards, as long as they are bounded, by replacing $r(s', a, s)$ by the expected reward for that transition.

Definition 2.2.2. *For a given MDP, $M = (S, A, p, r, p_0)$, a policy π is a map $S \to \mathcal{P}(A)$ with $\mathcal{P}(A)$ denoting the set of probability measures on A. A policy is called deterministic, if it maps all states to probability measures supported by a single point (Dirac measures). A policy that is not deterministic is also called stochastic.*

For the finite case, we will slightly overload notation and denote the action-probabilities induced by a policy π for a given s as $\pi(a|s) := \pi(s)(a)$. A deterministic policy can also be defined as a map $S \to A$ and it is usually more convenient to use this interpretation, when the discussion is restricted to deterministic policies.

The policy describes the decision making of the agent: when seeing a state s, an agent using policy π reacts with an action sampled from $\pi(s)$ (or just $\pi(s)$ in the deterministic case). In the finite case and given a policy π an MDP induces a finite Markov Chain on S with transition probabilities

$$P(s'|s) = \sum_{a \in A} \pi(a) p(s'|s,a)$$

and start distribution p_0. Usually, it also makes sense to consider the taken actions explicitly. The MDP combined with the policy induces random variables

$$s_0 \sim p_0, a_0 \sim \pi(s_0), s_1 \sim p(\cdot|s_0,a_0), ..., a_n \sim \pi(s_n), s_{n+1} \sim p(\cdot|s_n,a_n),$$

etc., as well as

$$r_t = r(s_{t+1}, a_t, s_t).$$

The stochastic process (s_t, a_t, r_t) has the Markov Property, since the distributions of all three individual variables only depend on s_{t-1}. We will denote expectations regarding functions f of this stochastic process as

$$\mathbb{E}_\pi[f(s_t, a_t, r_t, ..., s_0, a_0, r_0)]$$

and induced probabilities as P_π. In a slight abuse of notation, we will use

$$\mathbb{E}_\pi[f(s_t, a_t, r_t, ..., s_h, a_h, r_h)|s_h = s]$$

and

$$\mathbb{E}_\pi[f(s_t, a_t, r_t, ..., s_h, a_h, r_h)|s_h = s, a_h = a]$$

as shorthand for expectations of the process initialized at time h with $s_h = s$ or $s_h = s, a_h = a$ and everything sampled as above starting from there, even if $P(s_h = s)$ or $P(s_h = s, a_h = a)$ is equal to zero.

The Markov Property of the agent-environment interaction greatly simplifies a lot of reasoning about the decision making. As mentioned in [1], the restrictions the MDP-framework imposes on modeling real world decisions are most readily interpreted as restrictions on the definition of the state space: if one discovers that the transition probabilities depend on more than just the current state for the initial choice of state definition, it is always possible to define an augmented state space that either consists of the history of states in the original definition or includes other more detailed information, as discussed in [16]. However, that can lead to exponentially bigger state spaces, difficulties with actually observing states and algorithmic difficulties, for which reason generalizations to the MDP framework like Partially Observable Markov Decision Processes (POMDPs) are often a more useful model in those cases.

2.2.1 Value Functions

In the MDP-setting rewards are used to model the desirability of performing certain actions in certain contexts (states). In order to compare policies, we need to evaluate how good a certain policy is and for this we need to somehow accumulate the rewards received following this policy.

Definition 2.2.3. *For a given MDP, $M = (S, A, p, r, p_0)$ and a policy π the return after step t, G_t is defined as $\sum_{k=0}^{\infty} r_{t+k}$, as long the series converges absolutely. For $\gamma \in [0, 1)$ we define the γ-discounted return after step t as $G_t^\gamma = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. Once again, the discounted return is undefined, if the series does not converge absolutely.*

As long as the return is defined, it obviously fits the recursion equation $G_t = r_t + G_{t+1}$, while for the discounted return we have

$$G_t^\gamma = \sum_{k=0}^{\infty} \gamma^k r_{t+k} = r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} = r_t + \gamma G_{t+1}^\gamma.$$

We will make use of similar recursions for related objects throughout the rest of the text.

Now, the expected return at the beginning $\mathbb{E}_\pi[G_0]$ seems like a natural criterion to compare different policies. However, for MDPs with few terminal states (states s, such that $p(s|s, a) = 1$ and $r(s, a, s)$=0 for all $a \in A$), the series might diverge with nonzero probability, leaving the expectation undefined. In order to deal with this, MDPs are usually considered in combination with some discount factor $\gamma \in [0, 1)$. In this case, the expected γ-discounted return is well defined, at least for finite MDPs: Since the r_t are bounded by some constant c as function values of a function with discrete support,

$$|G_0^\gamma| = |\sum_{t=0}^{\infty} \gamma^t r_t| \le \sum_{t=0}^{\infty} |\gamma^t r_t| \le c \sum_{k=0}^{\infty} |\gamma^t| = \frac{c}{1-\gamma}$$

and the expected value exists. The discount factor can be interpreted as describing how much the future matters for decisions taken now, the higher γ, the more important the future. One popular interpretation of γ is to look at $1 - \gamma$ as the probability of the agent-environment interaction abruptly ending at a given step. Then, the likelihood of the interaction not having been interrupted before step t is γ^t and the discounted return is the expected return before the interaction is ended. This is equivalent to the expected return the "same" policy yields for the Modified MDP $M' = (S', A, p', r', p'_0)$ where

$$S' = S \cup \{T\} \cup S \times \{0\},$$

,

$$p'(T|T, a) = 1$$

for all $a \in A$,

$$p'(T|s,a) = 1 - \gamma$$

for all $s \in S, a \in A$,

$$r'(T,a,s) = 0$$

for all $a \in A, s \in S'$,

$$p'(s'|s,a) = \gamma p(s'|s,a)$$

for all $s, s' \in S, a \in A$,

$$r'(s',a,s) = r(s',a,s),\ p'(s'|(s,0),a) = p(s'|s,a)$$

and

$$r'(s',a,(s,0)) = r(s',a,s)$$

for all $s, s' \in S$ and $a \in A$, with $r'(s',a,s)$ arbitrary and $p(s'|s,a) = 0$ where undefined by the listed equations, as well as

$$p_0'((s,0)) = p_0(s)$$

for all $s \in S$ and zero else. In words, we added a terminal state such that every state in the base MDP has a probability of $1-\gamma$ to transition to it without reward, as well as copies of the base states that serve as initial states and behave as the base states, but are protected from transitions to the terminal state and can not be transitioned to. Now, if we expand the definition of π by setting arbitrary values for $\pi(a|T)$ and $\pi(a|(s,0)) = \pi(a|s)$, we get that the expected return for the modified MDP M' is equal to the discounted return for M:

$$\mathbb{E}_{\pi,M}[G_0^\gamma] = \mathbb{E}_{\pi,M'}[G_0].$$

Another interpretation comes from economic contexts where the reward models monetary returns. There, the discount factor γ is related to the interest rate, since positive interest can make money received now more valuable than money received later.

The return after t steps G_t gives us some measure of how well a policy is doing in the future, but without incorporating any information about what happened before step t it can have very high variance, making it hard to estimate. In order to alleviate this problem, we will condition it on the state s_t before taking expectations.

Definition 2.2.4. *For a policy π and discount rate γ we define the Value function $V_\pi^\gamma : S \to \mathbb{R}$ as*

$$V_\pi^\gamma(s) = \mathbb{E}_\pi[G_0^\gamma | s_0{=}s].$$

In the following, we will assume γ to be fixed and talk about

$$V_\pi := V_\pi^\gamma.$$

The Markov property gives us $V_\pi(s) = \mathbb{E}_\pi[G_t^\gamma | s_t{=}s]$, so we can interpret it as the expected, discounted future return after landing in s at any point in time. The independence of time granted by the Markov Property makes the value function a particullarly powerful tool: Since $G_t^\gamma = r_t + \gamma G_{t+1}^\gamma$:

$$V_\pi(s) = \mathbb{E}_\pi[G_0^\gamma | s_0{=}s] = \mathbb{E}_\pi[r_0 + \gamma G_1^\gamma | s_0{=}s] = \mathbb{E}_\pi[r_0 | s_0{=}s] + \gamma \mathbb{E}_\pi[G_1^\gamma | s_0{=}s].$$

By expanding the second expectation using the law of total expectation and using the convention that zero times an undefined term is equal to 0, we obtain

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[r_0 | s_0{=}s] + \gamma \sum_{s' \in S} P_\pi(s_1{=}s' | s_0{=}s) \mathbb{E}_\pi[G_1^\gamma | s_1{=}s', s_0{=}s] \\
&= \mathbb{E}_\pi[r_0 | s_0{=}s] + \gamma \sum_{s' \in S} \sum_{a \in A} p(s'|s,a) \pi(a|s) \mathbb{E}_\pi[G_1^\gamma | s_1{=}s'] \\
&= \mathbb{E}_\pi[r_0 | s_0{=}s] + \gamma \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a) \mathbb{E}_\pi[G_1^\gamma | s{=}s'] \\
&= \mathbb{E}_{\substack{a \sim \pi(a|s) \\ s' \sim p(s'|s,a)}} [r(s', a, s) + \gamma V_\pi(s')]
\end{aligned}$$

since because of the Markov property $\mathbb{E}_\pi[G_1^\gamma | s_1{=}s'] = \mathbb{E}_\pi[G_0^\gamma | s_0{=}s']$. Expanding the expectation again, we get

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s', a, s) + \gamma V_\pi(s')). \tag{2.2.1}$$

This equation is called the Bellman equation for V_π. It plays a central role in the theory of reinforcement learning and will allow us to iteratively calculate the value function, later on.

If we want to compare policies, we could now look at the expected returns at the beginning $\mathbb{E}_{s_0 \sim p_0} V_\pi(s_0)$. A more fine grained way that turns out to be more useful is to look at the values of all states instead:

Definition 2.2.5. *A policy π for a given MDP and discount rate is called better or equal to another policy π', if for all $s \in S$: $V_\pi(s) \geq V_{\pi'}(s)$ and better, if there also exists at least one s such that the inequality is strict.*

The defined relations are obviously transitive and "better or equal to" is antisymmetric and reflexive, making it a partial order.

While the V-function is a useful tool for comparing policies, it often encodes insufficient information: If a policy takes both actions a and b in half the cases in some state and has a high

V-value for that state, this could be because both actions are quite good, but also because one of them is terrific and the other one moderately terrible. Having this information and knowing which of the actions are actually good can be quite helpful when trying to modify a policy into a better one. The Q-Function solves this by encoding the expected future discounted return given a policy not for a state, but for a state action pair:

Definition 2.2.6. *For a policy π and discount rate γ we define the Action-Value function $Q^\gamma_\pi : S \times A \to \mathbb{R}$ as $Q^\gamma_\pi(s,a) := \mathbb{E}_\pi[G^\gamma_0|s_0{=}s, a_0 = a]$.*

In the following, we will assume γ to be fixed and talk about

$$Q_\pi := Q^\gamma_\pi.$$

Again, the Markov property gives us $Q_\pi(s,a) = \mathbb{E}_\pi[G^\gamma_t|s_t{=}s, a_t = a]$, so we can interpret it as the expected, discounted future return after taking action a in s at any point in time. Now, we get that

$$\begin{aligned}
&Q_\pi(s,a)\\
&= \mathbb{E}_\pi[G^\gamma_0|s_0{=}s, a_0 = a]\\
&= \mathbb{E}_\pi[r_0 + \gamma G^\gamma_1|s_0{=}s, a_0 = a]\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \mathbb{E}_\pi[G^\gamma_1|s_0{=}s, a_0 = a]\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a)\mathbb{E}_\pi[G^\gamma_1|s_0{=}s, a_0 = a, s_1 = s']\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a)\mathbb{E}_\pi[G^\gamma_1|s_1 = s']\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a)\mathbb{E}_\pi[G^\gamma_0|s_0 = s']\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a) \sum_{a'\in A} \pi(a'|s')\mathbb{E}_\pi[G^\gamma_0|s_0 = s', a_0 = a']\\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a) \sum_{a'\in A} \pi(a'|s')Q_\pi(s',a')\\
&= \mathbb{E}_{\substack{s'\sim p(s'|s,a)\\ a'\sim\pi(a'|s')}}[r(s',a,s) + \gamma Q_\pi(s',a')],
\end{aligned}$$

where we used that $G^\gamma_t = r_t + \gamma G^\gamma_{t+1}$, the Markov property of the process, as well as the law of total expectation. Put differently, we have

$$Q_\pi(s,a) = \sum_{s'\in S} p(s'|s,a)(r(s',a,s) + \gamma \sum_{a'\in A} \pi(a'|s')Q_\pi(s',a')) \tag{2.2.2}$$

Again, this Bellman equation plays an important theoretical role and will allow us to iteratively approximate Q_π later on. Using that

$$\begin{aligned}
&V_\pi(s)\\
&= \mathbb{E}_\pi[r_0|s_0{=}s] + \gamma \sum_{a\in A} \pi(a|s) \sum_{s'\in S} p(s'|s,a)\mathbb{E}[G_1^\gamma|s_1{=}s']\\
&= \sum_{a\in A} \pi(a|s)\mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{a\in A} \pi(a|s) \sum_{s'\in S} p(s'|s,a)\mathbb{E}[G_1^\gamma|s_1{=}s']\\
&= \sum_{a\in A} \pi(a|s)\mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \sum_{a\in A} \pi(a|s)\gamma \sum_{s'\in S} p(s'|s,a)\mathbb{E}[G_1^\gamma|s_1{=}s']\\
&= \sum_{a\in A} \pi(a|s)(\mathbb{E}_\pi[r_0|s_0{=}s, a_0 = a] + \gamma \sum_{s'\in S} p(s'|s,a)\mathbb{E}[G_1^\gamma|s_1{=}s'])\\
&= \sum_{a\in A} \pi(a|s)Q_\pi(s,a)\\
&= \mathbb{E}_{a\sim\pi(a|s)}[Q_\pi(s,a)],
\end{aligned}$$

it is also possible to obtain the mixed versions of the Bellman equation expressing Q_π in terms of V_π and vice versa:

$$Q_\pi(s,a) = \mathbb{E}_{\substack{s'\sim p(s'|s,a)\\ a'\sim\pi(a'|s')}}[r(s',a,s) + \gamma Q_\pi(s',a')] = \mathbb{E}_{s'\sim p(s'|s,a)}[r(s',a,s) + \gamma V_\pi(s')],$$

while

$$V_\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q_\pi(s,a)] = \mathbb{E}_{\substack{a\sim\pi(a|s)\\ s'\sim p(s'|s,a)\\ a'\sim\pi(a'|s')}}[r(s',a,s) + \gamma Q_\pi(s',a')].$$

2.2.2 Optimality

Definition 2.2.7. *For a given MDP and a fixed discount rate γ, a policy π is called optimal if it is better or equal to every other policy π'.*

In other words for all policies π' and $s \in S$ we have that $V_\pi(s) \geq V_{\pi'}(s)$. Oftentimes, finding an optimal policy is referred to as "solving" the MDP. It is fairly obvious that there can be multiple optimal policies for some MDPs. For example, consider an MDP with transition-independent rewards, or a determinstic navigation task (where the only nonzero reward is received upon reaching a terminal goal state) with multiple shortest paths towards the goal. In order to see that there exists a deterministic optimal policy for every MDP, we will outline a way of iteratively constructing it for a given MDP. This is done with help of the policy improvement theorem as proven in [1]:

Theorem 2.2.8. *For a given MDP M and a fixed discount rate γ as well as policy π and a determinstic policy π'*

$$Q_\pi(s,\pi'(s)) \geq V_\pi(s)$$

for all $s \in S$ implies

$$V_{\pi'}(s) \geq V_\pi(s)$$

for all $s \in S$, with strict inequalities for particular states implying strict inequalities for the same states.

Proof.

$$\begin{aligned}
&V_\pi(s) \\
&\leq Q_\pi(s, \pi'(s)) \\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = \pi'(s)] + \gamma \sum_{s' \in S} p(s'|s, \pi'(s)) \sum_{a' \in A} \pi(a'|s') Q_\pi(s', a') \\
&= \mathbb{E}_\pi[r_0|s_0{=}s, a_0 = \pi'(s)] + \gamma \sum_{s' \in S} p(s'|s, \pi'(s)) V_\pi(s') \\
&= \mathbb{E}_{s' \sim p(s'|s, \pi'(s))}[r(s', \pi'(s), s) + \gamma V_\pi(s')] \\
&= \mathbb{E}_{\pi'}[r_0|s_0{=}s] + \mathbb{E}_{\pi'}[\gamma V_\pi(s_1)|s_0{=}s] \\
&\leq \mathbb{E}_{\pi'}[r_0|s_0{=}s] + \mathbb{E}_{\pi'}[\gamma Q_\pi(s_1, \pi'(s_1))|s_0{=}s] \\
&= \mathbb{E}_{\pi'}[r_0|s_0{=}s] + \mathbb{E}_{\pi'}[\gamma \mathbb{E}_{s' \sim p(s'|s_1, \pi'(s_1))}[r(s', \pi'(s_1), s_1) + \gamma V_\pi(s')]|s_0{=}s] \\
&= \mathbb{E}_{\pi'}[r_0|s_0{=}s] + \mathbb{E}_{\pi'}[\gamma r_1|s_0{=}s] + \mathbb{E}_{\pi'}[\gamma^2 V_\pi(s_2)|s_0{=}s] \\
&= \mathbb{E}_{\pi'}[r_0 + \gamma r_1|s_0{=}s] + \gamma^2 \mathbb{E}_{\pi'}[V_\pi(s_2)|s_0{=}s] \\
&\leq .... \\
&\leq \mathbb{E}_{\pi'}[\sum_{t=0}^{n} \gamma^t r_t|s_0{=}s] + \gamma^n \mathbb{E}_{\pi'}[V_\pi(s_n)|s_0{=}s]
\end{aligned}$$

for every $n \in \mathbb{N}$. Because V_π is bounded, for a given ε, there exists an $N \in \mathbb{N}$ such that for all $n > N$, the last term is at most ε. This means that for all such n we have

$$V_\pi(s) \leq \mathbb{E}_{\pi'}[\sum_{t=0}^{n} \gamma^t r_t|s_0{=}s] + \varepsilon$$

and thus

$$V_\pi(s) \leq \mathbb{E}_{\pi'}[\sum_{t=0}^{\infty} \gamma^t r_t|s_0{=}s] + \varepsilon.$$

Because $\varepsilon > 0$ was arbitrary, this implies

$$\begin{aligned}
V_\pi(s) &\leq \mathbb{E}_{\pi'}[\sum_{t=0}^{\infty} \gamma^t r_t|s_0{=}s] \\
&= \mathbb{E}_{\pi'}[G_0^\gamma|s_0{=}s] \\
&= V_{\pi'}(s).
\end{aligned}$$

$\square$

This means that for any given policy π we can get a deterministic policy π' that is equal or better by acting greedily, i.e. setting

$$\pi'(s) = \operatorname{argmax}_{a \in A} Q_\pi(s,a) \geq \mathbb{E}_{a \sim \pi(a|s)}[Q_\pi(s,a)] = V_\pi(s),$$

with conflicts in the argmax resolved arbitrarily. The process of iteratively calculating Q_π and replacing π by the resulting greedy policy is often referred to as policy improvement.

Now assume we have found an optimal policy π^*. Then policy improvement cannot change the policy anymore and in particular it follows from

$$V_{\pi^*}(s) = \mathbb{E}_{a \sim \pi^*(a|s)}[Q_{\pi^*}(s,a)] \leq \max_{a \in A} Q_{\pi^*}(s,a)$$

that

$$V_{\pi^*}(s) = \max_{a \in A} Q_{\pi^*}(s,a),$$

because a strict inequality would give us an improvement by replacing $\pi^*(s)$ by $\operatorname{argmax}_{a \in A} Q_{\pi^*}(s,a)$. Thus, the Bellman optimality equation for $V^* := V_{\pi^*}$ becomes

$$\begin{aligned}
&V^*(s) = V_{\pi^*}(s) \\
&= \max_{a \in A} Q_{\pi^*}(s,a) \\
&= \max_{a \in A} \mathbb{E}_{\pi^*}[r_0 + \gamma G_1^\gamma | s_0{=}s, a_0 = a] \\
&= \max_{a \in A} \mathbb{E}_{s' \sim p(s'|s,a)}[r_0 + \gamma V^*(s') | s_0{=}s, a_0 = a].
\end{aligned}$$

By expanding the expectation, we get

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V^*(s')). \tag{2.2.3}$$

Similarly for $Q^* := Q_{\pi^*}$ we get the Bellman optimality equation

$$\begin{aligned}
&Q^*(s,a) = Q_{\pi^*}(s,a) \\
&= \mathbb{E}_{\pi^*}[r_0 | s_0{=}s, a_0 = a] + \gamma \sum_{s' \in S} p(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q_{\pi^*}(s',a') \\
&= \mathbb{E}_{\pi^*}[r_0 | s_0{=}s, a_0 = a] + \gamma \sum_{s' \in S} p(s'|s,a) V_{\pi^*}(s') \\
&= \mathbb{E}_{\pi^*}[r_0 | s_0{=}s, a_0 = a] + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a' \in A} Q_{\pi^*}(s',a') \\
&= \mathbb{E}_{s' \sim p(s'|s,a)}[r(s',a,s) + \gamma \max_{a' \in A} Q^*(s',a')],
\end{aligned}$$

or put differently:

$$Q^*(s,a) = \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \tag{2.2.4}$$

Next, we will use the Bellman optimality equation for Q^* to prove the well known fact that there exists a deterministic optimal policy that can be found by policy improvement.

Theorem 2.2.9. *For every MDP M, there exists a deterministic optimal policy.*

Proof. We set $\pi_{n+1}(s) = \text{argmax}_{a \in A} Q_{\pi_n}(s,a)$ for all $s \in S$. Then π_{n+1} is better or equal to π_n and since there are only finitely many policies and the better/equal relation is clearly transitive, we have $\pi_k = \pi_{k+1}$ for some $k \in N$ (and the same for all larger numbers). Now,

$$Q_{\pi_{k+1}}(s, \pi_{k+1}(s)) = Q_{\pi_k}(s, \pi_{k+1}(s)) = \max_{a \in A} Q_{\pi_k}(s,a) = \max_{a \in A} Q_{\pi_{k+1}}(s,a),$$

such that

$$\begin{aligned}
&Q_{\pi_{k+1}}(s,a) \\
&= \mathbb{E}_{\substack{s' \sim p(s'|s,a) \\ a' \sim \pi_{k+1}(a'|s')}} [r(s',a,s) + \gamma Q_{\pi_{k+1}}(s',a')] \\
&= \mathbb{E}_{s' \sim p(s'|s,a)}[r(s',a,s) + \gamma Q_{\pi_{k+1}}(s', \pi'(s_{k+1}))] \\
&= \mathbb{E}_{s' \sim p(s'|s,a)}[r(s',a,s) + \gamma \max_{a' \in A} Q_{\pi_{k+1}}(s',a')].
\end{aligned}$$

This means that $Q_{\pi_{k+1}}$ fulfills the Bellman optimality equation and is thus uniquely determined, which we will prove in theorem 2.3.5 in the next chapter. In particular the Q-values of the final policy $\pi = \pi_{k+1}$ and thus the π (which is greedy with respect to its on Q-values) itself are independent of the starting policy π_0. By construction, π is better or equal to all start policies π_0 and optimal since π_0 was arbitrary. $\square$

2.3 Dynamic Programming

We will now introduce a method of iteratively calculating the value function by a fixed point iteration that usually goes under the name of Dynamic Programming. For a more extensive but mathematically less rigorouous discussion of dynamic programming, see [1]. First, we recall Banach's Fixed Point Theorem:

Theorem 2.3.1. *Let V be a Banach Space (a complete, normed space) and $T : V \to V$ a strict contraction, i.e. $\|Tv - Tw\| \leq \gamma \|v - w\|$ for a $\gamma < 1$ and all $v, w \in V$. Then there exists a unique fixed point v^* of T with $Tv^* = v^*$ and the recursive sequence defined by $v_{n+1} = Tv_n$ converges to V^* for all start values $v_0 \in V$. Furthermore, we have*

$$\|v_n - v*\| \leq \frac{\gamma^n}{1-\gamma} \|v_1 - v_0\|$$

and in particular for any $v \in V$:

$$\|v - v*\| \leq \frac{1}{1-\gamma} \|Tv - v\|.$$

This is a standard theorem. For a proof see [10].

Definition 2.3.2. *We call an operator T that fulfills the condition from theorem 2.3.1 for a specific $\gamma < 1$ a γ-contraction.*

This can be used to get simple algorithms for approximating V_π, Q_π, V^* and Q^*, as described in [1]:

Theorem 2.3.3. *For a given MDP, $M = (S, A, p, r, p_0)$ and a policy π the iteration*

$$V_{n+1}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V_n(s')),$$

converges to the fixed point V_π for arbitrary initial values.

Proof. We have the Bellman equation 2.2.1 for V_π

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V_\pi(s')),$$

which can be rewritten as

$$V_\pi = TV_\pi$$

with

$$TV(s) := \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V(s')).$$

Now, after relabeling the states, it is enough to show that the so called Bellman operator T is a contraction with respect to the maximum norm on $\mathbb{R}^S$ in order to use theorem 2.3.1 and get convergence. Let V, W be in $\mathbb{R}^S$:

$$\begin{aligned}
&\|TV - TW\|_\infty \\
&= \max_{s \in S} |\sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V(s')) - \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma W(s'))| \\
&= \max_{s \in S} |\sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)\gamma(V(s') - W(s'))| \\
&\leq \max_{s \in S} \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)\gamma|V(s') - W(s')| \\
&\leq \max_{s \in S} \sum_{a \in A} \pi(a|s) \max_{s' \in S} \gamma|V(s') - W(s')| \\
&= \max_{s \in S} \max_{s' \in S} \gamma|V(s') - W(s')|
\end{aligned}$$

$$\begin{aligned}
&= \max_{s' \in S} \gamma |V(s') - W(s')| \\
&= \gamma \|V - W\|_\infty,
\end{aligned}$$

where we used that $\sum_{s'} p(s'|s,a) = 1$ and $\sum_a \pi(a|s) = 1$ for fixed s, a, respectively s. □

A similar approach works for Q_π and Q^*, again as described in [1]:

Theorem 2.3.4. *For a given MDP, $M = (S, A, p, r, p_0)$ and a policy π the iteration*

$$Q_{n+1}(s,a) = \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \sum_{a' \in A} \pi(a'|s') Q_n(s',a')),$$

converges to the fixed point Q_π for arbitrary initial values.

Proof. By equation 2.2.2, the Bellman operator

$$TQ(s,a) := \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \sum_{a' \in A} \pi(a'|s') Q(s',a'))$$

has Q_π as fixed point and it is enough to show that T contracts $\mathbb{R}^{S \times A}$ to see that the fixed point is unique and the iteration converges. Let Q, W be in $\mathbb{R}^{S \times A}$:

$$\begin{aligned}
&\|TQ - TW\|_\infty \\
&= \max_{s,a \in S,A} |\sum_{s' \in S} p(s'|s,a) \gamma \sum_{a' \in A} \pi(a'|s') Q_n(s',a') - \sum_{s' \in S} p(s'|s,a) \gamma \sum_{a' \in A} \pi(a'|s') W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \sum_{s' \in S} p(s'|s,a) \sum_{a' \in A} \pi(a'|s') |Q_n(s',a') - W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \sum_{s' \in S} p(s'|s,a) \max_{a' \in A} |Q_n(s',a') - W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \max_{s',a' \in S,A} |Q_n(s',a') - W_n(s',a')| \\
&= \gamma \max_{s',a' \in S,A} |Q_n(s',a') - W_n(s',a')| \\
&= \gamma \|Q - W\|_\infty,
\end{aligned}$$

using that $\sum_{s'} p(s'|s,a) = 1$ and $\sum_a \pi(a|s) = 1$ for fixed s, a, respectively s. □

Theorem 2.3.5. *For a given MDP, $M = (S, A, p, r, p_0)$, the iteration*

$$Q_{n+1}(s,a) = \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q_n(s',a')),$$

converges to the fixed point Q^* for arbitrary initial values.

Proof. We use equation 2.2.4 to see that the Bellman optimality operator

$$TQ(s,a) := \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a'))$$

has Q^* as fixed point and it is sufficient to prove that T is a γ-contraction on $\mathbb{R}^{S \times A}$ to see that the fixed point is unique and the iteration converges. Let Q, W be in $\mathbb{R}^{S \times A}$

$$\begin{aligned}
&\|TQ - TW\|_\infty \\
&= \max_{s,a \in S,A} |\sum_{s' \in S} p(s'|s,a)\gamma \max_{a' \in A} Q_n(s',a') - \sum_{s' \in S} p(s'|s,a)\gamma \max_{a' \in A} W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \sum_{s' \in S} p(s'|s,a)| \max_{a' \in A} Q_n(s',a') - \max_{a' \in A} W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \max_{s' \in S} | \max_{a' \in A} Q_n(s',a') - \max_{a' \in A} W_n(s',a')| \\
&\leq \gamma \max_{s,a \in S,A} \max_{s',a' \in S,A} |Q_n(s',a') - W_n(s',a')| \\
&= \gamma \|Q - W\|_\infty
\end{aligned}$$

where we used that $\sum_{s'} p(s'|s,a) = 1$ and that by assuming $\max_{a' \in A} Q_n(s',a') > \max_{a' \in A} W_n(s',a')$ without loss of generality, we get that

$$\begin{aligned}
&| \max_{a' \in A} Q_n(s',a') - \max_{a' \in A} W_n(s',a')| \\
&= \max_{a' \in A} Q_n(s',a') - \max_{a' \in A} W_n(s',a') \\
&\leq \max_{a' \in A} Q_n(s',a') - W_n(s', \operatorname{argmax}_{a' \in A} Q_n(s',a')) \\
&= Q_n(s', \operatorname{argmax}_{a' \in A} Q_n(s',a')) - W_n(s', \operatorname{argmax}_{a' \in A} Q_n(s',a')) \\
&\leq \max_{a' \in A} (Q_n(s',a') - W_n(s',a')) \\
&\leq \max_{a' \in A} |Q_n(s',a') - W_n(s',a')|.
\end{aligned}$$

□

A similar iteration based on the corresponding Bellman equation 2.2.3 called value iteration is possible for V^*. It has the disadvantage that V^* alone does not give us a policy but requires the transition probabilities and rewards. This makes it less useful for learning from simulations, when there is no precise model of the MDP and requires additional effort to calculate the policy, even when the model is known.

3 Decomposing an MDP

In order to use expert knowledge, which we formalize as a policy on part of the state space combined with additional knowledge about that policies' performance given the agent's behavior, we want to derive a way to decompose an MDP and be able to use local prior knowledge to our advantage. We will begin with decomposing policy evaluation into local equations and venture towards optimizing given fixed local policies from that.

3.1 Splitting Policy Evaluation

For a fixed policy π, we can also rewrite the Bellman equation for V_π in matrix form. If π is deterministic, we only need the matrices $P = (p(s'|\pi(s), s))_{s,s' \in S}$ and $R = r(s', \pi(s), s)_{s',s \in S}$ to write

$$\begin{aligned}
V_\pi(s) &= \sum_{s' \in S} p(s'|s, \pi(s))(r(s', \pi(s), s) + \gamma V_\pi(s')) \\
&= \sum_{s' \in S} P_{s,s'}(R_{s',s} + \gamma V_\pi(s')) \\
&= \operatorname{diag}(PR)(s) + \gamma P V_\pi(s)
\end{aligned}$$

where diag is the operator mapping a matrix $M \in \mathbb{R}^{d \times d}$ to its diagonal in $\mathbb{R}^d$. Note that P's rows, not necessarily columns are probability distributions, such that we compute a weighted average over the $V_\pi s'$. This gives us a compact equation for the value function

$$V_\pi = \operatorname{diag}(PR) + \gamma P V_\pi \tag{3.1.1}$$

and the bellman operator

$$TV = \operatorname{diag}(PR) + \gamma PV.$$

It is now even easier to see that the Bellman operator is a contraction: since $p(\cdot|a, s))$ is a probability density for all $a \in A, s \in S$, summing over the (non-negative!) elements of a row of P always yields one. But this means that $\|P\|_\infty = 1$ (since the matrix norm induced by the maximum norm is just the maximal sum over absolute values in a row) and therefore

$$\|TV - TW\|_\infty = \|\operatorname{diag}(PR) + \gamma PV - \operatorname{diag}(PR) - \gamma PW\|_\infty = \gamma \|PV - PW\|_\infty \leq \gamma \|V - W\|_\infty.$$

Nondeterministic policies lead to similar equations:

$$\begin{aligned}
V_\pi(s) &= \sum_{s' \in S} \sum_{a \in A} p(s'|s, a)\pi(s|a)(r(s', a, s) + \gamma V_\pi(s')) \\
&= \sum_{a \in A} \pi(s|a) \sum_{s' \in S} p(s'|s, a)(r(s', a, s) + \gamma V_\pi(s'))
\end{aligned}$$

and

$$TV(s) = \sum_{a \in A} \pi(s|a) \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma V(s')).$$

By redefining $P := (\sum_{a \in A} \pi(a|s)(p(s'|a,s)))_{s,s' \in S}$ and $R := (\sum_{a \in A} \sum_{s' \in S} p(s'|s,a)\pi(a|s)r(s',a,s))_s$,

$$V_\pi = R + \gamma P V_\pi \tag{3.1.2}$$

and

$$TV = R + \gamma P V$$

subsume both cases. Note, that the entries of P still define transition probabilities:

$$P_{s,s'} = \sum_{a \in A} \pi(a|s)p(s'|a,s) =: p_\pi(s'|s),$$

the probability of reaching state s' from state s in one step, following the policy π, while R denotes the expected immediate rewards when taking an action according to π for all states:

$$\begin{aligned} R_s &= \sum_{a \in A} \sum_{s' \in S} p(s'|s,a)\pi(a|s)r(s',a,s) \\ &= E_{a \sim \pi(a|s), s' \sim p(s'|s,a)}[r(s',a,s)]. \end{aligned}$$

This suggests an alternative method to obtain V_π: from

$$V_\pi = R + \gamma P V_\pi$$

we get

$$V_\pi = (I - \gamma P)^{-1} R.$$

In simple cases, this might be a better approach than dynamic programming. But it has a bunch of disadvantages: First, the inverse $(I - \gamma P)^{-1}$ does not need to be sparse even if P is. This means that for larger MDPs that can still be stored in a computer by exploiting sparsity in the transition matrix, calculating the inverse might be infeasible, while dynamic programming still allows to approximately evaluate $(I - \gamma P)^{-1} R$. Then, dynamic programming actually does not need to store the P in the computer at all, but is able to compute V_π given only black box access to P: all that is needed for an approximate solution is the evaluation of PV for some values V. This becomes especially important when we later try to learn from simulations or want to learn optimal behavior, as discussed before, since both of these use cases lend themselves to rather straightforward extension of dynamic programming. For a more extensive discussion of the merits of iterative methods of solving systems of linear equation, see [9].

We will instead use equation 3.1.2 as inspiration for a decomposition of dynamic programming. If

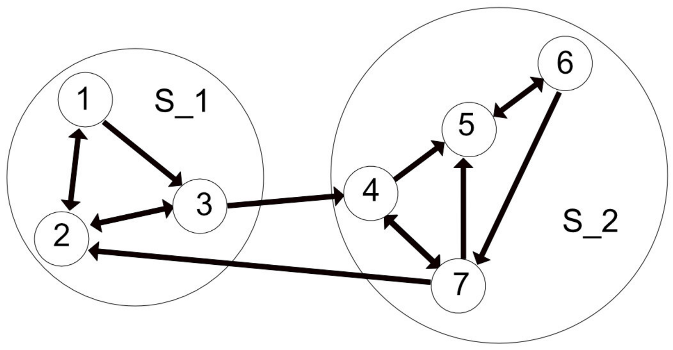


Figure 3: A decomposed MDP

there is a natural decomposition of the state space S into two disjoint "clusters" of states S_1 and S_2 with $S_1 \cup S_2 = S$, it is tempting to try and decompose the solution method as well. In case S_1 and S_2 are not reachable from the respective other cluster, this is trivial. Else, the influence of values in one cluster on those in the other has to be taken into account. After reindexing, we can assume that the first n states are in S_1 and the other m are in S_2. Then, we rewrite $P = \begin{pmatrix} P_1 & E_{1,2} \\ E_{2,1} & P_2 \end{pmatrix}$ with $P_1 \in \mathbb{R}^{n\times n}$, $P_2 \in \mathbb{R}^{m\times m}$, $E_{1,2} \in \mathbb{R}^{n\times m}$ and $E_{2,1} \in \mathbb{R}^{m\times n}$ for $n = |S_1|$ and $m = |S_2|$, where the entries of P_i are the probabilities for transitioning to a state in S_i when starting in such a state, while the entries of $E_{i,j}$ contain the probabilities for transitioning from S_i to S_j. Similarly, we can decompose the vector of expected rewards, $R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$ with $R_1 \in \mathbb{R}^n$ and $R_2 \in \mathbb{R}^m$, where R_i contains the immediate expected rewards for taking a step from a state in S_i. With this and after decomposing $V_\pi = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}$, the equation

$$V_\pi = R + \gamma P V_\pi$$

becomes

$$\begin{pmatrix} V_1 \\ V_2 \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} + \gamma \begin{pmatrix} P_1 & E_{1,2} \\ E_{2,1} & P_2 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \tag{3.1.3}$$

which implies

$$V_i = R_i + \gamma P_i V_i + \gamma E_{i,j} V_j$$

for $i \neq j$. This gives us

$$V_1 = (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} V_2), \tag{3.1.4}$$

where we use that $\|\gamma P_1\|_\infty \leq \gamma < 1$, since $\|P\|_\infty = 1$ and that the sums over row entries determining this norm cannot get larger for submatrices. Therefore $(I - \gamma P_1)^{-1} = \sum_{k=0}^{\infty} \gamma^k P_1^k$ and the inverse is in particular well defined. With this, we can eliminate the dependence of V_2 on V_1:

$$V_2 = R_2 + \gamma P_2 V_2 + \gamma E_{2,1} V_1 = R_2 + \gamma P_2 V_2 + \gamma E_{2,1}(I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} V_2).$$

Intuitively, the first term corresponds to the immediate reward obtained in R_2, while the second term describes the return until leaving S_2. The last term describes the reward obtained in S_1 before leaving (represented by the R_1 part) plus the return after going back to S_2 (represented by the V_2 part). Provided that the affinely linear right hand side still is a contraction, we can use it to iteratively solve for V_2, once we have $(I - \gamma P_1)^{-1}$: There are now two main cases, where this might be useful: Either $I - \gamma P_1$ has a special structure that makes the inverse easy to calculate, while this is not true for $I - \gamma P$ (the inverse of which could be used to directly solve for V_π), or $(I - \gamma P_1)^{-1}$ is hard to obtain but reusable: since this only depends on P_1 and thus the dynamics on S_1, it is possible to find lots of different MDPs that share the same S_1 including dynamics. If one wants to solve multiple of those, calculating $(I - \gamma P_1)^{-1}$ first might make sense, even if it was costly.

We will now show, that

$$\begin{aligned} TV &:= R_2 + \gamma P_2 V + \gamma E_{2,1}(I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} V) \\ &= \gamma E_{2,1}(I - \gamma P_1)^{-1} R_1 + R_2 + \gamma(P_2 + E_{2,1}(I - \gamma P_1)^{-1}\gamma E_{1,2})V \end{aligned}$$

mapping $\mathbb{R}^m$ to itself in fact defines a contraction. Since additive constants do not matter for the contraction property of affinely linear operators, it is sufficient to show that the matrix

$$M := \gamma P_2 + \gamma E_{2,1}(I - \gamma P_1)^{-1}\gamma E_{1,2}$$

has a norm of less than one. We will have a look at its ∞-matrix norm, i.e. $\max_i \sum_j |M_{i,j}|$ and note that this is not trivial because

$$\|(I - \gamma P_1)^{-1}\| = \|\sum_{k=0}^{\infty}(\gamma P_1)^k\| \leq \sum_{k=0}^{\infty}\|(\gamma P_1)^k\| \leq \sum_{k=0}^{\infty}\|(\gamma P_1)\|^k = \frac{1}{1 - \gamma\|P_1\|}$$

where all bounds are sharp for diagonal and positive P_1. Without using further information about the structure and assuming that all of the involved matrices have norm one, the tightest bound we can get on $\|M\|$ is thus $\gamma + \frac{\gamma^2}{1-\gamma}$ which is smaller than one only for steep discount rates

$\gamma < 0.5$. Luckily, there is more structure to the problem: First, we rewrite

$$\|M\|_\infty = \|\gamma P_2 + \gamma E_{2,1}(\sum_{k=0}^{\infty} \gamma^k P_1^k)\gamma E_{1,2}\|_\infty$$

$$= \|\gamma P_2 + \gamma^2 \sum_{k=0}^{\infty} \gamma^k E_{2,1} P_1^k E_{1,2}\|_\infty$$

$$\leq \|\gamma P_2 + \gamma^2 \sum_{k=0}^{\infty} E_{2,1} P_1^k E_{1,2}\|_\infty,$$

where we used continuity of linear operators and that the $E_{i,j}$ have a norm of at most one and therefore don't affect the (absolute!) convergence of the Von-Neumann series, as well as the non-negativity of all involved matrices (their entries are probabilities!) combined with the fact that pointwise dominance implies dominance in the maximum norm. Next, we will prove that $\|\gamma P_2 + \gamma^2 \sum_{k=0}^{\infty} E_{2,1} P_1^k E_{1,2}\|_\infty$ is actually well defined, as well as small.

In order to do this, we will look through the lens of probability: $(P_1)_{i,j} = p_\pi(j|i)$ for $j, i \leq n$, while $(E_{1,2})_{i,j} = p_\pi(j+n|i)$ for $i \leq n, j \leq m$ and $(E_{2,1})_{i,j} = p_\pi(j|i+n)$ for $i \leq m = |S_2|, j \leq n = |S_1|$. Accordingly, with $p_h(b|a)$ denoting the probability to be in state b after starting in a and following π for h steps, without visiting any state in S_2 (a state with index bigger than n) in between, we have that

$$(E_{2,1}E_{1,2})_{i,j} = \sum_{k=0}^{n} p(k|i+n)p(j+n|k) = p_2(j+n|i+n),$$

$$(P_1^2)_{i,j} = \sum_{k=0}^{n} p(k|i)p(j|k) = p_2(j|i),$$

and analogous

$$(P_1^h)_{i,j} = p_h(j|i).$$

Similarly,

$$(E_{2,1}P_1^h E_{1,2})_{i,j} = p_{h+2}(j+n|i+n),$$

while per definition

$$(P_2)_{i,j} = p_1(j+n|i+n).$$

Correspondingly,

$$(P_2 + \sum_{k=0}^{\infty} E_{2,1}P_1^k E_{1,2})_{i,j} = \sum_{k=1}^{\infty} p_k(j+n|i+n),$$

which is equal to the probability of the first state in S_2 that is visited being $j+n$, after starting in state $i+n$. Since there can be only one first visited state in S_2, summing over j yields at most one. But that is equivalent to

$$\|P_2 + \sum_{k=0}^{\infty} E_{2,1} P_1^k E_{1,2}\|_\infty \leq 1$$

and thus by positivity

$$\|M\|_\infty \leq \|\gamma P_2 + \gamma^2 \sum_{k=0}^{\infty} E_{2,1} P_1^k E_{1,2}\|_\infty \leq \gamma,$$

making M a γ-contraction. Keeping γ in the equations would yield

$$(P_2 + \sum_{k=0}^{\infty} \gamma E_{2,1} P_1^k E_{1,2})_{i,j} = \sum_{k=1}^{\infty} \gamma^k p_k(j+n|i+n),$$

so M represents the discounted transition probabilities within S_2, as in the option models used in Sutton, Precup and Singh [12].

The skeptical reader might ask herself about the point of this exercise: In general we still only get a γ-contraction and therefore no increase in the speed of convergence. However, getting rid of the powers of γ in the Von-Neumann series is far from a tight bound, especially if states usually stay in S_1 for a while and still the second term has a γ^2 in front, improving convergence speed, if $\|P_2\|_\infty$ is small, i.e. transitions from S_2 to S_1 are likely in all states. Furthermore, steps also get faster: The matrix-vector product and the vector addition in

$$V_\pi = R + \gamma P V_\pi$$

take us $O((n+m)^2)$ point operations per iteration step. The decomposed iteration

$$TV := \gamma E_{2,1}(I - \gamma P_1)^{-1} R_1 + R_2 + \gamma(P_2 + E_{2,1}(I - \gamma P_1)^{-1} \gamma E_{1,2})V$$

in $\mathbb{R}^m$ only needs $O(m^2)$ operations per step. This comes at the cost of an additional one time overhead of at least $O(n^2 m + nm^2)$ for the precomputation of

$$\gamma E_{2,1}(I - \gamma P_1)^{-1} R_1 + R_2$$

and

$$\gamma(P_2 + E_{2,1}(I - \gamma P_1)^{-1} \gamma E_{1,2})$$

(using standard matrix multiplication), as well as the cost of computing or approximating $(I - \gamma P_1)^{-1}$. If this is tractable, the outlined approach can thus be more attractive than iteratively solving the complete system, especially if γ is large and many steps are needed. Also, parts of the precomputations can be reused when solving an MDP with the same decomposition and other rewards, or even another policy on S_2 (which only influences P_2 and $E_{2,1}$). Trying to directly solve the complete system is usually not a good idea, especially for larger MDPs, as discussed earlier. When solving multiple similar problems (with the same S_1), the convergence of iterative approaches can also be sped up quite a bit by good initialization (for example, one could initialize V for a problem with the correct V for a previously solved similar problem). But the main advantage of the outlined approach is that it can be generalized to non-linear operators on S_2, such that we can (at least partially) decompose the optimization of our policy, as explored in the next section.

3.2 Asking the expert

Now, we extend the approach from policy evaluation to finding (partially) optimal policies. One could obviously use the obtained V_π and perform a greedy improvement in order to improve the policy, but if this changes the policy on S_1, we need to recompute $(I - \gamma P_1)^{-1}$ and $E_{1,2}$, which can be problematic. Thus, we focus on local improvements: can we efficiently improve the policy by just modifying it on S_2? This can be reframed as an expert giving the agent a fixed policy on S_1 (that should work reasonably well) and the agent learning, how to act on S_2 in order to leverage the expertise as good as possible (while opting to avoid states where the expert takes over, if beneficial). To reflect this, we rename

$$S_{fix} := S_1$$

and

$$S_{learn} := S_2.$$

In the following approach, the expert will also make an accuracte prediction of what their policy will achieve in the current situation. This way, the agent can learn to use the expert's knowledge without actually having to carry out their policy.

In fact, we want to find the optimal policy π_2^* on S_{learn}, given a fixed policy π_1 on S_{fix}. This exists in the sense of no further policy improvement on S_{learn} being possible. Formally, it is the restriction to S_{learn} of the optimal policy of a modified MDP $M|_{\pi_1} = (S, A, p', r', p_0)$, which always behaves as if policy π_1 was carried out on S_{fix}: the rewards r' and the transition probabilities p' for transitions starting in S_{learn} are equal to the ones in the base MDP M, r and p. In S_{fix}, transitions to the next state s' happen with probability $\sum_{a \in A} \pi_1(a|s)p(s|'s, a)$, independent of the taken action, whereupon the action-independent reward for this transition, $r'(s', s)$, is just the average reward for that transition $\sum_{a \in A} \pi_1(a|s)r(s', a, s)$. For $s \in S_{learn}$, the Bellman optimality equation for the optimal action value function of $M|_{\pi_1}$, $Q^*|_{S_{learn} \times A} =: Q^*|_{\pi_1}$ then becomes

$Q^*|_{\pi_1}(s, a)$

$$
\begin{aligned}
&= \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*(s',a')) \\
&= \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q|^*_{\pi_1}(s',a')) + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*(s',a')) \\
&= \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q|^*_{\pi_1}(s',a')) + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma V_{\pi_1}|_{\pi_2^*}(s'))
\end{aligned}
$$

with $V_{\pi_1}|_{\pi_2^*}$ denoting the V-values in $M|_{\pi_1}$ for the policy π_2^* on S_{learn} induced by $Q^*|_{\pi_1}$ (the policy on S_{fix} used in the modified MDP is irrelevant by definition), which is equal to $\max_{a' \in A} Q^*(s',a')$ for $s' \in S_{fix}$, since actions are irrelevant on S_{fix} for the modified MDP. Since the transitions starting from states in S_{fix} only depend on the policy π_1, we can reuse the matrices $(P_1)_{i,j} = p_{\pi_1}(j|i)$ and $(E_{1,2})_{i,j} = p_{\pi_1}(j+n|i)$ from the previous section to obtain that

$$
V_{\pi_1}|_{\pi_2^*} = (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} V_{\pi_2^*}|_{\pi_1}),
$$

using the same notation as above and with $V_{\pi_2^*}|_{\pi_1}$ denoting the values of states in S_{learn} given policy π_1 on S_{fix} and π_2^* on S_{learn}. Using that

$$
\max_{a' \in A} Q^*(s',a') = V_{\pi_2^*}|_{\pi_1}(s')
$$

for $s' \in S_{learn}$ and the optimal (conditional) deterministic policy on S_{learn}, π_2^*, the Bellman optimality equation can be rewritten as

$$
\begin{aligned}
&Q^*|_{\pi_1}(s,a) \\
&\quad = \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*|_{\pi_1}(s',a')) \\
&\quad + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma((I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q^*|_{\pi_1}))(s'))
\end{aligned} \tag{3.2.1}
$$

for $s \in S_{learn}$ and $\max Q^*|_{\pi_1}(s') = \max_{a' \in A} Q^*|_{\pi_1}(s',a')$. Intuitively, this corresponds to a normal Q-learning update for transitions within S_{learn}, while we use $\gamma(I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q^*|_{\pi_1})$ as a dynamic model for the return upon entering S_{fix}. When S_{fix} is entered, we do a two step update where the second, model-aided step actually corresponds to a multi-step update of variable length, since it tries to predict the return in S_{fix} from values in S_{learn} while the time needed to reach S_{learn} is stochastic.

Now by defining the Bellman expert operator by

$$
\begin{aligned}
TQ(s,a) := &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\
&+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma((I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q))(s')),
\end{aligned}
$$

we can formulate the first main theorem:

Theorem 3.2.1. *The Bellman expert operator $T : \mathbb{R}^{S_{learn} \times A} \to \mathbb{R}^{S_{learn} \times A}$ is a γ-contraction and therefore has $Q^*|_{\pi_1}$ as a unique fixed point, to which the sequence $T^n Q$ converges for all $Q \in \mathbb{R}^{S_{learn} \times A}$.*

In order to show that, we look at it from a different perspective: applying T implicitly calculates $V_{\pi_1|\pi_2^*}$ (in the second sum) and we will make it explicit again: Instead of looking at T, we will first focus on $T_1 T_2$ acting on the pair $\begin{pmatrix} V \\ Q \end{pmatrix}$ (where the Q-part is indexed by state-action pairs for states in S_{learn} and the V-part by states in S_{fix} without actions), where

$$T_2(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q) \\ Q \end{pmatrix}$$

with $\max Q(s') = \max_{a' \in A} Q(s', a')$ and

$$T_1(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} V \\ \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma V(s')) \end{pmatrix},$$

with s, a being the respective index in the corresponding equation. With help of the following lemma, it will be enough to show that both T_1 and T_2 contract parts of the (V, Q) space in a specific sense. It states that the composition of two operators that each contract some dimensions, while leaving the others invariant is a contraction in the maximum norm, as long as every dimension is contracted by exactly one of the operators. The simplest example is two matrices A and B with $\|A\|_\infty = \|B\|_\infty = \gamma < 1$ whose product $\begin{pmatrix} A & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & B \end{pmatrix} = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}$ is a contraction again. However, by the following lemma, the considered operators don't need to be linear and the non-invariant part is allowed to depend on the whole space:

Lemma 3.2.2. *Let $V = V1 \oplus V2$ be a vector space with $\oplus$ denoting the internal direct sum. In other words, every vector $v \in V$ can be written as a unique sum $v_1 + v_2$ for $v_1 \in V_1, v_2 \in V_2$. Furthermore, let the norm on V be compatible with the direct sum in the sense that $\|v_1 + v_2\| = \max\{\|v_1\|, \|v_2\|\}$. Let T_1 and T_2 be operators mapping V to itself with $T_1(v_1 + v_2) = \tilde{T}_1 v + v_2$ and $T_2(v_1 + v_2) = v_1 + \tilde{T}_2 v$ with $\tilde{T}_i$ mapping V to V_i and $\|\tilde{T}_i v - \tilde{T}_i w\| \le \gamma \|v - w\|$. Then $\|T_1 T_2 v - T_1 T_2 w\| \le \gamma \|v - w\|$. In other words, $T_1 T_2$ is a contraction.*

Proof. Decompose $v = v_1 + v_2, w = w_1 + w_2$ for $v_1, w_1 \in V_1$ and $v_2, w_2 \in V_2$. Then:

$$\begin{aligned} &\|T_1 T_2 v - T_1 T_2 w\| \\ &= \|T_1 T_2(v_1 + v_2) - T_1 T_2(w_1 + w_2)\| \\ &= \|T_1(v_1 + \tilde{T}_2 v) - T_1(w_1 + \tilde{T}_2 w)\| \\ &= \|\tilde{T}_1(v_1 + \tilde{T}_2 v) + \tilde{T}_2 v - \tilde{T}_1(w_1 + \tilde{T}_2 w) - \tilde{T}_2 w\| \end{aligned}$$

$$
\begin{aligned}
&= \max\{\|\tilde{T}_1(v_1+\tilde{T}_2 v) - \tilde{T}_1(w_1+\tilde{T}_2 w)\|, \|\tilde{T}_2 v - \tilde{T}_2 w\|\}\\
&\leq \max\{\ \gamma\|v_1+\tilde{T}_2 v - w_1 - \tilde{T}_2 w\|, \gamma\|v-w\|\}\\
&= \max\{\ \gamma\max\{\|v_1-w_1\|, \|\tilde{T}_2 v - \tilde{T}_2 w\|\}, \gamma\|v-w\|\}\\
&\leq \max\{\ \gamma\max\{\|v_1-w_1\|, \gamma\|v-w\|\}, \gamma\|v-w\|\}\\
&= \gamma\|v-w\|.
\end{aligned}
$$

□

As alluded to earlier, the decomposition of $\mathbb{R}^{(S_{learn}\times A)\cup S_{fix}}$ equipped with the maximum norm into subspaces generated by a partition of the standard basis is a use case of the lemma, which translates perfectly to the application to the operators T_1 and T_2 from above. In order to show that T_1T_2 is a γ-contraction in the maximum norm, it is then sufficient to show that $\tilde{T}_1$ and $\tilde{T}_2$ with

$$\tilde{T}_2(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} (I-\gamma P_1)^{-1}(R_1 + \gamma E_{1,2}\max Q) \\ 0 \end{pmatrix}$$

and

$$\tilde{T}_1(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} 0 \\ \sum_{s'\in S_{learn}} p(s'|s,a)(r(s',a,s)+\gamma\max_{a'\in A} Q(s',a')) + \sum_{s'\in S_{fix}} p(s'|s,a)(r(s',a,s)+\gamma V(s')) \end{pmatrix}$$

are. Since $\tilde{T}_2$ does not dependtd on V, it is enough to show that $(I-\gamma P_1)^{-1}\gamma E_{1,2}$ has at most norm one in order to see that the affine operator $\tilde{T}_2$ is a $\gamma-$contraction. Again, it helps to rewrite

$$\|(I-\gamma P_1)^{-1}\gamma E_{1,2}\|_\infty = \|(\sum_{k=0}^{\infty}\gamma^k P_1^k)\gamma E_{1,2}\|_\infty \leq \gamma\|\sum_{k=0}^{\infty} P_1^k E_{1,2}\|_\infty$$

and look through the lens of probability: with $p_h(a|b)$ denoting the probability to be in state b after h steps following policy π_1 and starting in a, without visiting any state in S_{learn} in between, we have for $n = |S_{fix}|$ and $m = |S_{learn}|$ that

$$(P_1^h)_{i,j} = p_h(j|i)$$

for $i,j \leq n$ and together with $(E_{1,2})_{i,j} = p_{\pi_1}(j+n|i)$, we get that

$$(P_1^h E_{1,2})_{i,j} = p_{h+1}(j+n|i)$$

for $j \leq m$ and $i \leq n$. As previously, this means that $(\sum_{k=0}^{\infty} P_1^k E_{1,2})_{i,j}$ is equal to the probability of the first state in S_{learn} that is reached being $j+n$, after starting in i. Again, this implies that summing over j for any i yields at most one, since there can only be one first visited state.

Therefore, $\|(I-\gamma P_1)^{-1}\gamma E_{1,2}\|_\infty \leq \gamma$ and $\tilde{T}_2$ fulfills the requirements of lemma 3.2.2 . Now,

$$\begin{aligned}
&\|\tilde{T}_1 \begin{pmatrix} V_1 \\ Q_1 \end{pmatrix} - \tilde{T}_1 \begin{pmatrix} V_2 \\ Q_2 \end{pmatrix}\|_\infty \\
&= \max_{s,a\in S,A} |\sum_{s'\in S_{learn}} p(s'|s,a)\gamma(\max_{a'\in A} Q_1(s',a')) - \max_{a'\in A} Q_2(s',a')) + \sum_{s'\in S_{fix}} p(s'|s,a)\gamma(V_1(s') - V_2(s'))| \\
&\leq \gamma \max_{s,a}(\sum_{s'\in S_{learn}} p(s'|s,a)|(\max_{a'\in A} Q_1(s',a')) - \max_{a'\in A} Q_2(s',a'))| + \sum_{s'\in S_{fix}} p(s'|s,a)|(V_1(s') - V_2(s'))|) \\
&\leq \gamma \max\{ \max_{s'\in S_{learn}} |\max_{a'\in A} Q_1(s',a')) - \max_{a'\in A} Q_2(s',a'))|, \max_{s'\in S_{fix}} |(V_1(s') - V_2(s'))|\} \\
&\leq \gamma \max\{ \max_{s',a'\in S_{learn},A} |Q_1(s',a')) - Q_2(s',a'))|, \max_{s'\in S_{fix}} |(V_1(s') - V_2(s'))|\} \\
&= \gamma\| \begin{pmatrix} V_1 \\ Q_1 \end{pmatrix} - \begin{pmatrix} V_2 \\ Q_2 \end{pmatrix}\|_\infty
\end{aligned}$$

since the $p(s'|s,a)$ sum to one for any $s,a \in S,A$ and all real convex combinations are bounded in absolute value by their constituent with the largest absolute value and therefore their maximum is as well. Thus, T_2 fulfills the requirements for lemma 3.2.2 as well and T_1T_2 is a contraction. Therefore, iterative application of T_1T_2 starting with an arbitrary $\begin{pmatrix} V \\ Q \end{pmatrix}$ pair will converge to $\begin{pmatrix} V_{\pi_1}|_{\pi_2^*} \\ Q^*|_{\pi_1} \end{pmatrix}$. In order to save memory and to simplify the later extension towards learning from samples, we can also extract an iteration that only operates on Q-values for S_{learn} and still converges to $Q^*|_{\pi_1}$, using the following lemma:

Lemma 3.2.3. *Let $V = V1 \oplus V2$ and T_1, T_2 be as in lemma 3.2.2 and let $\tilde{T}_2$ be independent of V_2. Then the operator T defined by*

$$T(v_1) := \tilde{T}_1(v_1 + \tilde{T}_2 v_1)$$

for all $v_1 \in V_1$ is a contraction. With $V_1^* \in V_1$ and $V_2^* \in V_2$ such that $V^* = V_1^* + V_2^*$ is the fixed point of T_1T_2, V_1^* is the fixed point of T.

Proof. Let v, w be in V_1:

$$\begin{aligned}
&\|Tv - Tw\| \\
&= \|\tilde{T}_1(v + \tilde{T}_2 v) - \tilde{T}_1(w + \tilde{T}_2 w)\| \\
&\leq \gamma\|v + \tilde{T}_2 v - w - \tilde{T}_2 w\| \\
&= \gamma \max\{\|v - w\|, \|\tilde{T}_2 v - \tilde{T}_2 w\|\} \\
&\leq \gamma \max\{\|v - w\|, \gamma\|v - w\|\} \\
&= \gamma\|v - w\|
\end{aligned}$$

Now,

$$\begin{aligned}
&V_1^* + V_2^* \\
&= T_1 T_2 (V_1^* + V_2^*) \\
&= T_1 (V_1^* + \tilde{T}_2 (V_1^* + V_2^*)) \\
&= T_1 (V_1^* + \tilde{T}_2 (V_1^*)) \\
&= \tilde{T}_1 (V_1^* + \tilde{T}_2 (V_1^*)) + \tilde{T}_2 (V_1^*)
\end{aligned}$$

And thus

$$V_1^* = \tilde{T}_1 (V_1^* + \tilde{T}_2 (V_1^*)) = T(V_1^*).$$

$\square$

Since $\tilde{T}_2$ does not depend on the V-part, we can apply the lemma to obtain that our operator T is a contraction with the desired fixed point.

By replacing the Bellman optimality equation for Q in this section with the Bellman equation for Q given a fixed policy on S_{learn}, we can also find Q_π for the joint policy on S_{fix} and S_{learn}. Because we are mostly interested in finding good policies instead of evaluating arbitrary ones, the discussion of this will be ommitted in order to safe space.

3.3 A committee of experts

Fixing a single policy on S_{fix} can potentially lead to a very suboptimal global policy. On the other hand, the gained efficiency and reusability of the outlined approach does not seem to readily combine with an optimality operator on S_{fix} instead of the evaluation operator (one could alternate the roles of S_{fix} and S_{learn} in order to converge to a globally optimal policy by local improvements, but since this would mean recalculating the inverse in every alternation, just solving the global problem on S would very likely be more efficient). We can, however, fix not just one policy on S_{fix} but multiple and always select the best situational policy. This corresponds to a committee of experts where each expert suggests a policy and gives us an accurate prediction of what the consequences of the policy will be in the current situation. Given a collection of policies on S_{fix}, $\Pi = (\pi^k)_{k \leq n \in \mathbb{N}}$, we want to act optimally on S_{learn} assuming that whenever we transition to S_{fix}, we have to select a fixed policy from Π to carry out until S_{fix} is left again.

This can be reframed as solving a modified MDP, where the dynamics and rewards for S_{learn} are the same as in the base MDP M for the actions in M, while for S_{fix} we introduce the policies π^k as actions that lead to transitions to a lifted state space. This lifted space includes the base state as well as the policy, in which transitions change the base state as that policy would, independently of the actual taken action and the reward is the reward for that transition,

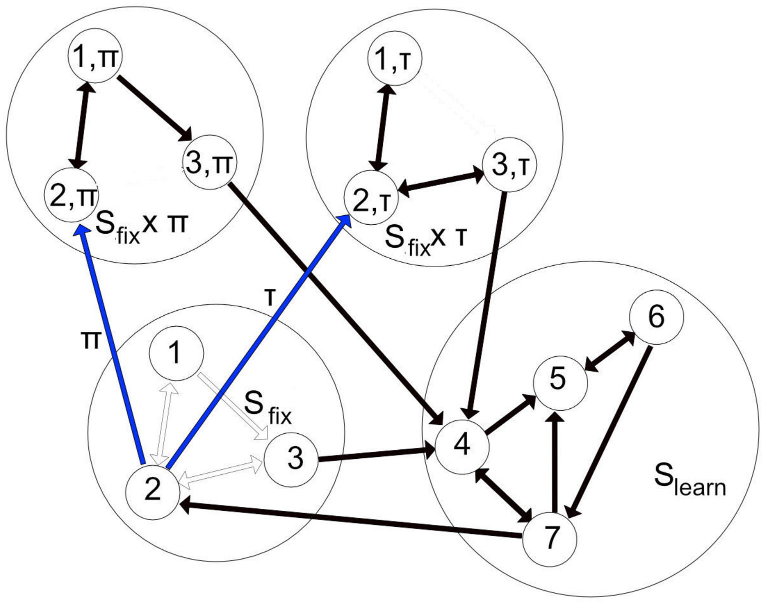


Figure 4: Lifted MDP

averaged over the actions in the policy. Then, a transition of the base state to a state s in S_{learn} reverses the lifting. In order to handle actions that don't make sense in that interpretation, incompatible actions are penalized with a maximally suboptimal transition to E, for which the reward is guaranteedly worse than the discounted return for any episode that does not visit E. (If we had allowed for action sets depending on the state in our definition of an MDP, we would not need to introduce E and could instead just adjust the action sets accordingly, but this would have further complicated notation in most other situations). More technically, we consider $M|_\Pi = ((S_{fix} \cup (S_{fix} \times \Pi) \cup S_{learn} \cup E, A \cup \Pi, q, b, p_0)$ defined as follows:

$$q(s'|s,a) = p(s'|s,a), b(s',a,s) = r(s',a,s)$$

for $a \in A, s \in S_{learn}$ and $s' \in S_{fix} \cup S_{learn}$,

$$q((s',\pi^k)|s,\pi^k) = \sum_{a \in A} p(s'|s,a)\pi^k(a|s)$$

as well as

$$b((s',\pi^k),\pi^k,s) = \sum_{a \in A} r(s',a,s)\pi^k(a|s)$$

for $s, s' \in S_{fix}, \pi^k \in \Pi$,

$$q((s', \pi^k)|(s, \pi^j), a) = \delta_{k,j} \sum_{a' \in A} p(s'|s, a')\pi^k(a'|s)$$

and

$$b((s', \pi^k), a, (s, \pi^j)) = \sum_{a' \in A} r(s', a', s)\pi^k(a'|s)$$

for $s, s' \in S_{fix}, \pi^k, \pi_j \in \Pi$ and $a \in A \cup \Pi$,

$$q(s'|(s, \pi^k), a) = \sum_{a' \in A} p(s'|s, a')\pi^k(a'|s)$$

as well as

$$b(s', a, (s, \pi^k)) = \sum_{a' \in A} r(s', a', s)\pi^k(a'|s)$$

for $s \in S_{fix}, s' \in S_{learn}, \pi^k \in \Pi$ and $a \in A \cup \Pi$. Lastly, we have

$$q(E|s, a) = 1$$

if $a \notin A, s \in S_{learn}$ or $a \in A, s \in S_{fix}$ or $s = E$ with arbitrary a and

$$q(E|s, a) = 0$$

else, as well as

$$b(E, a, s) = -\frac{1}{1-\gamma} \max_{s',a,s} |r(s', a, s)|$$

for all $a \in A \cup \Pi, s \in (S_{fix} \cup (S_{fix} \times \pi) \cup S_{learn} \cup E)$. The transitions probabilities q are zero for all configurations that weren't mentioned and the corresponding rewards can accordingly be chosen arbitrarily. The corresponding Bellman optimality equation for π_2^*, the optimal policy on $M|_\Pi$, restricted to S_{learn} can now be derived by setting $Q^*|_\Pi = Q^*|_{S_{learn} \times A}$ for $s \in S_{learn}$ and $a \in A$ and the optimal Q-function Q^* on $M|_\Pi$:

$$\begin{aligned}
Q^*|_\Pi(s, a) &= \sum_{s' \in S_{fix} \cup (S_{fix} \times \Pi) \cup S_{learn} \cup E} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) \\
&= \sum_{s' \in S_{fix} \cup (S_{fix} \times \Pi) \cup S_{learn}} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) + q(E|s, a)(b(E, a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(E, a')) \\
&= \sum_{s' \in S_{fix} \cup S_{learn}} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) + \sum_{s' \in S_{fix} \times \Pi} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) + 0 \\
&= \sum_{s' \in S_{learn}} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) + \sum_{s' \in S_{fix}} q(s'|s, a)(b(s', a, s) + \gamma \max_{a' \in A \cup \Pi} Q^*(s', a')) + 0
\end{aligned}$$

$$= \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*(s',a')) + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{\pi^k \in \Pi} Q^*(s',\pi^k)(s')),$$

since then $q(E|s,a) = 0$, direct transitions from S_{learn} to $S_{fix} \times \Pi$ don't happen and by construction $Q^*(s',\pi^k)$ cannot be greater than $Q^*(s',a)$ for $s' \in S_{learn}, a \in A$ and $\pi^k \in \Pi$ and will be greater for $s' \in S_{fix}$. Now, we take a closer look at $Q^*(s,\pi^k)$ for a given $s \in S_{fix}$: with P_{π^k} taking the role of P_1 from the last chapter for a given policy $\pi^k \in \Pi$ and denoting the transition probabilities from S_{fix} to itself in the original MDP M using π^k, E_{π^k} taking the role of $E_{1,2}$ and encoding the transitions from S_{fix} to S_{learn} in M using π^k and R_{π^k} taking the role of R_1 and denoting the expected immediate rewards for transitions from S_{fix} in M using π^k, we get

$$Q^*(s,\pi^k) = \sum_{a \in A} \pi^k(a|s) \sum_{s' \in S_{learn} \cup (S_{fix} \times \pi^k)} p(s'|s,a)(r(s',a,s) + \gamma V^*(s')) = R_{\pi^k} + \gamma (P_{\pi^k}, E_{\pi^k}) \begin{pmatrix} V^*|_{S_{fix} \times \pi^k} \\ V^*|_{S_{learn}} \end{pmatrix},$$

where V^* is the optimal value function on $M|_\Pi$. Next, we look at the decomposition from the previous chapter to derive $V^*|_{S_{fix} \times \pi^k}$ from values on S_{learn}: By reordering the state space as $(S_{fix} \times \pi_k, \{S_{fix} \times \pi_{\neq j} | k \neq j \leq n\}, S_{fix}, S_{learn})$ and omitting E, since we know that an optimal policy never reaches that state, we obtain the following transition matrix:

$$\begin{array}{ccccc} S_{fix} \times \pi_k & \{S_{fix} \times \pi_j\} & S_{fix} & S_{learn} & \\ P_{\pi^k} & 0 & 0 & E_{\pi^k} & S_{fix} \times \pi_k \\ 0 & \begin{pmatrix} P_{\pi^j} & \\ & \cdots \end{pmatrix} & 0 & \begin{pmatrix} E_{\pi^j} \\ \cdots \end{pmatrix} & \{S_{fix} \times \pi_j\} \\ * & * & 0 & * & S_{fix} \\ 0 & 0 & * & * & S_{learn} \end{array},$$

where the question marks correspond to parts that depend on the policy on S_{learn}. Using the procedure we used to obtain 3.1.4, we get

$$V^*|_{S_{fix} \times \pi^k} = (I - \gamma P_{\pi^k})^{-1} (R_{\pi^k} + \gamma \begin{pmatrix} 0 & 0 & E_{\pi^k} \end{pmatrix} \begin{pmatrix} V^*|_{\{S_{fix} \times \pi_j\}} \\ V^*|_{S_{fix}} \\ V^*|_{S_{learn}} \end{pmatrix}) = (I - \gamma P_{\pi^k})^{-1} (R_{\pi^k} + \gamma E_{\pi^k} (V^*|_{S_{learn}}).$$

In total, we get

$$Q^*(s,\pi^k) = R_{\pi^k} + \gamma (P_{\pi^k}, E_{\pi^k}) \begin{pmatrix} V^*|_{S_{fix} \times \pi^k} \\ V^*|_{S_{learn}} \end{pmatrix} = R_{\pi^k} + \gamma (P_{\pi^k} (V^*|_{S_{fix} \times \pi^k}) + E_{\pi^k} (V^*|_{S_{learn}}))$$
$$= R_{\pi^k} + \gamma (P_{\pi^k} (I - \gamma P_{\pi^k})^{-1} (R_{\pi^k} + \gamma E_{\pi^k} (V^*|_{S_{learn}})) + \gamma E_{\pi^k} (V^*|_{S_{learn}})$$
$$= R_{\pi^k} + \gamma (P_{\pi^k} \sum_{i=0}^{\infty} \gamma^i P_{\pi^k}^i (R_{\pi^k} + \gamma E_{\pi^k} (V^*|_{S_{learn}})) + \gamma E_{\pi^k} (V^*|_{S_{learn}})$$

$$
\begin{aligned}
&= \sum_{i=0}^{\infty} \gamma^{i+1} P_{\pi^k}^{i+1}(R_{\pi^k} + \gamma E_{\pi^k}(V^*|_{S_{learn}})) + R_{\pi^k} + \gamma E_{\pi^k}(V^*|_{S_{learn}}) \\
&= \sum_{i=0}^{\infty} \gamma^{i+1} P_{\pi^k}^{i+1}(R_{\pi^k} + \gamma E_{\pi^k}(V^*|_{S_{learn}}) + \gamma^0 P_{\pi^k}^0(R_{\pi^k} + \gamma E_{\pi^k}(V^*|_{S_{learn}})) \\
&= (I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k}(V^*|_{S_{learn}})).
\end{aligned}
$$

We can plug this in our equation for $Q^*|_\Pi$ to obtain

$$
\begin{aligned}
Q^*|_\Pi(s,a) = &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*|_\Pi(s',a')) \\
&+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} V^*|_{S_{learn}}))(s')) \\
= &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*|_\Pi(s',a')) \\
&+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q^*|_\Pi))(s'))
\end{aligned}
$$

for $s \in S_{learn}$, $a \in A$ with $\max Q^*|_\Pi(s') = \max_{a' \in A} Q^*|_\Pi(s',a')$ and using that $V^* = \max Q^*$ for any MDP. The first term corresponds to normal Q-learning, while the second term corresponds to first selecting the best situational expert policy (via the maximum) and then updating the Q-value to match the predicted return for taking this policy using the conditional model $(I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q^*|_\Pi)$ that predicts the return after choosing π^k when entering S_{fix}. Now, with defining

$$
\begin{aligned}
TQ(s,a) := &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\
&+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q))(s')),
\end{aligned}
$$

we can formulate the second main theorem:

Theorem 3.3.1. *The operator $T : \mathbb{R}^{S_{learn} \times A} \to \mathbb{R}^{S_{learn} \times A}$ is a $\gamma-$contraction and therefore has $Q^*|_\Pi$ as a unique fixed point, to which the sequence $T^n Q$ converges for all $Q \in \mathbb{R}^{S_{learn} \times A}$.*

In order to show that T contracts and the fixed point is unique, we will go the same route as in the previous chapter: for $\begin{pmatrix} V \\ Q \end{pmatrix} \in \mathbb{R}^{S_{learn} \times A \cup S_{fix}}$ we define

$$
T_2(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} \max_{\pi^k \in \Pi}(I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q) \\ Q \end{pmatrix}
$$

$$
T_1(\begin{pmatrix} V \\ Q \end{pmatrix}) := \begin{pmatrix} V \\ \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) + \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma V(s')) \end{pmatrix},
$$

where the maximum over policies is to be read pointwise, and show that $T_1 T_2$ contracts in order to see that T does so by lemma 3.2.3. Since T_1 has the same form as in the previous section, we already know that it fulfills the conditions for lemma 3.2.2. For T_2, it is only left to show that

$$\tilde{T_2}(\begin{pmatrix} V \\ Q \end{pmatrix}) = \begin{pmatrix} \max_{\pi^k \in \Pi}(I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q) \\ 0 \end{pmatrix}$$

contracts. This can be seen easily by combining the fact that this is a γ-contraction in the maximum norm for every Π only containing one element, as in the previous section, with the following lemma:

Lemma 3.3.2. *Let $\{f^j\}_{j\in J}$ be γ-contractions $\mathbb{R}^d \to \mathbb{R}^d$ for the same $\gamma < 1$ with respect to the maximum norm for some finite index set J. Then the pointwise maximum g of the f^j with $g_i(x) = \max_j\{f^j(x)_i\}$ is a γ-contraction as well.*

Proof. Since $\|g(x) - g(y)\|_\infty = \max_i |g(x)_i - g(y)_i|$, it is enough to show that $|g(x)_i - g(y)_i| \leq \gamma\|x - y\|_\infty$ for all components i. Now:

$$|g(x)_i - g(y)_i| = |\max_j\{f^j(x)_i\} - \max_j\{f^j(y)_i\}|.$$

There are two cases: in the first, the f^j that attains the maximum is the same for x and y. Then we are done, since every f^j is a γ-contraction. Else, we can rewrite $|g(x)_i - g(y)_i|$ as $|f^j(x)_i - f^k(y)_i|$ for specific indices $j, k \in J$. Since then $f^k(x)_i - f^j(x)_i \leq 0$ and $f^k(y)_i - f^j(y)_i \geq 0$, the intermediate value theorem yields a zero of the function $\theta \mapsto f^j(\theta x + (1-\theta)y)_i - f^k(\theta x + (1-\theta)y)_i$, mapping $[0, 1]$ to $\mathbb{R}$. Let θ be said zero: Then

$$\begin{aligned}
&|f^j(x)_i - f^k(y)_i| = \\
&|f^j(x)_i - f^j(\theta x + (1-\theta)y)_i + f^k(\theta x + (1-\theta)y)_i - f^k(y)_i| \\
&\leq |f^j(x)_i - f^j(\theta x + (1-\theta)y)_i| + |f^k(\theta x + (1-\theta)y)_i - f^k(y)_i| \\
&\leq \gamma\|x - \theta x - (1-\theta)y\|_\infty + \gamma\|\theta x + (1-\theta)y - y\|_\infty \\
&= \gamma(1-\theta)\|x - y\|_\infty + \gamma\theta\|x - y\|_\infty \\
&= \gamma\|x - y\|_\infty
\end{aligned}$$

and we are done. □

Correspondingly, by lemma 3.2.3, T is a γ-contraction with $Q^*_\Pi|_{S_{learn}}$ as unique fixed point, which proves the theorem.

In particular, if the (or one of the) globally optimal deterministic policy for the base MDP restricted to S_{fix}, $\pi^*|_{S_{fix}}$ is in Π, the policy of always selecting the action $\pi^*|_{S_{fix}}$ in S_{fix} while following $\pi^*|_{S_{learn}}$ in S_{learn} is optimal in $M|_\Pi$: because of the Markov property, choosing different actions for the same state s in S_{fix} at different visitations can not be strictly better in M than always taking the same optimal action $a^*(s) \in A$. This argument clearly extends to (partial) policies, such that there can be no benefit from choosing another policy over $\pi^*|_{S_{fix}}$ for any visit to S_{fix} in $M|_\Pi$. With always behaving like $\pi^*|_{S_{fix}}$ between entering and leaving S_{fix} and its lifted versions, one optimal policy on $M|_\Pi$ restricted to S_{learn} is clearly equal to $\pi^*|_{S_{learn}}$. Therefore

$$Q^*(s,a) = Q^*|_{\pi^*|_{S_{fix}}}(s,a) = Q^*|_\Pi(s,a)$$

for $s \in S_{learn}, a \in A$ as long as $\pi^*|_{S_{fix}} \in \Pi$ with $Q^*|_\pi = Q^*|_{\{\pi\}}$ as in the previous section.

Because $\tilde{T_1}\begin{pmatrix}Q\\V\end{pmatrix} \geq \tilde{T_1}\begin{pmatrix}Q'\\V'\end{pmatrix}$ for $\begin{pmatrix}Q\\V\end{pmatrix} \geq \begin{pmatrix}Q'\\V'\end{pmatrix}$, in both cases pointwise and because adding policies to Π can only increase the outputs of $\tilde{T_2}$, we also get that

$$Q^*|_\Pi(s,a) \geq Q^*|_{\Pi'}(s,a)$$

whenever $\Pi' \subset \Pi, s \in S_{learn}, a \in A$, for any choice of Π. In particular, this implies that

$$Q^*(s,a) = Q^*|_{\Pi \cup \pi^*|_{S_{fix}}}(s,a) \geq Q^*|_\Pi(s,a),$$

the obtained Q values are thus bounded from above by the correct Q^* for the globally optimal policy on $S_{fix} \cup S_{learn}$ in the base MDP M. This means that having access to more experts helping with the decision process cannot lead to worse decisions and that having an expert whose local solution is part of the globally optimal solution enables us to converge to the global optimum. There is another way to relate the obtained policy to policies in the base MDP M: there exists a policy π^*_Π on M such that π^*_Π restricted to S_{learn} is the greedy policy induced by $Q^*|_\Pi$ and we have that

$$Q_{\pi^*_\Pi}(s,a) \geq Q^*|_\Pi(s,a).$$

To see this, we note that $Q^*|_\Pi(s,a)$ is equal to the return after taking action a in state s in M and afterwards following the greedy policy induced by $Q^*|_\Pi(s,a)$ in S_{learn} while choosing the policy in Π that maximizes $(I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q)(s)$ at each transition to $s \in S_{fix}$. With the roles of S_{fix} and S_{learn} reversed, there exists an optimal response in the sense of the last chapter to the policy induced on S_{learn} by $Q^*|_\Pi(s,a)$ on S_{fix} in M, which yields π^*_Π when combined with the greedy policy on S_{learn}: Because of the Markov property, this optimal response is at least as good regarding returns as any of the potentially time-inconsistent policies on M induced by operating on $M|_\Pi$. In particular, we get

$$Q_{\pi^*_\Pi}(s,a) \geq Q^*|_\Pi(s,a),$$

as these Q-Values just represent the respective expected returns for the greedy policy induced by $Q^*|_\Pi$ in $M|_\pi$ and π^*_Π in M. This also illustrates that the process can be iterated to obtain a globally optimal policy: switching the roles of S_{fix} and S_{learn} again, we can find and optimal response to $\pi^*_\Pi|_{S_{fix}}$ and iterate. Each of these iterations will improve the global policy and once there is no more improvement, the globally optimal policy is reached. This is because by the policy improvement theorem 2.2.8, a policy π is not optimal, if and only if there exists a state s in which its value can be improved (by changing $\pi(s)$). But such a state has to either be in S_{fix} or in S_{learn}, such that an optimal response in the corresponding region would improve the policy for that state. Thus, if there is no improvement after a full cycle of optimal responses, the global policy is optimal.

4 Learning from Simulations

A simulation of an MDP is a mechanism to sample transitions ("experience") for that MDP. In practice, there is usually a termination condition to avoid infinite (looping) trajectories: either terminal states that only allow for rewardless transitions to themselves are flagged and the simulation restarted after encountering them, or a maximal length for an episode (unit of interaction with the environment before it is restarted) is specified. Learning from simulations of an MDP without knowing explicit transition probabilities can be a very powerful tool, since it is often a lot cheaper and easier to sample transitions than to precisely write down the transition probabilities. For example, a toddler is able to draw a hand of cards, while calculating the probability for each hand to be drawn eludes a lot of high school students. Similarly, even if the stock market fulfilled the Markov property, it would seem impossible to give explict probabilities for the effects selling a bunch of stocks would have on the wider market. Provided we are able to measure all relevant effects, the generation of a sample trajectory is trivial, though.

4.1 Policy evaluation

The goal of policy evaluation is to approximate the value functions Q_π or V_π for a given policy π. There are two main types of methods to do this based on sample trajectories. The first type, Monte Carlo methods use that

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{k=0} \gamma^k r_k | s_0{=}s]$$

and

$$Q_\pi(s,a) := \mathbb{E}_\pi[\sum_{k=0} \gamma^k r_k | s_0{=}s, a_0 = a]$$

to approximate the value functions by sampling trajectories and averaging over the respective returns. For more information about these in the context of reinforcement learning, see [1]. The main problem with this approach is that the returns can have very large variance for long trajectories, such that it can take a long time to get a good approximation of the value functions. Instead, we can try to leverage that both value functions fulfill Bellman equations and will use ideas from dynamic programming in order to iteratively improve an arbitrary initial approximation based on sampled transitions. These methods are called temporal difference methods. With the updates only depending on single transitions, instead of full trajectories their variance is rather small. This comes at the cost of some bias, though: While Monte Carlo methods have the correct expectation, no matter how many trajectories are used, the estimates made by temporal difference methods depend on the previous (usually faulty) estimate and are thus biased towards the initialization, at least for the first few episodes.

4.1.1 SARSA

SARSA (state, action, reward, state, action) is a classic temporal difference methods based on the Bellman equation 2.2.2 for Q_π:

$$Q_\pi(s,a) = \sum_{s'\in S} p(s'|s,a)(r(s',a,s)+\gamma \sum_{a'\in A} \pi(s',a')Q_\pi(s',a')) = \mathbb{E}_{\substack{s'\sim p(s'|s,a)\\ a'\sim\pi(a'|s')}}[r(s',a,s)+\gamma Q_\pi(s',a')] :$$

Q is initialized arbitrarily for any $s_0 \in S$, actions a_t are carried out according to $\pi(a_t|s_t)$ and states s_{t+1} generated according to $p(s_{t+1}|s_t,a_t)$, until a terminal state is reached and the episode ends. At this point, the simulation is restarted with a new s_0. Then, with $r_t = r(s_{t+1},a_t,s_t)$, Q is updated at every simulation step t by

$$Q(s_t,a_t) = (1-\alpha_t)Q(s_t,a_t) + \alpha_t(r_t + \gamma Q(s_{t+1},a_{t+1}))$$

for some step size α_t, with $Q(s_{t+1},a_{t+1})$ fixed to zero for terminal states.

Convergence to Q_π obviously requires some restrictions on the step sizes and that every state action pair is actually visited (sufficiently often). In the next section we will prove, that with χ_A denoting the characteristic function of a set A, the conditions

$$\sum_{t=0}^{\infty} \alpha_t \chi_{\{(s_t,a_t)=(s,a)\}} = \infty$$

and

$$\sum_{t=0}^{\infty} \alpha_t^2 \chi_{\{(s_t,a_t)=(s,a)\}} < C < \infty$$

for all $s \in S, a \in A$ with probability one are sufficient for convergence with probability one. The first condition guarantees, that the updates are large enough to overcome bad initial or intermittent values, while the second prevents the steps from being so large that already obtained information is drowned out by the noise of the updates. The simplest way to achieve this is to initialize the step count $v(s,a) := 0$ for all $s,a \in S,A$ at the beginning and update $v(s_t,a_t) = v(s_t,a_t)+1$ at every step t. Then with $\alpha_t = \frac{1}{v(s_t,a_t)+1}$, the conditions hold, as long as every state-action pair appears infinitely often:

Lemma 4.1.1. *Let π be a policy with $\pi(a|s) > 0$ for all $a, s \in S, A$ and let M be an MDP such that for every pair of states $s, s' \in S$ there exists a finite sequence of actions that leads from s to s' with probability bigger than zero (assuming transitions from terminal states according to the start state distribution p_0): Then for all $(s,a) \in S \times A$, $(s_t, a_t) = (s,a)$ for infinitely many $t \in \mathbb{N}$ with probability one.*

Proof. We first prove that for every pair of state action pairs $(s,a), (s',a') \in S \times A$ there exists a $n \in \mathbb{N}$ such that (s',a') will be reached from (s,a) after at most n steps with probability $q > 0$: Let s'' be any state with $p(s''|s,a) > 0$. Then with positive probability, the next state is s'', conditioned on which we can find a finite sequence of actions (whose length we call $n-1$) leading us to s' with positive probability. This is because by assumption and the Markov property, all finite sequences of actions have positive probability. So we will end up in state s' after n steps with positive probability whereupon action a' is taken, again with positive probability.

We continue by showing that every state-action-pair will be visited with probability one: Since our MDP is finite, there is a minimal $q > 0$ and a maximal $n \in \mathbb{N}$ with n and q as above. So for this combination of n and q and a fixed state-action-pair (s',a') no matter in which state-action-pair (s,a) we start, we have a probability of at least q to have visited (s',a') after n steps, a probability of at least $q + (1-q)q$ after $2n$ steps (since we are at least as likely to visit as we would be with a $n-$step visit probability of q) and so on. Since the events $A_k := \{(s',a') \text{ is visited after } k \cdot n \text{ steps}\}$ form an ascending sequence,

$$P((s',a') \text{ is visited after } t \text{ steps for any } t \in \mathbb{N}) = \lim_{n\to\infty} P(A_k) \geq q \sum_{k=0}^{\infty} (1-q)^k = \frac{q}{1-(1-q)} = 1.$$

Next, we show that every state will be visited infinitely often with probability one. For fixed (s,a) we get that for any for any real $t < \infty$

$$\begin{aligned}
&P(\sup_{r\geq 0}\{(s_r,a_r) = (s,a)\} = t) \\
&= P((s_t,a_t) = (s,a), (s_r,a_r) \neq (s,a) \text{ for } r > t) \\
&= P((s_r,a_r) \neq (s,a) \text{ for } r > t | (s_t,a_t) = (s,a)) P((s_t,a_t) = (s,a)) \\
&= P((s_r,a_r) \neq (s,a) \text{ for } r > 0 | (s_0,a_0) = (s,a)) P((s_t,a_t) = (s,a)) \\
&= 0
\end{aligned}$$

using that transitions do not depend on the history and that the first term was already proven to be zero. Since the supremum is $-\infty$ with probability zero because every state will be visited with probability one, we get that $(\sup_{r\geq 0}\{(s_r,a_r) = (s,a)\} = t) = \infty$ with probability one. In particular, this means that (s,a) is visited infinitely often. $\square$

Clearly, that won't work for deterministic policies, as most state-action pairs won't be visited at all. Following an $\varepsilon-$softened version of the policy π that acts according to π with probability $1-\varepsilon$ and (uniformly) randomly with probability ε can help with getting estimates for normally unvisited state-action pairs: Now, as long as the MDP is connected in the sense of every state being reachable from every other state, all state-action pairs will be visited infinitely often: By adjusting the update to

$$Q(s_t, a_t) = (1-\alpha_t)Q(s_t, a_t) + \alpha_t(r_t + \gamma Q(s_{t+1}, a'_{t+1}))$$

with $a'_{t+1} = a_{t+1}$ in case the action at time $t+1$ is according to the policy and $a'_{t+1} \sim \pi(a_{t+1}|s_{t+1})$ in case it was uniformly random, the expectation of the target is still correct, while we ensure that diverse actions are taken. This works because as long as the conditions on α_t hold and state action pairs are updated infinitely often, the precise order of updates to state-action pairs does not matter for the ultimate convergence of SARSA. This modification turns SARSA from an on-policy algorithm that learns about the policy that is actually carried out into an off-policy algorithm that learns about a policy while actually carrying out another one. Orseau and Armstrong employ a very similar idea in [13] to train agents that lack an incentive for avoiding to be shut off.

After adjusting the update, another problem is that the conditions don't have to work if one state is not reachable from other states at all. In episodic tasks where a terminal state is usually encountered quickly, a simple remedy can be provided by using so called exploring starts: s_0 is sampled randomly from S for each new episode, or in other words, p_0 is (artificially) set to the uniform distribution on S. But even with a determinstic start state, we will still visit some state-action pairs infinitely often: either every state is reachable from every other state (given our softened policy), or we will eventually enter a subset B of S, for which this is true, but no state outside B can be reached from B. If B is unique, we are guaranteed to visit every state-action pair (s, a) with $s \in B$ infinitely often by the same argument as above. If states also stay in B with probability one, we even get correct values for all states in B, since the value of a state s is not influenced by value estimates of states s' that are not reachable from s.

In practice, this problem can sometimes be circumvented by restarting the simulation (artificially setting the state to the start state, while skipping an update) regularly: If we restart every n steps, the argument from lemma 4.1.1 goes trough again for all state-action pairs (s, a) such that s can be reached from the start state within n steps with positive probability. Since every state that can be reached from the start state eventually (with positive probability) must also be reachable with positive probability within n steps for some $n \in \mathbb{N}$ and the state space is finite, there is some $N \leq |S|$ such that every state that is reachable at all from the start can be reached within N steps. Thus resetting every N steps ensures that all of those states will be visited infinitely often, using the same argument as in lemma 4.1.1. Because states that are not reachable from another state cannot influence that state's Q-values, this means that in this setting we will converge to the correct Q_π for all states that are reachable from the start.

Algorithm 1 Soft SARSA for approximating Q_π

Input: MDP: M, Number of episodes T, policy π, episode length N, $\varepsilon > 0$, discount rate γ, step size function α depending on time and visit count.
Output: Approximate values for Q_π

```
Initialize $Q(s,a) = v(s,a) = 0$ for all $s, a \in S, A$
for $e = 0$ to T do
    Sample $s' \sim p_0$
    Done = False
    $t = 0$
    while Done is False and $t \leq N$ do
        resample = False
        Sample $u \sim U((0,1))$
        if $u > \varepsilon$ then
            Sample $a' \sim \pi(a'|s')$
        else
            Sample $a' \sim U(A)$
            resample = True
        end if
        if $t > 0$ then
            if resample is True then
                $a'' \sim \pi(a''|s')$
            else
                $a'' = a'$
            end if
            $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a))(r + \gamma Q(s', a''))$
            $v(s,a) = v(s,a) + 1$
        end if
        $s = s'$
        $a = a'$
        Sample $s' \sim p(s'|s,a)$
        $r = r(s', a, s)$
        if $s'$ is terminal then
            $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a)) \cdot r$
            $v(s,a) = v(s,a) + 1$
            Done = True
        end if
        $t = t + 1$
    end while
end for
```

With infinite time and correctly decreasing step sizes as well as correctly placed restarts this converges to Q_π with probability one. The classical on-policy SARSA algorithm (as described in [1]) is recovered with $\varepsilon = 0$. In practice, we cannot run infinite episodes, so that true convergence won't happen. Furthermore, oftentimes a constant step size is used to simplify the algorithm and potentially speed up early learning, at the cost of formal guarantees.

Theorem 4.1.2. *For an MDP with state space S and a policy π, the Q-values computed by soft SARSA (algorithm 1) converge to Q_π with probability one, as long as $N \geq |S|$, $\varepsilon > 0$, $\alpha(t, v(s,a)) = \frac{1}{v(s,a)+1}$, $T = \infty$ and for all states $s' \in S$ there exists a policy such that s' can be reached from at least one start state (a state s with $p_0(s) > 0$) with positive probability.*

Proof. This follows directly from the discussion above and theorem 4.2.1, the main theorem of the next section by considering the operator defined by

$$TQ(s,a) := E_{\substack{s'\sim p(s',a)\\ a'\sim\pi(s',a')}}[r(s',a,s)+\gamma Q(s',a')]$$

for all $s,a\in S,A$, which is a contraction with Q_π as fixed point by theorem 2.3.4, with (s,a) playing the role of the index i in the theorem, as well as the sampled version defined by

$$\tilde{T}Q(s,a) = r(s',a,s)+\gamma Q(s',a')$$

for all $s,a\in S,A$ with s' and a' sampled independently for each (s,a) from $p(s'|s,a)$ and $\pi(a'|s')$. $\tilde{T}$ is a contraction, independent of the sampled values and we obviously have

$$\mathbb{E}[\tilde{T}Q] = TQ$$

and

$$\|\tilde{T}Q_\pi - TQ_\pi\|_\infty < K < \infty,$$

for some constant $K\in\mathbb{R}$ since there are only finitely many transitions that can be sampled. $\square$

4.1.2 Td(0)

A very similar classical approach, called td(0) works for V_π using the respective Bellman equation 2.2.1

$$V_\pi(s) = \sum_{a\in A}\pi(a|s)\sum_{s'\in S}p(s'|s,a)(r(s',a,s)+\gamma V_\pi(s')).$$

Again we need to follow an ε-soft version of the original policy π in order to guarantee infinite visits of every state. The classical *td*(0) algorithm described in [1] is recovered for $\varepsilon = 0$

Theorem 4.1.3. *For an MDP with state space S and a policy π, the V-values computed by td(0) converge to V_π with probability one, as long as $N\geq|S|$, $1>\varepsilon>0$, $\alpha(t,v(s,a)) = \frac{1}{v(s,a)+1}$, $T=\infty$ and for all states $s'\in S$ there exists a policy such that s' can be reached from at least one start state (a state s with $p_0(s)>0$) with positive probability. If the last condition holds for π and all states, ε can be set to zero.*

Proof. The proof it analogous to the proof of theorem 4.1.2. This time, we apply theorem 4.2.1 to

$$TV(s) := \mathbb{E}_{\substack{a\sim\pi(a|s)\\ s'\sim p(s'|s,a)}}[r(s',a,s)+\gamma V(s')]$$

and

$$\tilde{T}V(s) := r(s',a,s)+\gamma V(s')$$

for which we immediately see

$$\mathbb{E}[\tilde{T}V] = TV$$

Algorithm 2 Soft td(0) for approximating V_π

Input: MDP: M, Number of episodes T, policy π, episode length N, $\varepsilon > 0$, discount rate γ, step size function α depending on time and visit count.
Output: Approximate values for V_π

```
Initialize $V(s) = v(s) = 0$ for all $s \in S$
resample=True
for $e = 0$ to T do
  Sample $s' \sim p_0$
  Done = False
  $t = 0$
  while Done is False and $t \leq N$ do
    if resample is False then
      $V(s) = (1 - \alpha(t, v(s,a))V(s) + \alpha(t, v(s))(r + \gamma V(s'))$
      $v(s) = v(s) + 1$
    else
      resample = False
    end if
    Sample $u \sim U((0,1))$
    if $u > \varepsilon$ then
      Sample $a' \sim \pi(a'|s')$
    else
      Sample $a' \sim U(A)$
      resample = True
    end if
    $s = s'$
    $a = a'$
    Sample $s' \sim p(s'|s,a)$
    $r = r(s', a, s)$
    if $s'$ is terminal then
      $V(s) = (1 - \alpha(t, v(s))V(s) + \alpha(t, v(s,a)) \cdot r$
      $v(s) = v(s) + 1$
      Done = True
    end if
    $t = t + 1$
  end while
end for
```

as well as

$$\|\tilde{T}V_\pi - TV_\pi\|_\infty \leq K.$$

The modified conditions on ε and the reachability of states come from the fact, that it is only necessary to updated infinitely often in each state for approximating V, such that we don't need to ensure that every state-action pair is taken. □

On the other hand, we are not able to reuse transitions produced by ε-exploration in this algorithm, since we are unable to correct the action, as the V-function is action-agnostic. This means that ϵ should be fairly small to ensure efficiency.

4.2 Why Does it Work?

In order to see that theorem 4.1.2 holds, we need a way to generalize the reasoning based on contractions and fixed point we previously used to prove the convergence for dynamic programming to sampling based methods. We will closely follow [3] to prove the following theorem:

Theorem 4.2.1. *Let T be a γ-contraction in the maximum norm on $\mathbb{R}^d$ for $0 \leq \gamma < 1$ with fixed point Q^* and $Q_0 \in \mathbb{R}^d$ be arbitrary. Let α_t be a random sequence in $[0,1]^d$ and i_t be a random sequence of natural numbers smaller than d such that for all $i \leq d$*

$$\sum_{t\in\mathbb{N}} \alpha_t(i)\chi_{\{i_t=i\}} = \infty$$

and there exists a constant C such that

$$\sum_{t\in\mathbb{N}} \alpha_t^2(i)\chi_{\{i_t=i\}} < C < \infty,$$

both with probability one. Define

$$T_t(W,Q)(i) = \begin{cases} (1-\alpha_t(i))W(i) + \alpha_t(i)\tilde{T}_tQ(i) & \text{if } i_t = i \\ W(i), & \text{else} \end{cases}$$

and let $\tilde{T}_t$ be random $\gamma-$sup-norm contractions on $\mathbb{R}^d$ that are also independent of the history of the process, $\{i_t, ..., i_1, \alpha_t, ..., \alpha_1, \tilde{T}_{t-1}, ..., \tilde{T}_1\}$ and with

$$\mathbb{E}[\tilde{T}_tQ] = TQ$$

and

$$\|\tilde{T}_tQ^* - TQ^*\|_\infty < K < \infty.$$

Then, the iteration

$$Q_{t+1}(i) = T_t(Q_t, Q_t)$$

converges to Q^* with probability one.

For SARSA, the $\tilde{T}_t$ would be independent copies of the sampling operator $\tilde{T}$, that also don't depend on the past in any other way. In order to prove the theorem, we start with decoupling the learning for different states. With the following theorem from [3], it will be enough to show that we have convergence componentwise and with $\alpha_t(i)\tilde{T}_t Q(i)$ replaced by $\alpha_t(i)\tilde{T}_t Q^*(i)$, which is independent of what we have previously learned, in particular for other states.

Definition 4.2.2. *For a map T $\mathbb{R}^d \to \mathbb{R}^d$, we say that a sequence of random operators $T_t : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ approximates T at $Q \in \mathbb{R}^d$ with probability one, if the sequence $(W_t)_{t\in\mathbb{N}}$ recursively defined by $W_{t+1} = T_t(W_t, Q)$ converges to TQ with probability one for every $W_0 \in \mathbb{R}^d$.*

Theorem 4.2.3. *Let T be a γ-contraction in the maximum norm on $\mathbb{R}^d$ with fixed point Q^*. Let T_t approximate T at Q^* with probability one. Set $Q_0 \in \mathbb{R}^d$ arbitrarily and define $Q_{t+1} := T_t(Q_t, Q_t)$. Let $f_t(x)$ and $g_t(x)$ be random sequences in $[0,1]^n$. Then, Q_t converges to Q^*, if with probability one:*

1) $|T_t(W_1, Q^*)(i) - T_t(W_2, Q^*)(i)| \le g_t(i)|W_1(i) - W_2(i)|$ *for all* $i \le d$ *and* $W_1, W_2 \in \mathbb{R}^d$
2) $|T_t(W, Q^*)(i) - T_t(W, Q)(i)| \le f_t(i)\|Q^* - Q\|_\infty$ *for all* $i \le d$ *and* $W, Q \in \mathbb{R}^d$
3) $\lim_{n\to\infty} \prod_{t=k}^{n} g_t(i) = 0$ *for all* $k \in \mathbb{N}, i \le d$
4) $f_t(i) \le \gamma(1 - g_t(i))$ *for all* $i \le d$.

Proof. Let W_0 be arbitrary and define $W_{t+1} = T_t(W_t, Q^*)$. By assumption: W_t converges to Q^* with probability one. Set

$$\delta_t(i) = |W_t(i) - Q_t(i)|$$

and

$$\Delta_t(i) = |W_t(i) - Q^*(i)|.$$

Then:

$$\begin{aligned}
\delta_{t+1}(i) &= |W_{t+1}(i) - Q_{t+1}(i)| \\
&= |T_t(W_t, Q^*)(i) - T_t(Q_t, Q_t)(i)| \\
&\le |T_t(W_t, Q^*)(i) - T_t(Q_t, Q^*)(i)| + |T_t(Q_t, Q^*)(i) - T_t(Q_t, Q_t)(i)| \\
&\le g_t(i)|W_t(i) - Q_t(i)| + f_t(i)\|Q^* - Q_t\|_\infty \\
&= g_t(i)\delta_t(i) + f_t(i)\|Q^* - Q_t\|_\infty \\
&\le g_t(i)\delta_t(i) + f_t(i)(\|Q^* - W_t\|_\infty + \|W_t - Q_t\|_\infty) \\
&\le g_t(i)\delta_t(i) + f_t(i)(\|\Delta_t\|_\infty + \|\delta_t\|_\infty)
\end{aligned}$$

Intuitively, the process $\tilde{\delta}_{t+1} = g_t(i)\tilde{\delta}_t(i) + f_t(i)(\|\tilde{\Delta}_t\|_\infty + \|\tilde{\delta}_t\|_\infty)$ for $\tilde{\delta}_0 = \delta_0$ converges to zero: If $g_t(i)$ was smaller than some constant $\mu < 1$ infinitely often, and the noise term Δ_t was constant zero, the norm of $\tilde{\delta}_{t+1}$ gets contracted by a factor of $\gamma + (1-\gamma)g_t(i)$ at every step, which would be smaller than the constant $\gamma + (1-\gamma)\mu$ infinitely often and always at most 1. Thus, we would get convergence to zero. However, the argument is complicated by the fact that the noise term only converges to zero and that $\lim_{n\to\infty} \prod_{t=k}^{n} g_t(i) = 0$ for all $k \in \mathbb{N}, i \leq d$ does not imply $g_t(i) < \mu$ for some μ infinitely often. The convergence of $\tilde{\delta}_t$ to zero will be proven in full rigor in lemma A.0.7 in the appendix, with ϵ_t playing the role of $\|\Delta_t\|_\infty$. Then, δ_t is always positive and inductively bounded by $\tilde{\delta}_t$ and thus converges to zero as well. Therefore, Q_t and W_t have the same limit, Q^*. $\square$

In order to prove theorem 4.2.1 we need to show that T_t approximate T at V^* with probability one and check off all the other conditions. We will start by formulating a theorem that helps us showing that a sequence of operators approximates another operator with probability one: This is a modified version of a fairly standard result in stochastic approximation proven by Blum in [5] that allows for stochastic α_t, as long as they do not influence future sampling, under slightly stronger conditions than in the original theorem.

Theorem 4.2.4. *Let $P(y|x)$ be probability measures on $\mathbb{R}$ for every $x \in \mathbb{R}$ with corresponding expectations $M(x) := \int_{-\infty}^{\infty} y \; dP(y|x) = x - \theta$ for some constant $\theta \in \mathbb{R}$. Then for a sequence of random variables in $[0,1]$: $(\alpha_n)_{n\in\mathbb{N}}$, the iteration*

$$x_{n+1} = x_n - \alpha_n y_n$$

with y_n sampled from $P(y_n|x_n)$ converges to θ with probability one for any x_1, as long as there exists a constant C such that

$$\sum_{t\in\mathbb{N}} \alpha_t^2 < C < \infty$$

and

$$\sum_{t\in\mathbb{N}} \alpha_t = \infty,$$

both with probability one and a constant $K \in \mathbb{R}$ such that $|y_n - M(x_n)| \leq K < \infty$ with probability one for all $n \in \mathbb{N}$ and $x_n \in \mathbb{R}$.

We will first prove that theorem 4.2.1 follows from theorem 4.2.3 and theorem 4.2.4 and will prove theorem 4.2.4 in the appendix.

Proof. We need to show that the iteration $Q_{t+1}(i) = T_t(Q_t, Q_t)(i)$ for

$$T_t(W,Q)(i) = (1-\alpha_t(i)\chi_{\{i_t=i\}})W(i) + \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q(i) = \begin{cases} (1-\alpha_t(i))W(i) + \alpha_t(i)\tilde{T}_t Q(i) \text{ if } i_t = i \\ W(i), \text{ else} \end{cases}$$

converges to Q^* with probability one. In order to do this, we will apply theorem 4.2.3 for $g_t(i) = 1 - \alpha_t(i)\chi_{\{i_t=i\}}$ and $f_t(i) = \gamma\alpha_t(i)\chi_{\{i_t=i\}}$ We will begin by proving the numbered conditions.

1) Let $W_1, W_2 \in \mathbb{R}^d$ and $i \leq d$ be arbitrary. Then we have

$$\begin{aligned}
&|T_t(W_1, Q^*)(i) - T_t(W_2, Q^*)(i)| \\
&= |(1 - \alpha_t(i)\chi_{\{i_t=i\}})W_1(i) + \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q^*(i) - (1 - \alpha_t(i)\chi_{\{i_t=i\}})W_2(i) - \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q^*(i)| \\
&= (1 - \alpha_t(i)\chi_{\{i_t=i\}})|W_1(i) - W_2(i)| \\
&= g_t(i)|W_1(i) - W_2(i)|
\end{aligned}$$

2) Let $U, V \in \mathbb{R}^d$ and $i \leq d$ be arbitrary. Then we have

$$\begin{aligned}
&|T_t(W, Q^*)(i) - T_t(W, Q)(i)| \\
&= |(1 - \alpha_t(i)\chi_{\{i_t=i\}})W(i) + \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q^*(i) - (1 - \alpha_t(i)\chi_{\{i_t=i\}})W(i) - \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q(i)| \\
&= |\alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q^*(i) - \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_t Q(i)| \\
&\leq \alpha_t(i)\chi_{\{i_t=i\}}\|\tilde{T}_t Q^* - \tilde{T}_t Q\|_\infty \\
&\leq \gamma\alpha_t(i)\chi_{\{i_t=i\}}\|Q^* - Q\|_\infty \\
&= f_t(i)\|Q^* - Q\|_\infty,
\end{aligned}$$

Where we used that $\tilde{T}_t$ is a γ-contraction.

3) Let $k \in \mathbb{N}, i \leq d$ be arbitrary. Then

$$\lim_{n\to\infty} \prod_{t=k}^{n} g_t(i) = \lim_{n\to\infty} \prod_{t=k}^{n} (1 - \alpha_t(i)\chi_{\{i_t=i\}}).$$

This converges to zero, if and only if its logarithm that is the same as

$$\sum_{t=k}^{\infty} \log(1 - \alpha_t(i)\chi_{\{i_t=i\}})$$

converges to $-\infty$. But this follows from $\sum_{t\in\mathbb{N}} \alpha_t(i)\chi_{\{i_t=i\}} = \infty$ and thus $\sum_{t=k}^{\infty} \alpha_t(i)\chi_{\{i_t=i\}} = \infty$, which we have with probability one, together with $\log(1 - x) \leq -x$ for $x \in (-\infty, 1)$.

4) $f_t(i) = \gamma\alpha_t(i)\chi_{\{i_t=i\}} = \gamma(1 - 1 + \alpha_t(i)\chi_{\{i_t=i\}}) = \gamma(1 - g_t(i))$

Now, the only thing left is to show that T_t approximates T at Q^* with probability one. In other words, we need to show that

$$W_{t+1}(i) = T_t(W_t, Q^*)(i)$$

converges to $TQ^*(i)$ with probability one for arbitrary start values $W_1 \in \mathbb{R}^d$ and all $i \leq d$. We rewrite

$$T_t(W_t, Q^*)(i)$$

$$= (1 - \alpha_t(i)\chi_{\{i_t=i\}})W_t(i) + \alpha_t(i)\chi_{\{i_t=i\}}\tilde{T}_tQ^*(i)$$
$$= W_t(i) - \alpha_t(i)\chi_{\{i_t=i\}}(W_t(i) - \tilde{T}_tQ^*(i))$$

and check the conditions for theorem 4.2.4 with $y_t = W_t(i) - \tilde{T}_tQ^*(i)$, $x_t = W_t(i)$, $\alpha_t = \alpha_t(i)\chi_{\{i_t=i\}}$ and $\theta = Q^*(i)$: We get that the α_t fulfill the sum and bound conditions imposed on them in the theorem and that

$$M(x) = \mathbb{E}[(W_t(i) - \tilde{T}_tQ^*(i))|W_t(i) = x] = x - TQ^*(i) = x - Q^*(i)$$

since for all $Q : \mathbb{E}[\tilde{T}_tQ] = TQ$ and since $\tilde{T}_t$ is independent of the history $h_t := \{i_t, ..., i_1, \alpha_t, ..., \alpha_1, \tilde{T}_{t-1}, ..., \tilde{T}_1\}$, of which $W_t(i)$ is a measurable function. This independence also means that $y_t = W_t(i) - \tilde{T}_tQ^*(i)$ is independent from the history given $x_t = W_t$, or in other words that y_t is sampled from a distribution parameterized by x_t. Finally, since

$$\|\tilde{T}_tQ^* - TQ^*\|_\infty < K < \infty$$

with probability one, we get that

$$|y_t - M(x_t)| = |W_t(i) - \tilde{T}Q^*(i) - W_t(i) + TQ^*(i)| = |TQ^*(i) - \tilde{T}Q^*(i)| \leq K < \infty$$

with probability one and can apply theorem 4.2.4 to get that W_t converges to Q^* with probability one and T_t thus approximates T at Q^* with probability one. □

4.3 Q-learning

We can also use theorem 4.2.1 to easily prove the convergence of Q-learning [1], a popular off-policy algorithm that is known to converge and approximates the optimal action value function Q^* using an approach similar to SARSA. We start with the Bellman equation

$$Q^*(s,a) = \sum_{s' \in S} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q^*(s',a')) = E_{s' \sim p(s',a)}[r(s',a,s) + \max_{a' \in A} \gamma Q^*(s',a')]$$

and replace the Bellman operator

$$TQ(s,a) = \mathbb{E}[r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')],$$

which is a γ-contraction by theorem 2.3.5 with a sampled version

$$\tilde{T}Q(s,a) = r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')$$

for all $s, a \in S, A$ with s' sampled independently for each (s,a) from $p(s'|s,a)$. This is again a contraction independent of the sampled values by argument in theorem 2.3.5, as this it is just the Bellman optimality operator for a deterministic MDP with transitions equal to the sampled ones. It is again obvious that

$$\mathbb{E}[\tilde{T}Q] = TQ$$

and

$$\|\tilde{T}Q^* - TQ^*\|_\infty < K < \infty,$$

since there are only finitely many possible transitions. This means that the update rule

$$Q(s_t, a_t) = (1 - \alpha_t)Q(s_t, a_t) + \alpha_t(r_t + \gamma \max_{a' \in A} Q(s_{t+1}, a'))$$

for some step size α_t and $Q(s_{t+1}, a')$ fixed to zero for terminal states will lead to convergence to Q^* for arbitrary start values, as long as all state-action pairs are visited infinitely often and we have that

$$\sum_{t=0}^{\infty} \alpha_t \chi_{\{(s_t,a_t)=(s,a)\}} = \infty$$

and

$$\sum_{t=0}^{\infty} \alpha_t^2 \chi_{\{(s_t,a_t)=(s,a)\}} < C < \infty$$

for all $s \in S, a \in A$ with probability one. If all states are reachable from the start state, $\alpha_t = \frac{1}{v(s,a)+1}$ works, as long as actions are generated by an ε-soft policy that has a probability of at least ε for every action to be taken in every state for some $\varepsilon > 0$. For $Q-$learning, the most natural choice is an $\varepsilon-$greedy policy, that takes $\operatorname{argmax}_{a \in A} Q(s, a)$ in state s for the current estimate of the $Q-$function with probability $1 - \varepsilon$ and a (uniformly) random action else. This ensures convergence for all state-action pairs while still preferably visiting states with high value estimates, such that the optimal policy can usually be found long before convergence of all Q-values.

Theorem 4.3.1. *For an MDP with state space S, the Q-values computed by Q-learning (algorithm 3) converge to Q^* with probability one, as long as $N \geq |S|$, $\varepsilon > 0$, $\alpha(t, v(s, a)) = \frac{1}{v(s,a)+1}$, $T = \infty$ and for all states $s' \in S$ there exists a policy such that s' can be reached from at least one start state (a state s with $p_0(s) > 0$) with positive probability.*

Proof. This follows directly from the argument in lemma 4.1.1 and the subsequent discussion of restarts, as well as theorem 4.2.1, as in theorem 4.1.2. □

Interestingly, V^* does not allow for a similarly easy algorithm, since approximating the maximum in the Bellman equation 2.2.3:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} p(s'|s, a)(r(s', a, s) + \gamma V^*(s'))$$

without bias would require a sampled transition for all actions $a \in A$.

Algorithm 3 Q-learning for approximating Q^*

Input: MDP: M, Number of episodes T, episode length N, $\varepsilon > 0$, discount rate γ, step size function α depending on time and visit count.

Output: Approximate values for Q^*

```
Initialize $Q(s,a) = v(s,a) = 0$ for all $s, a \in S, A$
for $e = 0$ to T do
    Sample $s' \sim p_0$
    Done = False
    $t = 0$
    while Done is False and $t \leq N$ do
        Sample $u \sim U((0,1))$
        if $u > \varepsilon$ then
            $a' = \text{argmax}_{a' \in A} Q(s', a')$ with ties broken arbitrarily
        else
            Sample $a' \sim U(A)$
        end if
        if $t > 0$ then
            $Q(s,a) = (1 - \alpha(t, v(s,a)))Q(s,a) + \alpha(t, v(s,a))(r + \gamma \max_{a' \in A} Q(s', a'))$
            $v(s,a) = v(s,a) + 1$
        end if
        $s = s'$
        $a = a'$
        Sample $s' \sim p(s'|s,a)$
        $r = r(s', a, s)$
        if $s'$ is terminal then
            $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a)) \cdot r$
            $v(s,a) = v(s,a) + 1$
            Done = True
        end if
        $t = t + 1$
    end while
end for
```

4.4 Expert-Q-learning

Next, given an MDP for which S is the disjoint union of S_{fix} and S_{learn} and a policy π_1 on S_{fix}, we construct a way to learn the optimal conditional policy on S_{learn} given an expert policy π_1 on S_{fix}, $Q^*|_{\pi_1}$ from experience. From theorem 3.2.1, we know that this is the unique fixed point of the contraction operator T mapping $\mathbb{R}^{S_{learn}\times A}$ to itself, defined by

$$\begin{aligned} TQ(s,a) := &\sum_{s'\in S_{learn}} p(s'|s,a)(r(s',a,s)+\gamma \max_{a'\in A} Q(s',a')) \\ &+ \sum_{s'\in S_{fix}} p(s'|s,a)(r(s',a,s)+\gamma((I-\gamma P_1)^{-1}(R_1+\gamma E_{1,2}\max Q))(s')). \end{aligned}$$

In order to use theorem 4.2.1, we need to first find sampling operators $\tilde{T}_t$ with $\mathbb{E}[\tilde{T}_t Q] = TQ$ that can be interpreted as interactions with some simulation. But this is easily done by giving a reward of $r(s',a,s)$ for transitions to $s'\in S_{learn}$ while rewarding transitions to $s'\in S_{fix}$ with

$$r(s',a,s)+\gamma((I-\gamma P_1)^{-1}(R_1+\gamma E_{1,2}\max Q))(s')$$

and treating them as terminal in order to only interact with states in S_{learn}. Again, the interpretation is that we get a direct prediction of the return from the model $((I-\gamma P_1)^{-1}(R_1+\gamma E_{1,2}\max Q)$ whenever the agent enters S_{fix}. In another framing, an expert tells us what value following their advice would have given the agent's current knowledge, encoded by the current estimates for Q.

More formally, we update our estimate of $Q^*|_{\pi_1}$, Q_t at each time step t by sampling a transition to s_{t+1} and a reward r_t given the current state s_t and an action a_t from the dynamics induced by the modified MDP $M(Q_t) = (S, A, p', r', p'_0)$ and updating Q_t as in Q-learning. Here, $M(Q_t)$ is defined as follows:

$$p'(s'|s,a) = p(s'|s,a)$$

for $s\in S_{learn}, s'\in S$,

$$p'(s|s,a) = 1$$

and

$$r(s,a,s) = 0$$

for $s \in S_{fix}$,

$$r'(s',a,s) = r(s',a,s)$$

for $s, s' \in S_{learn}$ and

$$r'(s',a,s) = r(s',a,s)+\gamma((I-\gamma P_1)^{-1}(R_1+\gamma E_{1,2}\max Q_t))(s')$$

for $s, s' \in S_{learn}, S_{fix}$ and the current estimate Q_t, while all $p'(s'|s,a)$ that are undefined so far are equal to zero and $r(s', a, s)$ for those transitions arbitrary. We don't need the penalties for selecting non-allowed actions as in section 3.3 since learning is restricted to S_{learn} and we only allow basic actions in the first place. The start state distribution p'_0 obviously should assign zero probability to states in S_{fix}. Furthermore, it can happen that even if all states in the original MDP are reachable from that MDP's start states, the same is not true for S_{learn}. In this case we can either use exploring states or modify the start distribution p_0 such that every "connected component" of S_{learn} is started in with nonzero probability, for example by parameterizing p_0 by the last terminal state (in $M(Q)$), e and setting

$$p_0^e(s) \propto P(S_{fix} \text{ is exited through } s \text{ after being entered in } e)$$

for e in S_{fix} and $p_0^e = p_0$ else. The third main result is:

Theorem 4.4.1. *For an MDP M with state space $S = S_{fix} \cup S_{learn}$, and a policy π on S_{fix}, the Q-values computed by Expert-Q-learning (algorithm 4) converge to $Q^*|_{\pi_1}$ with probability one, as long as $N \geq |S_{learn}|$, $\varepsilon > 0$, $\alpha(t, v(s,a)) = \frac{1}{v(s,a)+1}$, $T = \infty$ and the start distributions p_0^e are such that for every two states $s, s' \in S_{learn}$ there is a policy for which s' can be reached from s within finitely many steps with positive probability, given that the next initial state in S_{learn}, s_{t+1} is sampled from $p_0^{s_t}$ whenever $s_t \in S_{fix}$.*

Proof. As before, lemma 4.1.1 guarantees infinite visits to all state-actions pairs $s, a \in S_{learn}, A$ with probability one under the outlined conditions. The choice of α gives us the necessary conditions for α_t in theorem 4.2.1. The sampling induces a stochastic operator $\tilde{T}$ defined by

$$\tilde{T}Q(s,a) = \begin{cases} r + \gamma \max_{a' \in A} Q(s', a')) \text{ for } s' \in S_{learn} \\ r + \gamma((I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q))(s') \text{ for } s' \in S_{fix} \end{cases}$$

for all $s, a \in S, A$ with s' sampled independently for each (s,a) from $p(s'|s,a)$. By definition,

$$\begin{aligned} E[\tilde{T}Q(s,a)] &= \sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\ &+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma((I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q))(s')) \\ &= TQ(s,a) \end{aligned}$$

for all $s, a \in S_{fix}, A$. Furthermore, since there are only finitely many possible sampling outcomes, $\|\tilde{T}Q^*|_{\pi_1} - TQ^*|_{\pi_1}\|_\infty$ is bounded. We know that T is contraction from theorem 3.2.1, so the only thing left to show is that $\tilde{T}$ is still a contraction. But by equation 3.1.3, $(I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q)$ only depends on $p(s'|s,a)$ for $s \in S_{fix}$, but not for $s \in S_{learn}$, such that $\tilde{T}$ is just the Bellman expert operator for a modified MDP with determinstic transitions from S_{learn} to the respectively sampled states s' and a contraction. Now, theorem 4.2.1 gives us convergence. $\square$

Algorithm 4 Expert-Q-learning for approximating $Q^*|_{\pi_1}$

Input: MDP: M with state space partitioned in S_{fix}, S_{learn}, start distribution $p_0^e(s)$ parameterized by the last terminal state or state in S_{fix}, e, correct affine linear function

$$F =: (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2}(\cdot))$$

for a policy π_1 on S_{fix}, Number of episodes T, episode length N, $\varepsilon > 0$, discount rate γ, step size function α depending on time and visit count.

Output: Approximate values for $Q^*|_{\pi_1}$

```
Initialize $Q(s,a) = v(s,a) = 0$ for all $s, a \in S_{learn}, A$
for $e = 0$ to T do
  Sample $s' \sim p_0^{s'}$
  Done = False
  $t = 0$
  while Done is False and $t \leq N$ do
    Sample $u \sim U((0,1))$
    if $u > \varepsilon$ then
      $a' = \text{argmax}_{a' \in A} Q(s', a')$ with ties broken arbitrarily
    else
      Sample $a' \sim U(A)$
    end if
    if $t > 0$ then
      $Q(s,a) = (1 - \alpha(t, v(s,a)))Q(s,a) + \alpha(t, v(s,a))(r + \gamma \max_{a' \in A} Q(s', a'))$
      $v(s,a) = v(s,a) + 1$
    end if
    $s = s'$
    $a = a'$
    Sample $s' \sim p(s'|s,a)$
    $r = r(s', a, s)$
    if $s' \in S_{fix}$ then
      $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a))(r + \gamma((I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2} \max Q))(s'))$
      $v(s,a) = v(s,a) + 1$
      Done = True
    else if $s'$ is terminal then
      $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a)) \cdot r$
      $v(s,a) = v(s,a) + 1$
      Done = True
    end if
    $t = t + 1$
  end while
end for
```

Alternatively, one can use standard Q-learning on the modified, dynamic MDP $M(Q_t)$, which is equivalent. The simplest way of doing this uses a wrapper class that knows M and F_{π_1} and emulates $M(Q_t)$ for arbitrary Q_t. In particular, the theorem holds if S_{learn} is connected in the sense that every state can be reached from every other state with positive probability for some policy. Another example would be a connected S in the same sense for policies that are equal to π when restricted to S_{fix} combined with

$$p_0^e(s) > 0 \Leftrightarrow P(S_{fix} \text{ is exited through } s \text{ after being entered in } e) > 0$$

for $e \in S_{fix}$.

In order to actually use this algorithm, we need to somehow calculate our model for the return,

$$F_\pi := (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2}(\cdot))$$

for a policy π. In most cases, we won't have access to the actual transition probabilities, such that we also need to try learning this from simulations. The first approach would be to approximately learn the transition probabilities and rewards for transitions starting in S_{fix} from simulations. This has a few downsides: we need to store $|S_{fix}|^2 + |S_{fix}||S_{learn}|$ probabilities and invert a $|S_{fix}|$ dimensional quadratic matrix. This takes $O(|S_{fix}|^3)$ operations assuming standard matrix inversion and might be numerically unstable, which is bad because we can only approximate P_1. Lastly, there is no obvious way to generalize this to a continuous state space, which almost seems to be a prerequisite for usage in modern applications.

Instead, we notice that by construction and equation 3.1.4, F_{π_1} maps correct V_2-values on S_{learn} onto the correct V_1-values on S_{fix} for any policy π that is equal to π_1 on S_{fix}. This means, that we would be able to solve for the coefficients defining the affinely linear operator F_{π_1}, after systematically calculating those V_1-values for different V_2. Luckily, we can do this, without needing to evaluate any policies on S_{learn}: by replacing all states in $s \in S_{learn}$ by terminal states and giving an additional reward for reaching them equal to $\gamma V_\pi(s)$, we leave the Bellman equation for states in S_{learn} unchanged. This means that performing any converging policy evaluation method on S_{fix} for the policy π_1 with transitions to S_{learn} treated as terminal will give us V_1 for $V_2 = 0$, while adding a bonus reward of $b \in \mathbb{R}$ to the transition to $s' \in S_{learn}$ yields V_1 for $V_2(s') = b$. Instead of iteratively learning the values for V_2 equal to a basis of $\mathbb{R}^{S_{learn}}$ and the zero vector, in order to solve for the affinely linear F_{π_1}, we can treat the $V_{s'}$ as variables instead and directly learn $V_1(s)$ in the form $B(s) + \sum_{s' \in S_{learn}} W_{s'}(s) V_\pi(s')$. This way, we only need a single evaluation.

More formally, we will consider a modified MDP $M_v = (S, A, p', r', p'_0)$ with $1 + |S_{learn}|$ dimensional, vector-valued rewards and states in S_{learn} treated as terminal, but transitions to them receiving an extra reward in the corresponding component.

$$p'(s'|s,a) = p(s'|s,a)$$

for $s' \in S, s \in S_{fix}$,

$$p'(s|s,a) = 1$$

for $s \in S_{learn}$,

$$r'(s',a,s) = (r(s',a,s), 0, ..., 0)$$

for $s' \in S_{fix}, s \in S_{fix}$,

$$r'(s',a,s) = (r(s',a,s), 0, ..., 0, \gamma, 0, ...0)$$

for $s' \in S_{learn}, s \in S_{fix}$, where the γ is in the component corresponding to s', $s' - |S_{fix}| + 1$. Previously undefined transition probabilities are equal to zero and undefined rewards arbitrary. p'_0 is chosen such that all states in S_{fix} are reachable from states s with $p'_0(s) > 0$ given the policy π_1. All natural choices set $p'_0 = 0$ for states in S_{learn} (and other known terminal states). Intuitively, the component of the V-value for a policy π that corresponds to an exist state s is equal to the expectation of γ to the power of the time needed to arrive in this state.

We will use SARSA to approximate the Q-function of π in M_v, Q_{v,π_1} and translate the obtained values to V-values using

$$\mathbb{E}_{a \sim \pi_1}[Q_{v,\pi_1}(s,a)] = V_{v,\pi_1}(s).$$

Since SARSA converges to the correct values for scalar rewards, it will also do so componentwise for reward vectors. The same goes for any other method of policy evaluation. The $s' - |S_{fix}| + 1$th component of $V_{v,\pi_1}(s)$ is then equal to $W_{s'}(s)$: it is equal to the additional expected return gained from transitions to s' after starting in s, given that $V_2(s') = 1$. But this gained reward is clearly linear in $V_2(s')$, such that the additional reward is $(V_{v,\pi_1}(s))_{s'-|S_{fix}|+1} V_2(s')$ for arbitrary values of $V_2(s')$ which means that $(V_{v,\pi_1}(s))_{s'-|S_{fix}|+1} = W_{s'}(s)$. Similarly, the first component of the values $(V_{v,\pi_1}(s))_1$ is equal to the expected return until leaving S_{learn} and corresponds to the only term in $V_1(s)$ that is independent of V_2, $B(s)$.

The td(0) algorithm to approximate V_{v,π_1} on S_{fix} has higher variance in updates than SARSA, especially if the choice of action greatly influences the expected return, but saves memory and updates every state significantly more often (since there are fewer states than state-action pairs). Depending on the MDP and resource constraints, it might be a better choice. In the algorithm, the policy π_1 could either actually be performed by an expert, be prespecified by some formal method, or have been learned before by a neural network via imitation learning[49]. From the previous discussion combined with the convergence of SARSA, it is obvious that this would yield

Algorithm 5 SARSA based algorithm for approximating F_{π_1}

```
Input: MDP: $M$ with decomposed state space $S = S_{fix} \cup S_{learn}$, policy $\pi_1$ on $S_{fix}$, parameters for SARSA
Output: An approximation of $F$ in the form $W(\cdot) + B$
  Initalize $|S_{fix}| \times |L|$ array $W$, list of length $|S_{fix}|$: $B$
  Obtain $Q_{v,\pi}$ for the modified MDP $M$ via SARSA (Or any other method for approximating $Q_\pi$ or $V_\pi$) while
  adding the vector wise rewards componentwise
  for $k = 1$ to $|S_{fix}|$ do
      $s = k$
      $B[s] = \sum_{a \in A} \pi_1(a|s)(Q_{\pi_1}(s,a))_1$.
      for $i = 1$ to $|S_{learn}|$ do
          $s' = i$
          $W[s,s'] = \sum_{a \in A} \pi_1(a|s)(Q_{\pi_1}(s,a))_{s'+1}$
      end for
  end for
```

F_{π_1} given SARSA actually puts out the correct Q_{π_1}. With finite time, SARSA will only get close to the correct values, such that the result is only an approximation of F_{π_1}. It might also make sense to store the probabilities to transition to states in S_{learn} from different states, in order to better inform the choice of starting probabilities in $M(Q)$.

The algorithm can be framed as a combination of learning the reward and the transition model of π_1 interpreted as an option and was first described in [12]. More information on options can be found in the chapter on related work.

Of course in our case, we only really need the outputs of F_{π_1} for states in S_{fix} reachable from S_{learn}, such that we only need to calculate and save the corresponding "rows" of F_{π_1}, if we are given the possible entry states to S_{fix}. Similarly, we know that the corresponding column of W is zero, whenever a state in S_{learn} is not reachable from S_{fix} via π_1. Thus, if we know these states in advance, we can considerably reduce the size of the reward vector and the matrix W. Since this approach only requires Q-values to obtain F_{π_1}, it is simple to extend to function approximation: As long as the set of states in S_{learn} that is reachable from S_{fix} via π_1 is sufficiently small, a neural network can be trained to learn the map from (s, a) to the corresponding value of Q-vectors. This would be particularly useful if S_{fix} was continuous or very large, while there is a bottleneck between S_{fix} and S_{learn}, such that only few states in S_{learn} can be reached from S_{fix}. If these few "exit states" are known a priori, a low dimensional Q-vector can be learned. In the stylized MDP (Figure 5), F_{π_1} only needs to take in values for the blue nodes 4 and 5 and we only need to evaluate the second component, because we can only transition to node 2 from S_{learn}.

Instead of pre-specifying a policy for S_{fix}, the policy could also be learned to be optimal conditional on the values of the exit states to S_{learn} as done by Hauskrecht et al. [14] and Sutton, Precup and Singh[12], both of whom do not seem to combine this directly with learning F_{π_1}. As described in [42] and [43], Q-learning can easily be modified to learn the optimal policy for some linear combination of subrewards r_i, while learning Q-values Q_i that accurately predict the cumu-

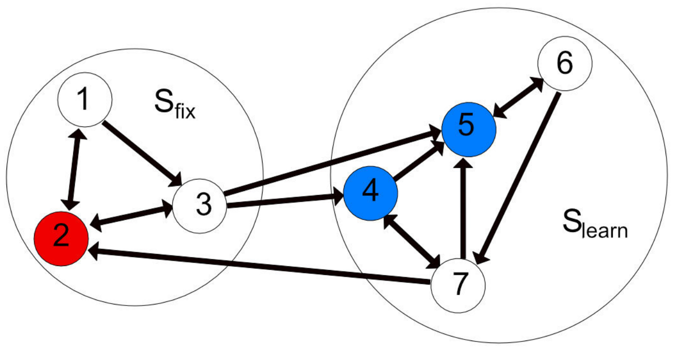


Figure 5: Entry nodes to S_{fix} and S_{learn}

lative discounted subrewards for that policy. This is achieved by replacing the $\max_{a' \in A} Q_i(s', a')$ in the updates by

$$Q_i(s, \text{argmax}_{a' \in A} \sum_n w_n Q_n(s', a')) \tag{4.4.1}$$

for some weights w_n and taking the action that maximizes the weighted sum of Q-values. If w_1 is set to 1, the learned policy is the optimal policy on S_{fix} conditioned on the values for the exit states to S_{learn} corresponding to the rest of the w_i. This policy (and accordingly the corresponding F_{π_1}) does not usually change with small changes in w and for some problems, a quick grid search of the w_i might be able to uncover all policies that have the potential to be optimal on S_{fix} for any values of the exit states. This can be done efficiently, since old Q-values are likely to provide a good initialization for the problem with slightly changed w.

Furthermore, the amount of necessary interactions with the simulation can be reduced significantly by using experience replay[11]. The basic idea of experience replay is to exploit the fact that the order of updates does not relevantly affect convergence for methods based on sampling the Bellman operator like SARSA and Q-learning by storing transitions and the corresponding rewards in a memory and repeatedly using them to update instead of throwing away every transition after performing the respective update. This way, we need to sample fewer transitions in order to converge, which can be helpful, especially if transitions are sampled from costly simulations or even real world data. As the rewards in M_v are independent from the w, we can reuse experience across multiple policies with this approach.

In practice, the finite runtime of SARSA and the corresponding positive lower bound on α will usually lead to errors in the components of W and B that are proportional to the final α: At the fixed point Q_{π_1}, updates will shift Q away from the fixed point, as long as the expectation in

$$Q_{\pi_1}(s,a) = \mathbb{E}_{\substack{s' \sim p(s'|s,a) \\ a' \sim \pi(a'|s')}}[r(s',a,s) + \gamma Q_{\pi_1}(s',a')]$$

is not attained by every sample. The shift is by the deviation times α, leading to large deviations for large final α. Because errors in F_{π_1} propagate to errors in the fixed point of expert-Q-learning $Q^*|_{\pi_1}$, it is advisable to use a (final) learning rate for the SARSA in the algorithm smaller than what one wants to use later. In order to quantify the effect of badly approximated weights on the fixed point $Q^*|_{\pi_1}$ of

$$\begin{aligned} TQ(s,a) = &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\ &+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma (F_{\pi_1}(\max Q))(s'), \end{aligned}$$

we rewrite the affinely linear operator $F_{\pi_1}(\cdot)$ as $W(\cdot) + B$ and consider a perturbed version $\tilde{F}_{\pi_1}(\cdot) = \tilde{W}(\cdot) + \tilde{B}$ as well as the induced perturbed operator

$$\begin{aligned} \tilde{T}Q(s,a) := &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\ &+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma (\tilde{F}_{\pi_1}(\max Q))(s') \end{aligned}$$

with fixed point $\tilde{Q}^*|_{\pi_1}$. By theorem 2.3.1, we get

$$\begin{aligned} &\|Q^*|_{\pi_1} - \tilde{Q}^*|_{\pi_1}\|_\infty \\ &\leq \frac{1}{1-\gamma} \|\tilde{T}Q^*|_{\pi_1} - Q^*|_{\pi_1}\|_\infty \\ &\leq \frac{1}{1-\gamma} \|\tilde{T}Q^*|_{\pi_1} - TQ^*|_{\pi_1}\|_\infty \\ &= \frac{1}{1-\gamma} \max_{s \in S_{learn}, a \in A} |\sum_{s' \in S_{fix}} p(s'|s,a)\gamma(F(\max Q^*|_{\pi_1})(s') - \tilde{F}(\max Q^*|_{\pi_1})(s'))| \\ &= \frac{1}{1-\gamma} \max_{s \in S_{learn}, a \in A} |\sum_{s' \in S_{fix}} p(s'|s,a)\gamma(W(\max Q^*|_{\pi_1})(s') + B(s') - \tilde{W}(\max Q^*|_{\pi_1})(s') - \tilde{B}(s'))| \\ &\leq \frac{\gamma}{1-\gamma} \|W \max Q^*|_{\pi_1} + B - \tilde{W} \max Q^*|_{\pi_1} - \tilde{B}\|_\infty \\ &\leq \frac{\gamma}{1-\gamma} (\|W \max Q^*|_{\pi_1} - \tilde{W} \max Q^*|_{\pi_1}\|_\infty + \|B - \tilde{B}\|_\infty) \\ &\leq \frac{\gamma}{1-\gamma} (\|W - \tilde{W}\|_\infty \|\max Q^*|_{\pi_1}\|_\infty + \|B - \tilde{B}\|_\infty) \end{aligned}$$

using that $\sum_{s' \in S_{fix}} p(s'|s,a) \leq 1$ for all $s, a \in S_{learn}, A$. This does seem bad for large γ. However, expert policies can be reused for different MDPs whose state space includes S_{fix}, which allows for spending more time on the SARSA used to obtain the estimates for W and B and a lower final learning rate, which should bring down the error to an acceptable level. Furthermore, it does seem plausible that the bound is far from sharp for a lot of MDPs: The third inequality can be quite loose, especially if transitions from S_{learn} to S_{fix} are somewhat rare. Also, after long enough training there is little reason to believe that the entries $W - \tilde{W}$ all have the same sign, while $\max Q^*|_{\pi_1}$ will often have the same sign in all entries, such that

$$\|(W - \tilde{W}) \max Q^*|_{\pi_1}\|_\infty \ll \|W - \tilde{W}\|_\infty \|\max Q^*|_{\pi_1}\|_\infty.$$

While there is no similar argument for $B - \tilde{B}$, the corresponding errors are at least not amplified by the magnitude of the Q-values.

4.5 Multi-Expert-Q-learning

Given a set of policies Π on S_{fix}, we can approximate $Q^*|_\Pi(s,a)$ using an analogous approach for the contracting Bellman operator

$$\begin{aligned} TQ := &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\ &+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q))(s')). \end{aligned}$$

Again, theorem 4.2.1 is used for stochastic operators $\tilde{T}_t$ with $\mathbb{E}[\tilde{T}_t Q] = TQ$ that can be interpreted as sampling transitions for some form of MDP. This time, the modified MDP $M(Q_t)$ is defined as in the previous section, but with

$$r'(s',a,s) = r(s',a,s) + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_{\pi^k})^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q_t))(s')$$

for $s, s' \in S_{fix}, S_{learn}$ and the current estimate Q_t.

With this, we can formulate the last and most important main result:

Theorem 4.5.1. *For an MDP M with state space $S = S_{fix} \cup S_{learn}$, and a set of policies Π on S_{fix}, the Q-values computed by Multi-Expert-Q-learning (algorithm 6) converge to $Q^*|_\Pi$ with probability one, as long as $N \geq |S_{learn}|$, $\varepsilon > 0$, $\alpha(t, v(s,a)) = \frac{1}{v(s,a)+1}$, $T = \infty$ and the start distributions p_0^e are such that for every two states $s, s' \in S_{learn}$ there is a policy for which s' can be reached from s within finitely many steps with positive probability, given that the next initial state in S_{learn}, s_{t+1} is sampled from $p_0^{s_t}$ whenever $s_t \in S_{fix}$.*

Algorithm 6 Multi-Expert-Q-learning for approximating $Q^*|_\Pi(s,a)$

Input: MDP: M with state space partitioned in S_{fix}, S_{learn}, start distribution $p^e(s)$ parameterized by the last terminal state or state in S_{fix}, e, correct

$$F =: \max_{\pi^k \in \Pi} (I - \gamma P_1)^{-1}(R_{\pi^k} + \gamma E_{\pi^k}(\cdot))$$

for a policy π_1 on S_{fix}, Number of episodes T, episode length N, $\varepsilon > 0$, discount rate γ, step size function α depending on time and visit count.

Output: Approximate values for $Q^*|_\Pi$

```
Initialize $Q(s,a) = v(s,a) = 0$ for all $s, a \in S_{learn}, A$
for $e = 0$ to T do
    Sample $s' \sim p_0^{s'}$
    Done = False
    $t = 0$
    while Done is False and $t \leq N$ do
        Sample $u \sim U((0,1))$
        if $u > \varepsilon$ then
            $a' = \text{argmax}_{a' \in A} Q(s', a')$ with ties broken arbitrarily
        else
            Sample $a' \sim U(A)$
        end if
        if $t > 0$ then
            $Q(s,a) = (1 - \alpha(t, v(s,a)))Q(s,a) + \alpha(t, v(s,a))(r + \gamma \max_{a' \in A} Q(s', a'))$
            $v(s,a) = v(s,a) + 1$
        end if
        $s = s'$
        $a = a'$
        Sample $s' \sim p(s'|s,a)$
        $r = r(s', a, s)$
        if $s' \in S_{fix}$ then
            $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a))(r + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_1)^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q))(s'))$
            $v(s,a) = v(s,a) + 1$
            Done = True
        else if $s'$ is terminal then
            $Q(s,a) = (1 - \alpha(t, v(s,a))Q(s,a) + \alpha(t, v(s,a)) \cdot r$
            $v(s,a) = v(s,a) + 1$
            Done = True
        end if
        $t = t + 1$
    end while
end for
```

Proof. We repeat the proof of theorem 4.4.1 with $\tilde{T}$ defined by

$$\tilde{T}Q(s,a) := \begin{cases} r + \gamma \max_{a' \in A} Q(s',a')) \text{ for } s' \in S_{learn} \\ r + \gamma \max_{\pi^k \in \Pi}((I - \gamma P_1)^{-1}(R_{\pi^k} + \gamma E_{\pi^k} \max Q))(s')) \text{ for } s' \in S_{fix} \end{cases}.$$

□

Again, perturbations of the operator $F =: \max_{\pi^k} F_{\pi^k} =: \max_{\pi^k \in \Pi}(I - \gamma P_1)^{-1}(R_{\pi^k} + \gamma E_{\pi^k}(\cdot))$ will propagate to errors in the fixed point. This time with writing

$$F(\cdot) = \max_{\pi^k \in \Pi} W_{\pi^k}(\cdot) + B_{\pi^k}$$

and the perturbed version

$$\tilde{F}(\cdot) = \max_{\pi^k \in \Pi} \tilde{W}_{\pi^k}(\cdot) + \tilde{B}_{\pi^k}$$

and the corresponding fixed point $\tilde{Q}^*|_\Pi$ of

$$\begin{aligned}\tilde{T}Q(s,a) := &\sum_{s' \in S_{learn}} p(s'|s,a)(r(s',a,s) + \gamma \max_{a' \in A} Q(s',a')) \\ &+ \sum_{s' \in S_{fix}} p(s'|s,a)(r(s',a,s) + \gamma(\tilde{F}(\max Q))(s'),\end{aligned}$$

we have

$$\begin{aligned}
&\|Q^*|_\Pi - \tilde{Q}^*|_\Pi\|_\infty \\
&\leq \frac{1}{1-\gamma}\|\tilde{T}Q^*|_\Pi - Q^*|_\Pi\|_\infty \\
&\leq \frac{1}{1-\gamma}\|\tilde{T}Q^*|_\Pi - TQ^*|_\Pi\|_\infty \\
&= \frac{1}{1-\gamma} \max_{s \in S_{learn}, a \in A} |\sum_{s' \in S_{fix}} p(s'|s,a)\gamma(F(\max Q^*|_\Pi)(s') - \tilde{F}(\max Q^*|_\Pi)(s'))| \\
&= \frac{1}{1-\gamma} \max_{s \in S_{learn}, a \in A} |\sum_{s' \in S_{fix}} p(s'|s,a)\gamma(\max_{\pi^k \in \Pi}(W_{\pi^k}(\max Q^*|_\Pi)(s') + B_{\pi^k}(s')) \\
&\quad - \max_{\pi^k \in \Pi}(\tilde{W}_{\pi^k}(\max Q^*|_\Pi)(s') + \tilde{B}_{\pi^k}(s')))| \\
&\leq \frac{\gamma}{1-\gamma}\|\max_{\pi^k \in \Pi}(W_{\pi^k} \max Q^*|_\Pi + B_{\pi^k}) - \max_{\pi^k \in \Pi}(\tilde{W}_{\pi^k} \max Q^*|_\Pi + \tilde{B}_{\pi^k})\|_\infty \\
&\leq \frac{\gamma}{1-\gamma} \max_{\pi^k \in \Pi}\|(W_{\pi^k} \max Q^*|_\Pi + B_{\pi^k}) - (\tilde{W}_{\pi^k} \max Q^*|_\Pi + \tilde{B}_{\pi^k})\|_\infty \\
&\leq \frac{\gamma}{1-\gamma} \max_{\pi^k \in \Pi}(\|W_{\pi^k} \max Q^*|_\Pi - \tilde{W}_{\pi^k} \max Q^*|_\Pi\|_\infty + \|B_{\pi^k} - \tilde{B}_{\pi^k}\|_\infty) \\
&\leq \frac{\gamma}{1-\gamma} \max_{\pi^k \in \Pi}(\|W_{\pi^k} - \tilde{W}_{\pi^k}\|_\infty \|Q^*|_\Pi\|_\infty + \|B_{\pi^k} - \tilde{B}_{\pi^k}\|_\infty)
\end{aligned}$$

using that $\sum_{s' \in S_{fix}} p(s'|s,a) \leq 1$ for all $s, a \in S_{learn}, A$ and that by the argument used in theorem 2.3.5

$$\begin{aligned}
&|\max_{\pi^k \in \Pi}(W_{\pi^k}(\max Q^*|_\Pi)(s') + B_{\pi^k}(s')) - \max_{\pi^k \in \Pi}(\tilde{W}_{\pi^k}(\max Q^*|_\Pi)(s') + \tilde{B}_{\pi^k}(s'))| \\
&\leq \max_{\pi^k \in \Pi} |(W_{\pi^k}(\max Q^*|_\Pi)(s') + B_{\pi^k}(s')) - (\tilde{W}_{\pi^k}(\max Q^*|_\Pi)(s') + \tilde{B}_{\pi^k}(s'))|
\end{aligned}$$

for all $s' \in S_{fix}$. This means that the error bound is as high as it would have been for the policy π^k with the worst approximations for W_{π^k} and B_{π^k}. Again, most bounds are likely to be far from sharp. Since the perturbations will essentially be random for the approximations outlined in the last section and the expected value of the maximum of random variables can exceed the maximum of expected values by far (especially if there are a lot of variables), we still expect bigger deviations in practice and it might make sense to train for longer and/or with a lower final learning rate than in the single expert case, in order to keep the error under control.

Because F_{π_1} is affinely linear for every policy on S_{fix}, π^k, it is convex in every component, i.e.

$$F_{\pi^k}(tx + (1-t)y) \leq tF_{\pi^k}(x) + (1-t)F_{\pi^k}(y)$$

holds componentwise, since we actually have equality. Thus F is convex as well:

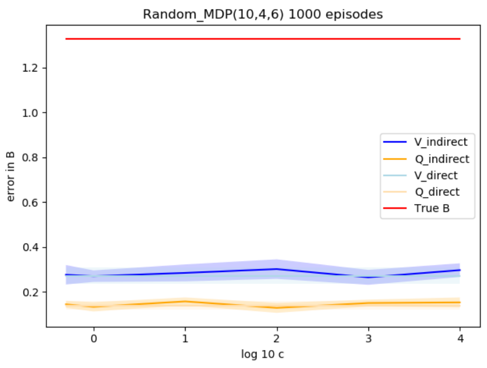
$$\begin{aligned}
F(tx + (1-t)y) &= \max_{\pi^k} F_{\pi^k}(tx + (1-t)y) \\
&\leq \max_{\pi^k}(tF_{\pi^k}(x) + (1-t)F_{\pi^k}(y)) \\
&\leq t \max_{\pi^k} F_{\pi^k}(x) + (1-t) \max_{\pi^k} F_{\pi^k}(y) \\
&= tF(x) + (1-t)F(y)
\end{aligned}$$

because the maximum of a sum cannot be larger than the sum of the maxima. This means that increasing larger $\max Q$-Values will often have a stronger effect on the output of F than increasing smaller ones. The basic intuition behind this is that the larger $\max Q$ is in a state $s \in S_{learn}$, the more optimization pressure exists for the policy in S_{fix} to increase the likelihood of terminating in s quickly. But the quicker a policy terminates in s, the faster its returns will grow with increasing $\max Q(s)$. This is mostly the case for large Π where the maximum in F is attained by a variety of policies at different points during training. Surely, the amount of optimization pressure does not really matter, if there is little to optimize. For small Π, or Π containing a lot of dominated policies, the maximum in F will only be attained by a few policies and the always piecewise linear F will have only a few nonlinearities. Because the entries in W_{π^k} are equal to the discounted probability of exiting S_{fix} via a specific state in S_{learn} they are non-negative. This means that F is monotonuosly growing in all components of $\max Q$ in every component, which should not come as a surprise because larger Q-Values outside of S_{fix} cannot lead to worse returns given the conditionally optimal behavior enforced by the maximization over policies.

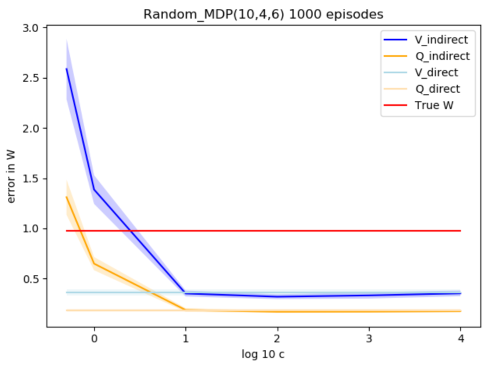
5 Experiments

At first, we compare the approach to learning F_π from algorithm 5 with learning values for fixed proxy rewards and solving the resulting system of equations. We also compare our approach with estimating the transition probabilities $P_1, E_{1,2}$ as well as the reward R_1 for a policy π and directly calculating $F_\pi := (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2}(\cdot))$. In order to do this, we randomly generate MDPs with 10 states and 4 actions. This is done by sampling rewards $r(a, s)$ that are independent of the transition uniformly and independently from $[0, 1]$ for all state-action pairs $s, a \in S, A$. The transition probabilities $p(\cdot|s, a)$ starting from s, a are obtained by sampling uniformly from $[0, 1]$ in every component and then normalizing. S_{fix} is equal to the first 6 states, while the last 4 represent S_{learn}.

First we look at the uniform policy and compare algorithm 5, using $SARSA$ *(Q-direct)* and td(0) *(V-direct)* with the other approach based on solving a system of equations discussed in 4.4 before the introduction of algorithm 5. For this, we use different scalar multiples c of the standard basis *(Q-indirect)* and *(V-indirect)*. The true $F_\pi = W_\pi + B_\pi$ is calculated from the true transition matrix and we plot $\|\tilde{W} - W\|_\infty$ and $\|\tilde{B} - B\|_\infty$ for $\tilde{W}$ and $\tilde{B}$ obtained by the respective algorithm with $\alpha_t = 0.1$ constant, $\varepsilon = 0$, the episode length N limited to 100 and $T = 1000$ training episodes (for each approximation in the indirect case). γ is equal to 0.99. Each experiment is run on 25 different randomly generated MDPs and the error bars represent 95% confidence intervals under the assumption of normaly distributed errors.



(a) approximation of B

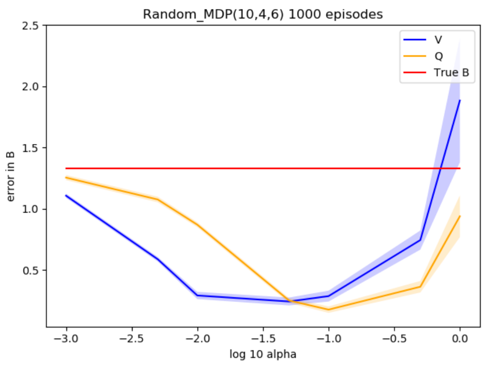


(b) approximation of W

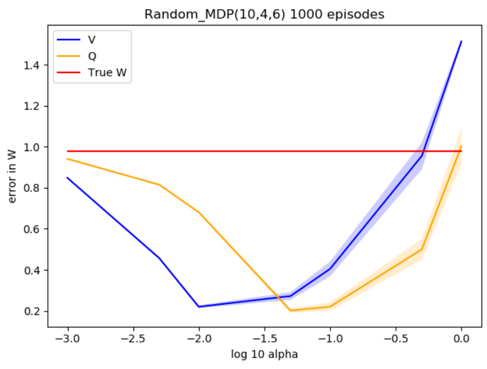
Figure 6: Multireward (direct) vs system of equations (indirect) approach to approximating F a) For B, the direct/indirect distinction does not seem to make a difference. The Q-based approach works better than the V-based one. Also, c does not seem to matter, which makes sense because the values are derived from SARSA, without additional calculations. b) For W, small c lead to bad approximations. For large c, the direct/indirect distinction ceases to matter. Again, the Q-based method works better than the B-based one.

In figure 6, we can see that the indirect approach works as well as the direct one only for large enough c. This is because in the indirect approach we increase the variance by subtracting two estimators instead of using a single estimator. For small c, the effect is larger because for those, the real difference is small and the noise plays a proportionally larger role. The results validate the fact that the more elegant direct approach works. Because the indirect approach needs to learn the values multiple times in a naive implemenation, we will focus on the direct one for efficiency. Interestingly, the Q-based algorithms consistently outperform the corresponding V-based ones in this regime.

Next, we investigate the role of α using the same setting as before and only considering direct methods.



(a) approximation of B

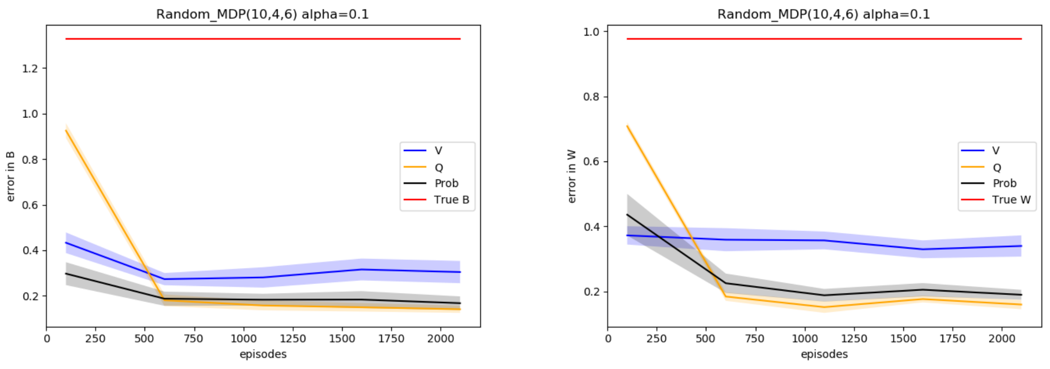


(b) approximation of W

Figure 7: Q vs. V based direct methods of approximating F with varying α. a) For B, both the Q- and the V-based approach first improve with increasing α and then get worse again. For large α, the Q-based approach is better than the V-based one. b) For W, the behavior is qualitatively similar. The relative errors are larger for large α than for B.

Figure 7 shows allows us to conclude the following: First, the choice of V- or Q-based algorithms does not seem to be too important for policy evaluation, as long as α is chosen appropriately and the memory overhead of the $Q-$based methods is ignored. On the other hand, the choice of α greatly matters. If α is too small or too large, the error is large. The error for small α is likely due to initialization bias, while the error for the large alpha is caused by high-variance updates. Interestingly, there is very little difference in the relative errors for B and W, even though one is expressed in a vector norm, while the other one uses a matrix norm.

Now, we check whether directly learning a model of the transitions and rewards by saving visitation frequencies and averaging rewards followed by direct computation of $F_\pi = (I - \gamma P_1)^{-1}(R_1 + \gamma E_{1,2}(\cdot))$ is competitive with dynamic programming. With the same settings as before and α fixed to 0.1, we compare the error for the direct Q and V methods with the model-based approach *(prob)* for different numbers of episodes.

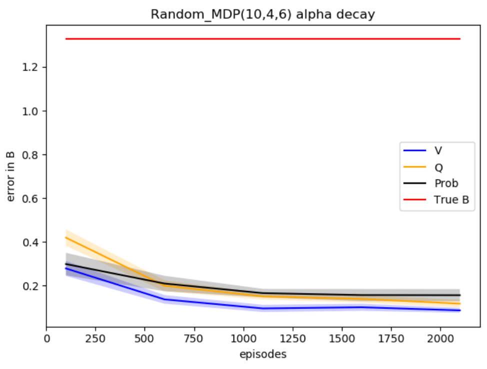


(a) approximation of B (b) approximation of W

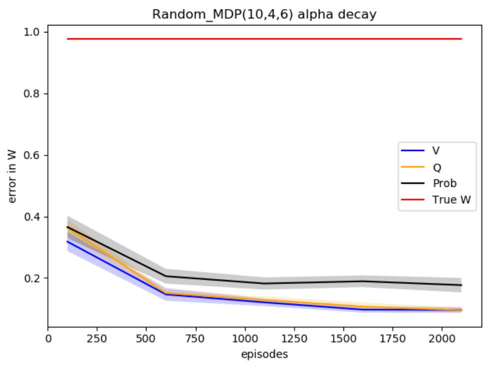
Figure 8: Direct and model-based methods of approximating F with varying number of episodes a) For B, the model-based method performs best for small amounts of episodes, but only improves very little with additional episodes, such that the $Q-$based approach seems to be better for large numbers of episodes. The V-based approach seems to stagnate quite early. b) Qualitatively the picture is similar for W, but the differences in relative error are more clear.

We can see two interesting effects in figure 8: first, the accuracy of the V-based approach barely changes with the number of episodes. It begins better than the Q-based version but stagnates immediately, while the Q-based approach keeps improving and quickly becomes more accurate. This is probably because of the aggregation of different actions in the V-function: this increases the variance of the final estimate but allows for more updates, which corresponds to a faster reduction in bias of the value estimates used as target. Secondly, the model-based approach is competitive with the Q-based one in this regime. However, both approaches stagnate which seems to be harder to combat for the model-based one, where there is no parameter α to adjust. A possible reason for this might be related to numerical problems in direct matrix inversion.

We repeat the experiment with decaying α equal to one over the visit count of a state or state-action pair (as in theorem 4.1.2) instead of constant α.



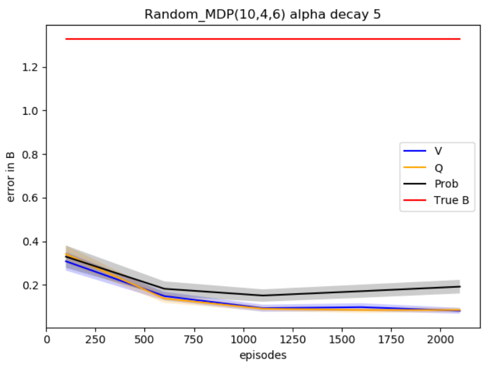
(a) approximation of B



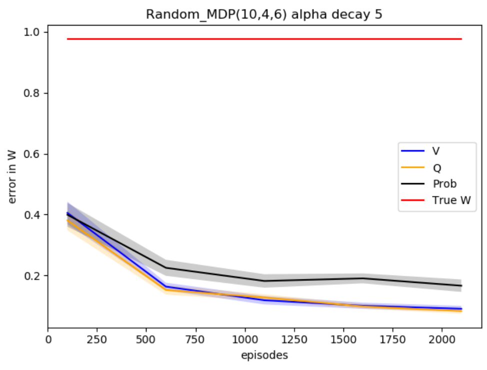
(b) approximation of W

Figure 9: Direct and model-based methods of approximating F with decaying alpha. a) The V-based method clearly beats the model-based one, as well as the Q-based one in terms of error for B, except for the case with very few episodes, where the model-based approach is on par. b) For W, both the Q- and the V-based approach clearly beat the model-based approach for non-small amounts of episodes. There is no clear difference between the Q- and the V-based method's error.

In figure 9, we can see that letting alpha decay can make a large difference. For longer episodes, both the V and the Q-based method clearly outperform the model-based approach. V seems to profit more: It progresses from being consistently worse than the other two approaches to better or equal. One possible explanation is that the decay in α reduces variance and variance is a larger problem for V than for Q. The field becomes more even, if we replace $\alpha_t = \frac{1}{v(s,a)+1}$ by the slower decaying $\alpha_t = \frac{5}{v(s,a)+5}$:



(a) approximation of B

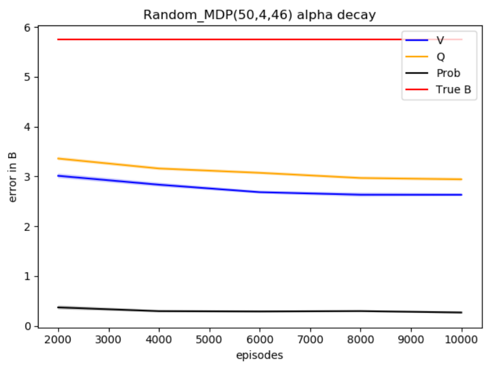


(b) approximation of W

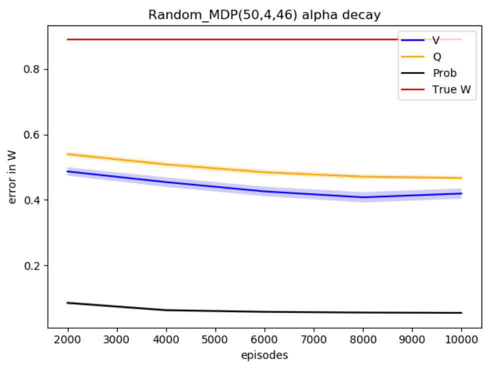
Figure 10: Direct and model-based methods of approximating F with slower decaying alpha. a) The Q- and the V-based approach produce similar amounts of error and beat the model-based approach for large episode lengths for B. b) For W, the results are qualitatively the same, but the relative errors are larger in the few-episode regime.

Figure 10 shows that the Q-based approach benefits from slower decay in the learning rate, whereas the V-based one does not seem to be influenced by small changes in the speed of decay. The former is probably because larger α in the beginning allow for a quicker correction of the bias towards the inital values, which is more problematic for SARSA than for td(0).

Next we try a larger MDP (50 states and the first 46 in S_{fix}) and repeat the comparision of the V,Q and model-based approach with decaying alpha ($\frac{1}{v(s,a)+1}$) and more episodes to compensate for the larger state space.



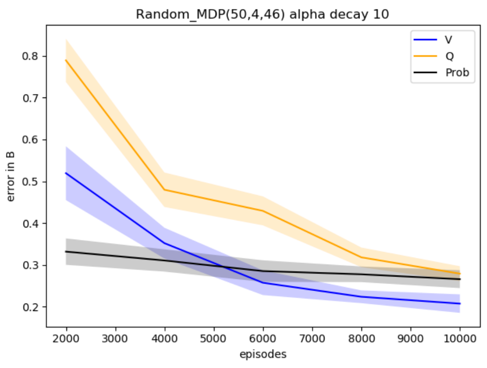
(a) approximation of B



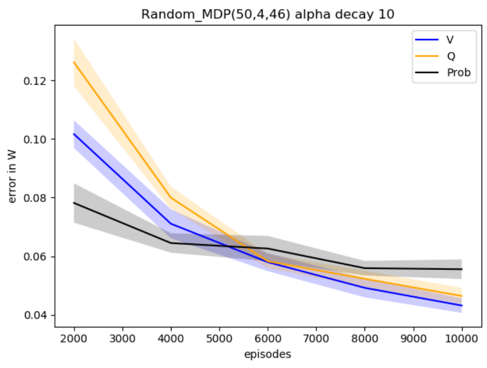
(b) approximation of W

Figure 11: Direct and model-based methods of approximating F on larger MDP. a) The model-based approach clearly beats both the Q-based and the V-based approach for B. The Q-based approach performs worse than the V-based one and both improve only slowly with more episodes. b) The same happens for W.

In figure 11 we can see that the model-based approach completely outperforms the Q-based and the V-based approaches with the standard α schedule. Again, we try to change this by adapting the speed of decay. This time, we use $\alpha_t = \frac{10}{v(s,a)+10}$.



(a) approximation of B



(b) approximation of W

Figure 12: Direct and model-based methods of approximating F on larger MDP with slow α decay. a) The V-based approach is now the best one for large amounts of episodes, and the Q-based approach is on par with the model-based one in that regime for B. For small amounts of episodes, the model-based approach still wins. b) For W, the relative difference between the Q- and V-based approach are less pronounced and both beat the model-based one for large amounts of episodes and lose to it for small amounts.

Figure 12 shows that the model-based approach can be beaten by value function based approaches for the larger MDP with a better choice of α-decay. So while value based options seem to consistently be able to beat model-based approaches in the long run, they seem to require more parameter tuning to work well.

Now, to validate algorithm 4 and 6, we consider the following MDP where a robot can choose between different tasks with different reward and difficulty profiles and needs to refuel in between: The state space consists of the robot's current energy level (starting at 16) as well as two progress meters for the more difficult tasks (both starting at zero) and an indicator that shows whether or not the robot is refueling. At every step, the robot can choose between four actions. While the robot is not refueling, action 0 corresponds to doing a simple task that yields a reward of one but reduces the energy meter by one. As the robot has difficulty performing when its energy is depleted, action 0 has no effect when the robot has no or only one energy left. Action 1 starts a medium difficult task. This reduces the robot's energy by five and raises the first progress meter by one if it started at zero. If the meter started at 1, the robot also gets a reward of ten. If the energy level is below five, action 1 does nothing (it also does not reduce the robot's energy then). Action 2 corresponds to starting a hard task that reduces the robot's energy by five but raises the second progress meter by one if it was below two and rewards 100 if the meter was at two. If the robot starts with less than five energy, action 2 does nothing. Action 3 starts the refueling process and resets all progress meters. While the robot is refueling, action 3 increases the robot's energy level by nine, as long as it starts below two, by four if it starts at six or higher and smaller than twelve and by three up to a maximum of 16, else; The robot's batteries are a lot easier to charge if they're close to empty. The refueling process is interrupted by chosing any action different from 3. The action stopping the refuel has no further effects.

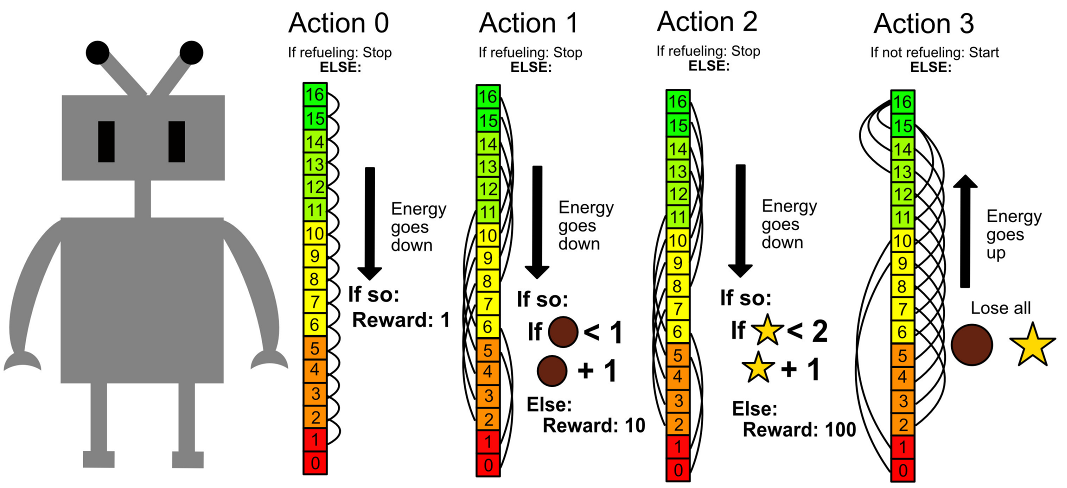
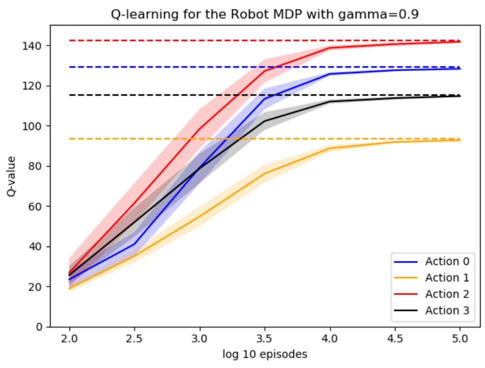
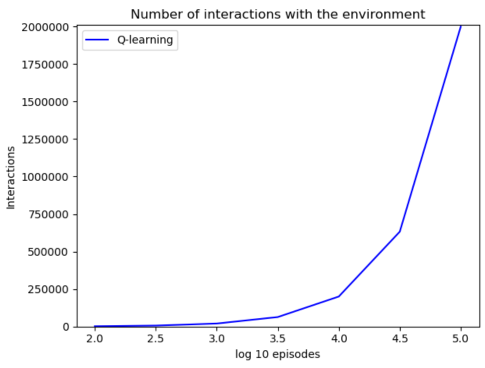


Figure 13: The robot MDP. The circle and the star represent the first and second progress meter.

We begin with Q-learning with an episode length of 20, $\varepsilon = 0.25$ and $\alpha_t = \frac{10}{v(s,a)+10}$ and different amounts of episodes. The results are averaged over 100 episodes to avoid spurious effects due to random exploration.



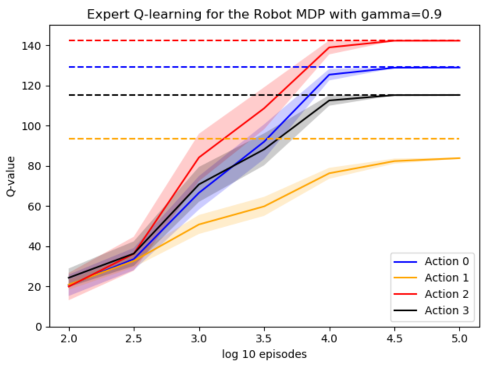
(a) Q-values



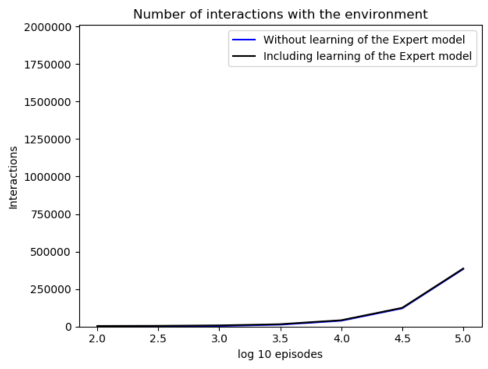
(b) Interactions

Figure 14: Q-values for the initial state in the robot MDP. The dashed lines represent the correct Q^*-values. a) Q-learning seems to converge to the correct $Q-$values. b) 2 million interactions with the environment are needed to produce the last set of Q-values.

Even though the state space is rather small and the MDP is deterministic, figure 14 shows that Q-learning needs around 100000 episodes and a total of 2000000 interactions with the environment to get close to the correct Q-value for action 2. This Q-value is around 142 and corresponds



(a) Q-values



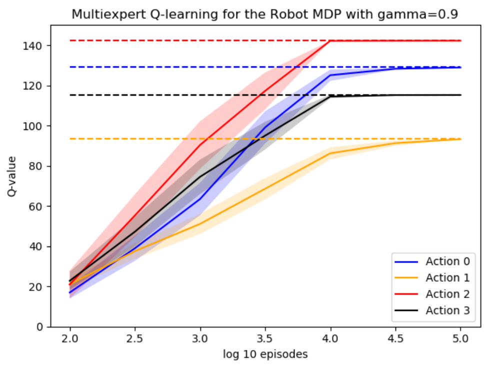
(b) Interactions

Figure 15: Q-values for the initial state using expert-Q-learning. The dashed lines represent the correct Q^*-values. a) Expert-Q-learning converges for all actions except for action 1. b) Less than five hundred thousand interactions with the environment are needed. The interactions needed for learning F barely affect the total number of interactions.

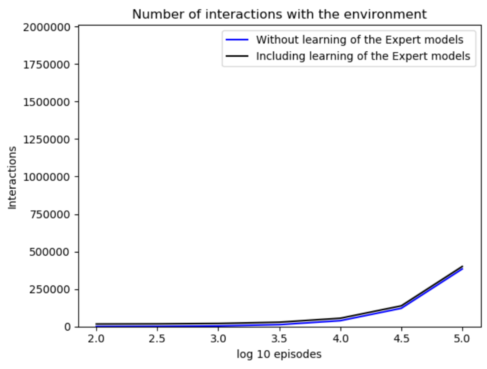
to taking action 2 three times in the beginning and then looping action 3 four times followed by action 2 four times. One problem for Q-learning is that the agent needs a fair amount of random exploration to realize that action two yields the most return in the long run. But as long as action 2 is seen as bad, there is little incentive to fully refuel, which makes it even harder to realize that action 2 is good.

Next, we combine our MDP with the expert policy of always fully refueling, once the robot started to refuel. S_{fix} thus corresponds to states, where the robot refuels, while S_{learn} contains all other states. We manually specify the policy and use algorithm 5 with SARSA, $N = 5$, $T = 1000$, $\gamma = 0.9$, $\varepsilon = 0$, starts in refueling mode with uniformly random energy and α as above to learn F_π and apply expert-Q-learning with the same parameters as used previously. We can see a few things in figure 15: First, the Q-values learned by Expert-Q-learning are close to the correct ones, except for action 1, which is far from optimal and thus irrelevant for the greedy policy. However, expert-Q-learning has learned in less than a quarter of the interactions with the environment that Q-learning needed, even when factoring in the learning of F and is thus way more efficient. Using prior knowledge actually makes training a lot easier.

Now we add two more expert policies: refueling until the energy is at least 11 and refueling until it is at least 8. We learn the models F_π for these policies as before and combine them via pointwise maximization. Then, we apply Multi-expert-Q-learning with the same parameters we used in the the last two examples.



(a) Q-values

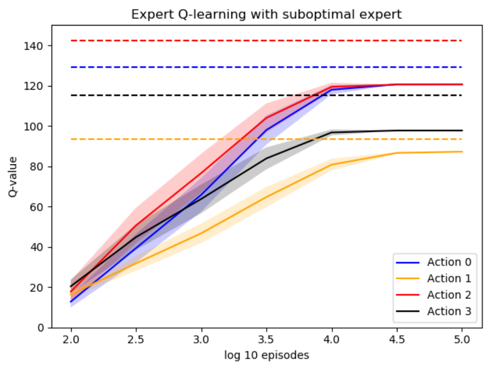


(b) Interactions

Figure 16: Q-values for the initial state using Multiexpert-Q-learning. The dashed lines represent the correct Q^*-values. a) Multiexpert-Q-learning converges for all actions. b) Again, less than five hundred thousand interactions with the environment are needed. This time, the effect of including the learning of F is more noticeable, but it is still negligible.

Figure 16 shows us that Multiexpert-Q-learning converges for the initial state as well. Due to two more learned models, it slightly less efficient with respect to environment interactions than expert-Q-learning when learning the F is factored in. However, it gets a lot closer to the correct value for action 1 while still needing only a quarter of the interactions Q-learning took. This is likely caused by more efficient early updates of the value of action 1: As the learning rate α_t decays with the visit count, early updates can have a large effect on learning speed. Having multiple policies yields larger "lower bounds" for the value of entering S_{fix}, because the update is always based on the current best guess for the optimal policy in Π. The conditional choice of policy induces optimism early on and leads to faster convergence to the positive correct Q-values in our case. At the fixed point, the true optimal fixed policy is chosen, such that the optimism does not lead us to overestimate the correct Q-values.

Next, we look at expert-Q-learning with a suboptimal policy trained as before: The robot only refuels as long as its energy is below 14, such that the optimal loop of four times 3 and four times 2 (after starting with three times 2) does not work anymore.

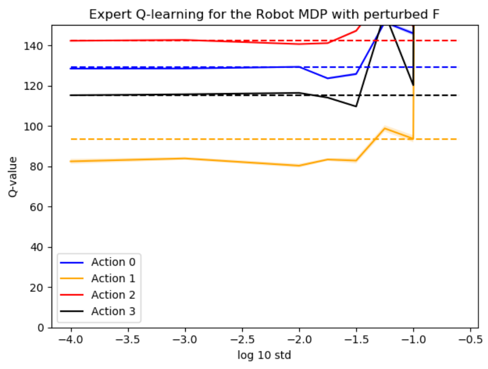


(a) Q-values

Figure 17: Q-values for the initial state using a suboptimal expert. The dashed lines represent the correct Q^*-values. The Q-values for all actions stay below the correct values for optimal behavior. The values for action 0 and 2 seem indistinguishable for large amounts of episodes.

We see that in figure 17, action 0 and 2 converge to the same value. This is because the robot now needs to complete the simple task once before the hard one is completed in order to fully drain its energy and be able to refuel completely. Because this can be done before starting the hard task or after having started it, both actions should converge to the same Q-value given the expert policy, as they seem to do. This experimente also illustrates, that the quality of expert matters a lot. Chosing a suboptimal expert has severe effects on the Q-values in the presented case.

Next, we investigate the effect of errors in the estimation of F on the learned Q-values. We calculate F for the policy of always fully refueling by algorithm 5 as before but with $T = 100000$. Then, we perturb all components of F (both W and B) by independent normal noise with different standard deviations and look at the Q-values for the intial state produced by Expert-Q-learning with the usual parameters and $T = 100000$.

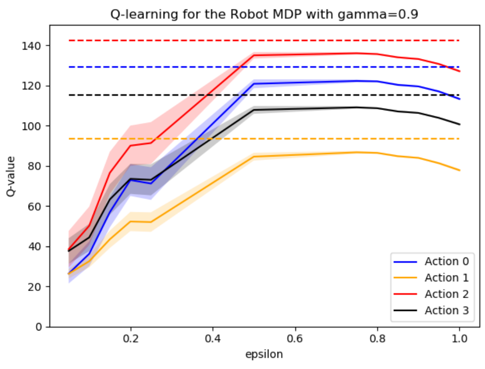


(a) Q-values

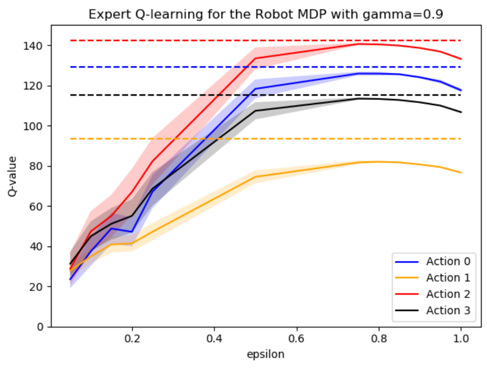
Figure 18: Q-values for the initial state using Expert-Q-learning with a perturbed model. The values are almost identical to the ones learned by Expert-Q-learning with a non-perturbed model for standard deviations up to around 0.03; For larger standard deviations, the Q-values explode.

Figure 18 shows an interesting effect: small perturbations of F don't seem to affect expert-Q-learning too much, at least on average, but perturbations above a certain threshold cause massive distortions. The most plausible explanation for this seems to be that perturbations have expectation zero and therefore mostly balance each other out for small amounts of noise, since all Q-values are positive. Once the noise gets too large, the contraction-property is broken and the iteration diverges.

We also want to investigate the dependence of the different algorithms on some of the learning algorithms' parameters. Since exploration seems to be the main bottleneck in our MDP, we test different values of ϵ for Q-learning and expert Q-learning with all of the other parameters as usual and $T = 1000$.



(a) Q-values

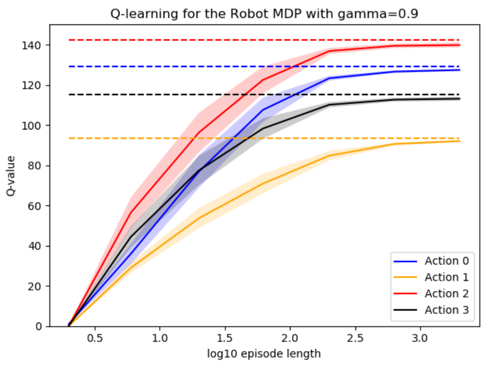


(b) Q-values

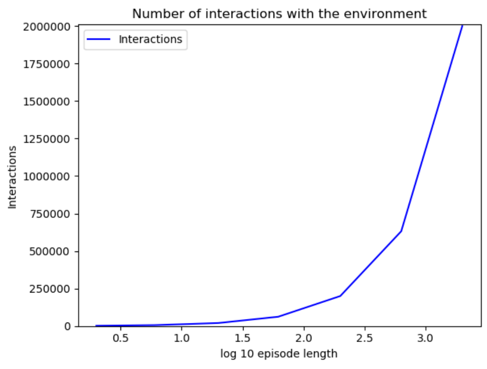
Figure 19: Q-values for the initial state using Multiexpert-Q-learning and Q-learning. a) For Q-learning, increasing ε improves performance up to a value of around 0.8; Afterwards, increasing ε leads to worse Q-values. b) For Expert-Q-learning, the same happens. However, at the optimal epsilon, Expert-Q-learning is closer to the correct values for all actions but action 1.

Figure 19 shows us that both algorithms benefit from increasing epsilon up to a threshold at which the effect reverses. Interestingly, the drop in performance for larger ε, due to the chains becoming harder to complete once learned, seems to be sized similarly for both algorithms.

The prespecified episode length N in Q-learning might also affect the training. We use $T = 1000$ episodes, leave all other parameters unchanged and vary how long we interact with the environment in each episode until we reset. In principle, this can be as long as we want, as the environment allows to reach all other states in finite time from every state.



(a) Q-values

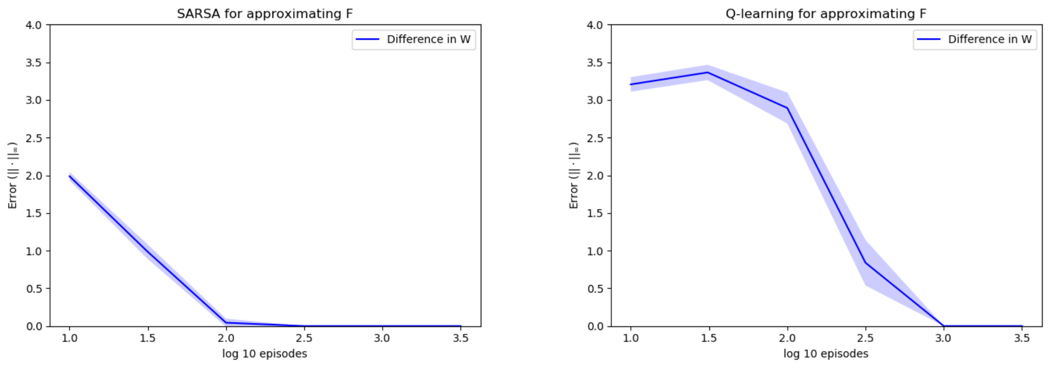


(b) Interactions

Figure 20: Q-values for the initial state using Q-learning with different episode lengths. a) Q-learning again seems to converge to the correct Q-values, albeit slightly slower than before. b) Again, in the end 2 million interactions with the environment are used.

As expected, the results are bad for very short episode lenghts, as the optimal chain takes eight steps and cannot be learned in episodes shorter than that. Interestingly, using very long episodes only leads to slightly worse performance compared to the same amount of interactions with episodes of length 20. Since fully refueling is not optimal when only actions 0 and 3 are taken and one would expect the agent to initially mostly take action 0, as it yields immediate reward, the agent should seldomly fully refuel in the beginning of training. This way, it should be harder for the agent to discover that action 2 is actually the best without frequent resetting. However, looking at the actual training curves, this effect does not seem to be very important: In most experiments, the agent seems to learn that action 2 is the best rather early, at least averaged over the different random seeds.

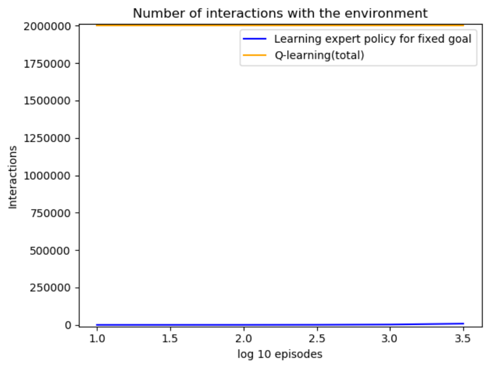
At last, we compare the model learned by our prespecified policy to one learned with the Q-based algorithm based on replacing the targets by term 4.4.1. We set the optimization weights to one for the usual reward and one for reaching full energy and train the Q-based algorithm with the usual parameters and average the difference between the obtained model and a fixed model trained by the SARSA based algorithm 5 with the same parameters but $N = 100$ and $T = 100000$ over 25 episodes. We do the same with $SARSA$ to compare.



(a) Error (b) Error

Figure 21: Errors caused by using SARSA and Multivalue Q-learning. a) The SARSA-based approach converges to the correct F very quickly. b) The Q-learning-based approach has large errors in the beginning, but eventually converges as well.

As figure 21 shows, the approach based on multivalue-Q-learning works, and the error essentially becomes 0 after a thousand episodes of training. As it is often hard to exactly prespecify policies, while specifying goal states can be easier, multivalue-Q-learning seems quite promising. Since it involves significant exploration to find the optimal policy SARSA, perhaps unsurprisingly given the deterministic nature of our MDP, converges to the optimal values a lot faster.



(a) interactions

Figure 22: Interactions needed by Multivalue Q-learning to learn F versus total interactions needed by Q-learning. The number of interactions needed to learn F are miniscule compared with the total interactions Q-learning needs to converge.

In figure 22, we can see that the exploratory overhead added by learning the expert-policy when only the desired end state is known is very manageable in this case. In particular this shows that splitting the exploration by adding subgoals can greatly increase the efficiency of reinforcement learning.

We have shown, that expert Q-learning works in practice and can reduce the amount of necessary interactions with the environment considerably. While we have used a simple MDP for experimental clarity, any MDP with the same S_{learn} and S_{fix} replaced by something more complicated should get the same results, as long as F looks the same and can be reasonable approximated. For example, the refueling process might be arbitrarily complicated and/or dangerous, for example due to involving fuel collection, as long as perfect behavior yields the same results as in the toy example. In that case, the merits of the expert approach are more emphasized. This also yields a nice interpretation for the resetting of the energy at the beginning of each episode, which is important for convergence in this case, because the robot would never see states with high energy in expert-Q-learning without it: When training a robot to be deployed in a crisis, we want the robot to learn how to act given that it has enough fuel and to fetch fuel whenever necessary. However, during training of the robot fuel might be plentiful and there are many humans around, such that it is more efficient and safe to let humans do the refueling. The model-based predictions of the effects of collecting fuel (for example provided by a general crisis-refueling module) allow the robot to still learn accurate Q-values.

6 Related Work

6.1 Options

The presented approach is a special case of the option framework outlined by Sutton, Precup and Singh[12] which formalizes the notion of higher level behavior consisting of a sequence of contextual actions. In the option framework, an agent can not only choose between actions, but also between options that represent temporally or locally limited policies:

Definition 6.1.1. *For a given MDP $M = (S, A, p, r, p_0)$, an option is a tuple $(\mathcal{I}, \pi, \beta)$ with $\mathcal{I} \subset S, \beta : S \to [0, 1]$ and $\pi : S \to \mathcal{P}(A)$.*

The set $\mathcal{I}$ is called the initiation set of the option. An agent can only choose an option o in state s, if $s \in \mathcal{I}(o)$. When the agent chooses option o instead of a basic action $a \in A$, policy $\pi(o)$ will be executed until the option is terminated, which happens with probability $\beta(s')$ after every transition to state s'. This way, an MDP M combined with a set of options O induces a modified MDP similar to the one outlined in section 3.3, except that basic actions are always compatible and therefore never penalized, while options are penalized, if and only if they are selected in a state that is not in their respective initiation set. The dynamics get complicated by the potentially stochastic termination modulated by β. In the setting of section 3.3, we have one option executing policy π for every $\pi \in \Pi$, $\mathcal{I}$ is always equal to S_{fix} and $\beta = \chi_{S_{learn}}$ where χ_U is the characteristic function of a set U. A more convenient way to deal with the predetermined actions while carrying out an option is to consider Semi-Markov Decision Processes:

Definition 6.1.2. *A Semi-Markov Decision Process (SMDP) is the tuple $((S, \Sigma, \mu), A, p, r, p_0)$ for a measure space (S, Σ, μ), an arbitrary set A, $p : S \times \mathbb{N} \times S \times A \to R$, such that $p(\cdot, \cdot|s, a)$ is a probability density on $S \times \mathbb{N}$ for all $s \in S, a \in A$, $r : S \times \mathbb{N} \times A \times S \to \mathbb{R}$, as well as a probability density on S, p_0.*

Given a state-action pair, a SMDP does not just provide a distribution over possible next states, but also over the amount of time steps this transition will take. The γ-discounted return at a time t is then defined as $G_t^\gamma = r_t + \gamma^{t_1} r_{t+t_1} + \gamma^{t_1+t_2} r_{t+t_1+t_2} + ...$ where t_i is the transition time from the $i-1$-th to the i-th state after s_t. Q- and V-Values can now be defined as in the normal MDP case and it is possible to obtain (slightly more complicated) Bellman equations.

Now, if for a set of states S_{noact} in an MDP M, the chosen action neither influences the transitions probabilities, nor the reward, we can reformulate M into an SMDP with a reduced state space $S \setminus S_{noact}$: $p(s', j|s, a) = \sum_{s'' \in S_{noact}} p_{j-1}(s''|s, a) p(s''|s)$ is equal to the probability of transitioning to the next state in which actions matter, s' after j steps (and thus taking $j - 1$ steps where actions do not matter), while $r(s', j, a, s)$ is distributed like the accumulated discounted reward for a transition from s to $s' \in S \setminus S_{noact}$ within j steps. In particular, M_Π from section 3.3 can be reinterpreted as an SMDP on $S_{fix} \cup S_{learn}$ with action space $A \times \Pi$, all transitions from S_{learn}

being instantaneous and transition probabilities from S_{fix}, $p(s', j|s, a)$ equaling $({P_{\pi^k}}^{j-1} E_{\pi^k})_{s',s}$, while the expected rewards for a transition starting in $s \in S_{fix}$ are equal to $\sum_{n=0}^{\infty} \gamma^n {P_{\pi^k}}^n R_{\pi^k}(s)$, because $\sum_{n=0}^{j} {P_{\pi^k}}^n$ denotes the probabilities of staying within S_{fix} for j steps. By splitting the return into the expected immediate rewards in S_{fix} and the expected return after leaving S_{fix} discounted appropriately, we obtain

$$V_\pi = \sum_{j=0}^{\infty} \gamma^j {P_{\pi^k}}^j R_{\pi^k} + \sum_{j=1}^{\infty} \gamma^j {P_{\pi^k}}^{j-1} E_{\pi^k} V_\pi = \sum_{j=0}^{\infty} \gamma^j {P_{\pi^k}}^j (R_{\pi^k} + \gamma E_{\pi^k} V_\pi) = (I - \gamma P_{\pi^k})^{-1} (R_{\pi^k} + \gamma E_{\pi^k} V_\pi)$$

for any state $s \in S_{fix}$ and any policy π on the SMDP induced by simplyfying M_Π. As expected, this is exactly what we obtained for $V_*|_{S_{fix}}$ for $M|_\pi$ after simplifying in the last chapter. In this sense, the Operator T proposed in section 3.3 is a mixture between one and two steps of a dynamic programming update for the Q-function in the induced SMDP. This mixed update allows us to only update Q-values for S_{learn}, whereas a one-step update in the SMDP induced by M_Π or a double update in M_Π would both require explicit learning on S_{fix} as well.

The expected reward of following an option until the random termination time k is then defined as

$$r_s^o = \mathbb{E}_o[r_0 + \gamma r_1 + ... + \gamma^k r_k | o \text{ initiated in } s \text{ at } t]$$

and the time-discounted transition probabilities as

$$p_{ss'}^o = \sum_{k=1}^{\infty} p(s', k|s) \gamma^k.$$

With these, the authors define a single step update for learning correct Q-values in the presence of options:

$$Q_O^*(s, o) = r_o^s + \sum_{s'} p_{ss'}^o \max_{o' \in O} Q_{O_{s'}}^*(s', o'),$$

where the option set $O_{s'}$ contains all options that can be exercised in s', as well as all actions of the base MDP, which are also called primitive options. Here, $p_{ss'}^o$ is the general form of $(I - \gamma P_{\pi^k})^{-1} E_{\pi^k} = \sum_{j=0}^{\infty} \gamma^j P_{\pi^k}^j E_{\pi^k}$ with the drawback that we potentially need to take into account intermediate transitions over the whole state space instead of a subset because of more complicated termination functions β. They also propose an iterative algorithm for learning r_s^o and $p_{ss'}^o$ based on Bellman equations for these quantities, which are a general form of the equation used in algorithm 5. This is a special case of intra-option methods that learn how to evaluate or improve an option, while it is being executed.

In [14], Hauskrecht et al. define macro-actions as local policies $S_i \mapsto A$ for a partition of the state space $S = \bigcup_{i \leq n} S_i$ as in the present text. They then derive a set of equations characterizing the transition and reward models $p_{ss'}^o$ and r_s^o. Using this, they define an abstract MDP on the union of entry and exit nodes of the S_i where actions correspond to macro-actions and transitions happen according to the models. This is solved in order to approximate the globally optimal policy on the base MDP. Macros on S_i can be learned by solving a local MDP on S_i and its periphery (the set of states to which transitions can happen from states in S_i) and treating transitions out of S_{fix} as terminal receiving a seed value $\sigma(a)$ as reward. The seed value $\sigma(s)$ is a proxy for $V_{\pi^*}(s)$. If both values are close to each other, the solution to the abstract MDP using such a macro for every S_i is close to optimal. Lastly, they consider hybrid MDPs that combine the abstract states for some S_i with the base states for others. These are basically equivalent to the SMDP induced by M_Π in section 3.3, besides the fact that they are only defined via the discounted transition model $p^o_{ss'}$ and thus technically do not allow for a clear definition of transition probabilities. This hybrid MDP is then solved by value iteration and no approach to learning it from samples is proposed.

[17] presents a general abstract framework for obtaining models for composite options from the constituent's models in the case where the initialization set is equal to the whole state space. The authors also define a range of bellman operators for policy evaluation and planning with options under several different restrictions. In [18], a modified version of Q-learning is used to learn from experience while treating transitions out of regions as terminal with an additional proxy reward which is updated periodically. Initially this is faster than global Q-learning but seems to converges to a suboptimal policy. Furthermore, no theoretical results on convergence are provided. Meanwhile, Dean and Lin [19] use a decomposition technique from linear programming to iteratively refine local policies and the proxy rewards for regions and prove their approaches convergence. This approach might scale somewhat badly, since it is unclear whether and how linear programming could be combined with function approximation, which seems necessary for very large or continuous state spaces.

Parr [20] notice the linear dependence of the returns of stochastic finite state controllers, a generalized option with a possibly non-Markov policy, on the values at possible states of termination. With this information, they apply SMDP-Q-learning to learn the optimal Q-values for options. However, instead of learning the option models, they suggest to simulate the option behavior whenever it is initialized during training, which might slow training down quite relevantly if some good options take a lot of time to be executed. At last, they discuss how to initialize options by either guessing values for exit nodes or by prescribing behavior. [21] implicitly uses the linear correspondence in conjunction with value iteration, but only for simple navigation tasks with a reward of one for reaching a goal and a reward of zero, else.

In [22], McGovern and Sutton empirically analyse the viability of using macro actions. They find that the choice of macro actions is important when Q-learning over options and actions is used: In a grid world navigation task, macro actions that cause movement towards the edge of the grid world speed up training, when the goal is at an edge and slow down training for more central goals. They find that their macro actions bias exploration towards the edges and increase the speed at which changes in the value function propagate from the edges. Notably, the approach in the present work is different: exploration of S_{fix} happens completely separately while the implicit update of values in S_{fix} makes the propagation of values from states in S_{learn} to states in S_{fix} instantaneous. Jong et al. [23] similarly find that options mostly bias exploration for better or worse and base their criticism of attempts to automatically identify options on this finding: Useful options work by providing a bias towards exploring the right states, but how can an algorithm for option identification that relies on exploration itself identify the correct bias? One possible answer is generalization: sequences of actions that were useful in previous similar settings might be useful now. In an attempt to model human intrinsic motivation, [24] lets the agent gradually build options that lead to so called salient events in the state space: the first time such a state is visited, an option with the previous state as initialization set is created. Then, the initialization set is gradually expanded with states from which transitions to it happen. States associated with options that are worse at reaching their goals state are explored preferentially to improve these options, much like learning children prefer to engage in behaviors they haven't fully mastered yet. Meanwhile, Bacon [25] uses techniques from graph theory to identify suitable subgoals or "bottlenecks" and corresponding options from sample interactions with the environment.

Bowling and Veloso [26] use local MDPs with a reward of one for reaching subgoals, zero else to decompose navigation tasks and find that the choice of subgoals matters. They then proceed to proving optimality bounds based on the distance between the values of different goal states for an optimal policy in the local MDP. Next, [27] uses local MDPs to solve an MDP with a natural hierarchy of subtasks, computing a so called recursively optimal policy. [28] combines this approach with RMAX [29], an algorithm that learns an explicit model for the environment. In [31], the authors use macro actions to deal with cases where a single neural network would have difficulties to represent the global policy. These policies are trained using policy gradient methods on a local MDP as in [14]. Meanwhile, Das et al. [30] hierarchically decompose a MDP by learning a master policy over subgoals as well as parameterized subpolicies to achieve these subgoals. Both levels of the hierarchy are trained by actor-critic policy gradient methods and the resulting policy outperforms non-hierarchical training for a simulated task involving the identification of objects in different rooms.

In [32], Bacon et al. derive a policy-gradient theorem for parametrized options, terminations as well as a parameterized global policy over the options. This outperformed DQN ([33]), a version of Q-learning that uses a neural network, on some Atari games. In order to avoid options getting too short (since the optimal solution does not have to involve any options), they artificially increase the value estimates for all options. A similar regularization method for preventing options from gradually being replaced by primitive actions is suggested in [34]. Jain et al. use similar methods to learn a "safe" policy that induces little variability in the returns in [35].

6.2 Multiple rewards and learning option models

In "Universal Value Functions" [36] Schaul et al. use neural networks to learn goal-parameterized optimal value functions for different navigation tasks. When learned properly, this essentially represents a large set of options that navigate to their respective goals. In [37], Szepesvari et al. decompose the expected cumulative reward when executing an option o into the expected cumulative discounted visit count until option termination $u^o(s,s') = \mathbb{E}_{s,o}[\sum^{T-1} \gamma^k \chi_{\{s_k=s'\}}]$ times the vector of immediate rewards in the state s', $r_{\pi_o}(s')$. They then expand this to general sets of functions of states instead of just indicators and use this to generalize option models to different rewards using linear function approximation of the V function. At last, they propose a way of learning correct value functions for policies over options by combining the previous result with discounted terminal state distributions for the option. Barreto et al. [38] use a similar decomposition, but for the whole value function (which can be seen as a degenerate option model) and for the Q- instead of the V-function. This approach is titled successor features. In particular, their approach allows Q-values for the same policy to be easily generalized to various rewards, given that these rewards are linear functions of the features used in training. The approach used in the present work to approximate the operator F_{π_1} in algorithm 5 can be seen as a special case with the features being the reward signal as well as the indicators of entry states in S_{learn}. In [39], similar ideas are developed in the setting of average reward reinforcement learning, an alternative way of dealing with infinite time horizons that avoids temporal discounting. The authors also notice that the dependence of the optimal policy's values on the weight parameters determining the reward is convex. Successor features are combined with universal value functions as "Universal successor features approximators" in [40] to learn a function that is capable of representing the correct value for different rewards with respect to the optimal policy for varying goals. As in Sutton et al.'s Horde architecture [41], this can be done with high sample efficiency by leveraging off-policy learning, such that multiple policies can be updated at the same time.

6.3 Other approaches

Apart from the potential benefits of simplified transfer learning and streamlined exploration by using partially prespecified policies that adhere to human priors, the idea of options localized in the state space presented here can also be used to avoid costly/risky interaction with parts of the environment during training, either by interrupting the training whenever these states are entered or by prescribing behavior that is known as safe. Another technique to minimize unsafe interactions with the environment is constrained reinforcement learning, where an agent has to learn to keep a cost function below a certain threshold while maximizing its reward. This has recently received increased attention due to the release of Open AI's benchmarking suite safety gym [44], as well as Lagrangian versions of trust-region policy optimization that relevantly outperformed the previously proposed constrained policy optimization method [45]. While these approaches are a lot more flexible when it comes to the structure of the given MDP, they require the careful specification of a cost function and provide little guarantees on the behavior during training. Turchetta et al.[46][47] and Sui et al. [48] model the reward function as a Gaussian process over a continuous state representation and use this to avoid "unsafe" states with low expected reward. Limitations include the need for regularity in the state-reward map and potential myopia: visiting states with low reward might be necessary for transitioning to parts of the state space where returns are usually large.

Another approach to train agents to act safely is imitation learning [49]. In this framework the agent learns to imitate an expert demonstrator's behavior in an MDP without having access to direct rewards. One method of imitation learning is behavioral cloning [50], where a map from states to behavior, i.e. a policy is learned via supervised learning targeting an expert's actual behavior. In [51], this is done without having to observe the expert's actions: Instead, actions are inferred from their effect on the state via a pretrained inverse dynamics model. Instead of behavioral cloning, inverse reinforcement learning [54] [55] can be used. In inverse reinforcement learning (IRL), the expert is assumed to follow a policy that is close to optimal for the MDP's reward and a reward is then inferred from the expert's behavior. Standard reinforcement learning or planning techniques can then be used to imitate the expert. [53] avoids the additional loop by directly learning a policy that induces the same distribution over state-action pairs as the expert, which is dual to IRL. In [52], the authors use similar techniques combined with self-supervised learning to learn to play Atari games with challenging exploration like Montezuma's rage with the help of YouTube videos of experts playing the game and manage to achieve results way above the human average, whereas DQN [33] and the widely used proximal policy optimization [56] are far below the average human for that game. In the training of Deep Mind's recent AlphaStar [57], an agent that reached the 99.8 percentile of human ability in Star Craft 2, a complex real time strategy game, imitation learning was used to initially bias the learning towards "reasonable" regions of the vast policy space. Imitation learning can be seen as complementary to the presented approach: Imitation learning can be used on problematic subtasks or ones that occur in many different problems in order to learn the rest of the task using expert-Q-learning. [58] employs a similar approach combining a hierarchical task structure with behavioral cloning and the option to query experts for feedback on both levels of the hierarchy. This approach also manages to outperform DQN in Montezuma's rage.

7 Discussion and Further Research

We have presented an approach for learning a policy on a part of the state space S_{learn} that responds optimally to a set of expert policies that represent previously learned or safe behavior on the rest of the state space S_{fix}. We have also elementarily and comprehensively proven convergence of the introduced algorithms and verified this in experiments. The employed ideas are closely related to the options framework and ideas from multireward reinforcement learning. More specifically, the main algorithm 6 can be framed as a conditional mixture between two step SMDP Q-learning with an implicit choice of expert policies/options and normal Q-learning on S_{learn}. The implicit option selection restricts the learning to S_{learn} after prelearning on S_{fix}, which reduces the amount of necessary interactions with the environment and is helpful if access to S_{fix} is restricted, costly or unsafe and especially if the same expert policies on S_{fix} can be reused for multiple MDPs that all contain S_{fix}. We have demonstrated, that the approach can require less interactions than normal Q-learning, even if a model for the policy on S_{learn} has to be learned from scratch. Depending on the MDP's size, or complexity in the case of learning with function approximation, not having to learn Q-values for S_{fix} might bring computational advantages.

Since transitions to S_{fix} still have to be simulated, practical applications would either need a way to replace them by simulated transitions or some safety margin, such that the entry states to S_{fix} are still unproblematic. To fully leverage that the algorithm does not need to learn in S_{fix} it is also important that the training environment can easily be reset upon transitions to S_{fix}, or that these are easily reversible. As a potential real world application, one could imagine a robot learning to operate a warehouse to automatically go back to its charging station, when it reaches a certain room, for example one that is currently used by humans, with which the robot cannot yet safely interact. Alternatively, the robot could shut itself off and its position would be reset by a human operator at a convenient time.

The entry states of S_{learn} are of practical importance for another reason: while all presented algorithms are easily extended to the case of a function approximator representing Q-values (sadly without rigorous guarantees on convergence), the model for the values in S_{fix}, F_{π_1} can only be learned via algorithm 5 if the amount of entry states to S_{learn} is small. This means that a successful application of the presented ideas in deep reinforcement learning would require a problem with a small (unidirectional) bottleneck between S_{fix} and S_{learn}. This does not necessarily have to be discrete, as a grid-based discretization might work for sufficiently smooth rewards and dynamics, but still presents a major problem for many potential applications. Exploration is another interesting issue: in the text, we have mostly assumed that we can modify the starting distribution p_0 to ensure that all states in S_{learn} are seen. This might be infeasible for some applications and it might be necessary to choose another decomposition for which S_{learn} is more connected or another approach that allows for easier exploration in those cases.

The main algorithms 4 and 6 are cases of convergent reinforcement learning with variable rewards that depend on the agent's current beliefs about the returns it will obtain (its Q-values).

Similar ideas might be employed for constraining learned behavior in a more elaborate way or to further improve exploration building upon the ideas presented in [24]. It might even be possible to change the MDPs dynamics in certain ways while retaining convergence. Another avenue of further research concerns partial observability: does the approach still work for POMDPs, at least if entry states to S_{learn} are uniquely identifiable from the observations? Then, there is the question how to guarantee convergence to the globally optimal policy. In principle, this works if Π contains all potentially optimal policies on S_{fix}, but saving the model for all of them seems infeasible. Techniques employing neural networks for generalization, similar to universal successor feature approximators, might help with efficient generalization between policies, especially given the somewhat simple structure of F as the pointwise maximum of affine linear functions. For example, the convexity of such functions could be leveraged for more efficient generalization by so called input convex neural networks [59], which are built to represent convex functions for all parameter choices.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction.* MIT press, (2018).

[2] Jaakkola, Tommi, Michael I. Jordan, and Satinder P. Singh. "Convergence of stochastic iterative dynamic programming algorithms." *Advances in neural information processing systems.* (1994).

[3] Szepesvári, Csaba, and Michael L. Littman. "Generalized markov decision processes: Dynamic-programming and reinforcement-learning algorithms." *Proceedings of International Conference of Machine Learning. Vol. 96.* (1996).

[4] Szepesvári, Csaba. "Synthesis of neural networks: the case of cascaded Hebbians." *Technical ReportTR-96-102, Research Group on Artifcial Intelligence, JATE-MTA* (1996).

[5] Blum, Julius R. "Approximation methods which converge with probability one." *The Annals of Mathematical Statistics* 25.2 (1954): 382-386.

[6] Loève, Michel. "On almost sure convergence." *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability.* The Regents of the University of California, (1951).

[7] Meintrup, David, and Stefan Schäffler. *Stochastik: Theorie und Anwendungen.* Springer-Verlag, (2006).

[8] Billingsley, Patrick. *Probability and measure.* John Wiley & Sons, (2008).

[9] Trefethen, Lloyd N., and David Bau III. *Numerical linear algebra.* Vol. 50. Siam, (1997).

[10] Kreyszig, Erwin. *Introductory functional analysis with applications.* Vol. 1. New York: wiley, (1978).

[11] Lin, Long-Ji. "Self-improving reactive agents based on reinforcement learning, planning and teaching." *Machine learning* 8.3-4 (1992): 293-321.

[12] Sutton, Richard S., Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning." *Artificial intelligence* 112.1-2 (1999): 181-211.

[13] Orseau, Laurent, and M. S. Armstrong. "Safely interruptible agents." (2016).

[14] Hauskrecht, Milos, et al. "Hierarchical solution of Markov decision processes using macro-actions." *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence.* Morgan Kaufmann Publishers Inc., (1998).

[15] Puterman, Martin L. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, (2014).

[16] Jiang, Nan. "On Value Functions and the Agent-Environment Boundary." *arXiv preprint* arXiv:1905.13341 (2019).

[17] Silver, David, and Kamil Ciosek. "Compositional planning using optimal option models." *arXiv preprint* arXiv:1206.6473 (2012).

[18] Bernstein, Daniel S., and Shlomo Zilberstein. "Reinforcement learning for weakly-coupled MDPs and an application to planetary rover control." *Sixth European Conference on Planning.* (2014).

[19] Dean, Thomas, and Shieu-Hong Lin. "Decomposition techniques for planning in stochastic domains." *IJCAI.* Vol. 2. (1995).

[20] Parr, Ronald. "A unifying framework for temporal abstraction in stochastic processes." *Proceedings of the Symposium on Abstraction Reformulation and Approximation (SARA-98).* (1998).

[21] Lane, Terran, and Leslie Pack Kaelbling. "Toward hierarchical decomposition for planning in uncertain environments." *Proceedings of the 2001 IJCAI workshop on planning under uncertainty and incomplete information.* (2001).

[22] McGovern, Amy, and Richard S. Sutton. "Macro-actions in reinforcement learning: An empirical analysis." *Computer Science Department Faculty Publication Series* (1998): 15.

[23] Jong, Nicholas K., Todd Hester, and Peter Stone. "The utility of temporal abstraction in reinforcement learning." *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1. International Foundation for Autonomous Agents and Multiagent Systems*, (2008).

[24] Chentanez, Nuttapong, Andrew G. Barto, and Satinder P. Singh. "Intrinsically motivated reinforcement learning." *Advances in neural information processing systems.* (2005).

[25] Bacon, Pierre-Luc. "On the bottleneck concept for options discovery: Theoretical underpinnings and extension in continuous state spaces." *Diss. McGill University Libraries*, (2014).

[26] Bowling, Michael, and Manuela Veloso. "Bounding the suboptimality of reusing subproblems." *IJCAI.* Vol. 99. (1999).

[27] Dietterich, Thomas G. "Hierarchical reinforcement learning with the MAXQ value function decomposition." *Journal of artificial intelligence research* 13 (2000): 227-303.

[28] Jong, Nicholas K., and Peter Stone. "Hierarchical model-based reinforcement learning: R-max+ MAXQ." *Proceedings of the 25th international conference on Machine learning.* ACM, (2008).

[29] Brafman, Ronen I., and Moshe Tennenholtz. "R-max-a general polynomial time algorithm for near-optimal reinforcement learning." *Journal of Machine Learning Research* 3.Oct (2002): 213-231.

[30] Das, Abhishek, et al. "Neural modular control for embodied question answering." *arXiv preprint* arXiv:1810.11181 (2018).

[31] Mankowitz, Daniel J., Timothy A. Mann, and Shie Mannor. "Iterative hierarchical optimization for misspecified problems (IHOMP)." *arXiv preprint* arXiv:1602.03348 (2016).

[32] Bacon, Pierre-Luc, Jean Harb, and Doina Precup. "The option-critic architecture." *Thirty-First AAAI Conference on Artificial Intelligence.* (2017).

[33] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529.

[34] Mankowitz, Daniel J., Timothy A. Mann, and Shie Mannor. "Time regularized interrupting options." *Proceedings of the 31st International Conference on Machine Learning.* PMLR 32(2):1350-1358, (2014).

[35] Jain, Arushi, Khimya Khetarpal, and Doina Precup. "Safe option-critic: Learning safety in the option-critic architecture." *arXiv preprint* arXiv:1807.08060 (2018).

[36] Schaul, Tom, et al. "Universal value function approximators." *International Conference on Machine Learning.* (2015).

[37] Szepesvari, Csaba, et al. "Universal option models." *Advances in Neural Information Processing Systems.* (2014).

[38] Barreto, André, et al. "Successor features for transfer in reinforcement learning." *Advances in neural information processing systems.* (2017).

[39] Mehta, Neville, et al. "Transfer in variable-reward hierarchical reinforcement learning." *Machine Learning* 73.3 (2008): 289.

[40] Borsa, Diana, et al. "Universal successor features approximators." *arXiv preprint* arXiv:1812.07626 (2018).

[41] Sutton, Richard S., et al. "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction." *The 10th International Conference on Autonomous Agents and Multiagent Systems*-Volume 2. International Foundation for Autonomous Agents and Multiagent Systems, (2011).

[42] Van Moffaert, Kristof, Madalina M. Drugan, and Ann Nowé. "Scalarized multi-objective reinforcement learning: Novel design techniques" *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL).* IEEE, (2013).

[43] Yang, Runzhe, Xingyuan Sun, and Karthik Narasimhan. "A Generalized Algorithm for Multi-Objective Reinforcement Learning and Policy Adaptation." *Advances in Neural Information Processing Systems.* 2019.

[44] Ray, Achiam and Amodei "Benchmarking Safe Exploration in Deep Reinforcement Learning" *https://github.com/openai/safety-gym* (2019) accessed 14.01.2020.

[45] Achiam, Joshua, et al. "Constrained policy optimization." *Proceedings of the 34th International Conference on Machine Learning*-Volume 70. JMLR. org, (2017).

[46] Turchetta, Matteo, Felix Berkenkamp, and Andreas Krause. "Safe exploration in finite markov decision processes with gaussian processes." *Advances in Neural Information Processing Systems.* (2016).

[47] Turchetta, Matteo, Felix Berkenkamp, and Andreas Krause. "Safe Exploration for Interactive Machine Learning." *Advances in Neural Information Processing Systems.* (2019).

[48] Sui, Yanan, et al. "Safe exploration for optimization with Gaussian processes." *International Conference on Machine Learning.* (2015).

[49] Schaal, Stefan. "Is imitation learning the route to humanoid robots?." *Trends in cognitive sciences* 3.6 (1999): 233-242.

[50] Pomerleau, Dean A. "Efficient training of artificial neural networks for autonomous navigation." *Neural Computation* 3.1 (1991): 88-97.

[51] Torabi, Faraz, Garrett Warnell, and Peter Stone. "Behavioral cloning from observation." *arXiv preprint* arXiv:1805.01954 (2018).

[52] Aytar, Yusuf, et al. "Playing hard exploration games by watching youtube." *Advances in Neural Information Processing Systems.* (2018).

[53] Ho, Jonathan, and Stefano Ermon. "Generative adversarial imitation learning." *Advances in neural information processing systems.* (2016).

[54] Abbeel, Pieter, and Andrew Y. Ng. "Apprenticeship learning via inverse reinforcement learning." *Proceedings of the twenty-first international conference on Machine learning.* ACM, (2004).

[55] Ziebart, Brian D., et al. "Maximum entropy inverse reinforcement learning." *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence* (2008).

[56] Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint* arXiv:1707.06347 (2017).

[57] Vinyals, Oriol, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning." *Nature* (2019): 1-5.

[58] Le, Hoang M., et al. "Hierarchical imitation and reinforcement learning." *arXiv preprint* arXiv:1803.00590 (2018).

[59] Amos, Brandon, Lei Xu, and J. Zico Kolter. "Input convex neural networks." *Proceedings of the 34th International Conference on Machine Learning*-Volume 70. JMLR. org, (2017).

Appendices

A Proofs

Before we prove theorem 4.2.4, we state it again for the reader's convenience.

Theorem. *Let $P(y|x)$ be probability measures on $\mathbb{R}$ for every $x \in \mathbb{R}$ with corresponding expectations $M(x) := \int_{-\infty}^{\infty} y \, dP(y|x) = x - \theta$ for some constant $\theta \in \mathbb{R}$. Then for a sequence of random variables in $[0,1] : (\alpha_n)_{n\in\mathbb{N}}$, the iteration*

$$x_{n+1} = x_n - \alpha_n y_n$$

with y_n sampled from $P(y_n|x_n)$ converges to θ with probability one for any x_1, as long as there exists a constant C such that

$$\sum_{t\in\mathbb{N}} \alpha_t^2 < C < \infty$$

and

$$\sum_{t\in\mathbb{N}} \alpha_t = \infty,$$

both with probability one and a constant $K \in \mathbb{R}$ such that $|y_n - M(x_n)| \le K < \infty$ with probability one for all $n \in \mathbb{N}$ and $x_n \in \mathbb{R}$.

The idea is to show that both x_n and $x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j)$ converge to some random variables with probability one. From that, one gets convergence of $\sum_{j=1}^{n} \alpha_j M(x_j)$ as well and it is then possible to use that $M(x)$ is large far away from θ combined with the fact that $\sum_{j=1}^{n} \alpha_j$ diverges to obtain a contradiction, if x_n would not converge to θ with probability one.

Following the approach in [5], we begin by providing a simple criterion for the convergence of random variables with probability one, as used in [6].

Lemma A.0.1. *Let x_n be a sequence of random variables on a normed space. Then, x_n converges to zero with probability one, if and only if for every $\epsilon > 0, \delta > 0$ we can find a $T \in \mathbb{N}$ such that $P(\sup_{m\ge T}\|x_m\| \le \varepsilon) > 1 - \delta$.*

Proof. A sequence converges to zero, if and only if for all $\varepsilon > 0$ there exists a $N \in \mathbb{N}$ with $\|x_m\| \le \varepsilon$ for $m > N$. In other words, $x_n(\omega)$ converges to 0, if and only if $\omega \in \bigcup_{N=0}^{\infty} \bigcap_{m=N}^{\infty} \{\omega \in$

$\Omega | \|x_m\| \leq \varepsilon\}$. Because the intersections the union is taken over clearly form an ascending sequence, we can use continuity of measures to show that

$$P(\bigcup_{N=0}^{\infty} \bigcap_{m=N}^{\infty} \{\omega \in \Omega | \|x_m\| \leq \varepsilon\}) = \lim_{N\to\infty} P(\bigcap_{m=N}^{\infty} \{\omega \in \Omega | \|x_m\| \leq \varepsilon\}) = \lim_{N\to\infty} P(\sup_{m\geq N} \{\|x_m\|\} \leq \varepsilon).$$

But this ascending limit is one, if and only if for a fixed δ we can find T such that the expression in the limit is larger than $1-\delta$ for $N = T$. $\square$

In particular we have $P(\sup_{m\geq n} \|x_m\| \leq \varepsilon) > 1-\delta$ for all $n \geq T$ as well, since the limit is taken over a monotonously growing sequence. A similar criterion works without knowing the limit:

Lemma A.0.2. *Let x_n be a sequence of random variables on a Banach space X. Then, x_n converges with probability one, if and only if that for every $\varepsilon > 0, \delta > 0$ we can find a $T \in \mathbb{N}$ such that $P(\sup_{m\geq T} \|x_m - x_{T-1}\| \leq \varepsilon) > 1-\delta$.*

Proof. Since X is a Banach space, x_n converges, if and only if there exists a $N \in \mathbb{N}$ with $\|x_m - x_n\| \leq \varepsilon$ for $m, n \geq N$. Equivalently, one can find an $M \in \mathbb{N}$ with $\|x_{M-1} - x_m\| \leq \varepsilon$ for $m \geq M$: Doing so for $\frac{\varepsilon}{2}$ gives the N from before, since $\|x_n - x_m\| \leq \|x_n - x_{M-1}\| + \|x_{M-1} - x_m\|$, while we can just use $m = N = M-1$ for the other direction. Thus $x_n(\omega)$ converges, if and only if $\omega \in \bigcup_{M=0}^{\infty} \bigcap_{m=M}^{\infty} \{\omega \in \Omega | \|x_m - x_{M-1}\| \leq \varepsilon\}$. Again, the union is taken over an ascending sequence and we get

$$P(\bigcup_{M=0}^{\infty} \bigcap_{m=M}^{\infty} \{\omega \in \Omega | \|x_m - x_{M-1}\| \leq \varepsilon\}) = \lim_{M\to\infty} P(\sup_{m\geq M} \{\|x_m - x_{M-1}\|\} \leq \varepsilon).$$

But this ascending limit is one, if and only if for any fixed δ we can find T such that the expression in the limit ist larger than $1-\delta$ for $M = T$. $\square$

Next, we follow [6] and prove an extension to the Kolmogoroff inequality by P.Lévy that will help us with applying the just proven lemmata to series of random variables:

Lemma A.0.3. *Let x_n be real random variables with finite variance and $u_n = \sum_{i=1}^{n} x_i$ for $n \leq v \in \mathbb{N}$. If $\mathbb{E}[x_n | x_{n-1}, ..., x_1] = 0$ for $1 \leq n \leq v \in \mathbb{N}$ with probability one, we have that*

$$P(\max_{n\leq v} |u_n| > \varepsilon) \leq \frac{1}{\varepsilon^2} \mathbb{E}[u_v^2]$$

Proof. Set $A_i = \{|u_i| > \varepsilon\}$ and $B_i = A_i \backslash \bigcup_{n=0}^{i-1} A_n$. Then,

$$\bigcup_{n\leq v} A_i = \{\max_{n\leq v} |u_n| > \varepsilon\} = \bigcup_{n\leq v} B_i,$$

with the second union being disjoint. Now, slightly abusing notation and writing $\mathbb{E}[X|E]$ as the expectation of a random variable X under the conditional probability measure $P(\cdot|E)$, we get

$$\mathbb{E}[u_v^2|B_i] - \mathbb{E}[u_i^2|B_i] = \mathbb{E}[(u_v - u_i)^2|B_i] + 2\mathbb{E}[(u_v - u_i)u_i|B_i]$$

for $i \leq v$, where the first term is non-negative and the second is zero because

$$\begin{aligned}\mathbb{E}[(u_v - u_i)u_i|B_i] &= \mathbb{E}[\sum_{k=i+1}^{v} x_k u_i|B_i] \\ &= \sum_{k=i+1}^{v} \mathbb{E}[x_k u_i|B_i] \\ &= \sum_{k=i+1}^{v} \frac{\mathbb{E}[x_k u_i \chi_{B_i}]}{P(B_i)} \\ &= \sum_{k=i+1}^{v} \frac{\mathbb{E}[\mathbb{E}[x_k u_i \chi_{B_i}|x_{k-1}, .., x_1]]}{P(B_i)} \\ &= \sum_{k=i+1}^{v} \frac{\mathbb{E}[u_i \chi_{B_i} \mathbb{E}[x_k|x_{k-1}, .., x_1]]}{P(B_i)} \\ &= \sum_{k=i+1}^{v} \frac{\mathbb{E}[u_i \chi_{B_i} \cdot 0]}{P(B_i)} = 0,\end{aligned}$$

since u_i and χ_{B_i} are measurable functions of $\{x_n\}_{n \leq i}$ and therefore measurable by the respectively generated σ-algebra as well as its refinements. All terms are well defined, because the x_i have finite variances and thus finite pairwise covariances by the Cauchy-Schwarz inequality, such that the u_i have finite variances as well as covariances. Thus, we have that

$$\mathbb{E}[u_v^2|B_i] \geq \mathbb{E}[u_i^2|B_i] \geq \varepsilon^2$$

and use it to obtain

$$\mathbb{E}[u_v^2] = \sum_{n \in \mathbb{N}} P(B_n)\mathbb{E}[u_v^2|B_n] \geq \sum_{n \leq v} P(B_n)\mathbb{E}[u_v^2|B_n] \geq \sum_{n \leq v} P(B_n)\varepsilon^2 = \varepsilon^2 P(\max_{n \leq v} |u_n| > \varepsilon)$$

Note: Technically, conditioning on B_i is only well defined if $P(B_i) > 0$, but we only need to consider those cases in the sum, such that this is not a problem. □

The next lemma uses what we just have proven in order to give us a convergence criterion that we will use to control $x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j)$. It is an adaption of Lemma 5.1 from [6] that allows for conditioning on more general histories.

Lemma A.0.4. *Let v_n be a sequence of random variables, such that $\sum_{n=1}^{\infty} \mathbb{E}[(v_n^2)] < \infty$. Let*

$(h_n)_{n\in\mathbb{N}}$ be σ-algebras such that v_n is h_{n+1}-measurable and $h_n \subset h_{n+1}$. Then,

$$\sum_{j=1}^{n} v_j - \mathbb{E}[v_j|h_j]$$

converges to some real random variable v for $n \to \infty$ with probability one.

Proof. Since $\mathbb{E}[v_n^2]$ is finite, $\mathbb{E}[|v_n|]$ is finite as well and it makes sense to talk about conditional expectations of v_n. We set $x_n = v_n - \mathbb{E}[v_n|h_n]$ and obtain

$$\begin{aligned}
&\mathbb{E}[x_n^2]\\
&= \mathbb{E}[(v_n - \mathbb{E}[v_n|h_n])^2]\\
&= \mathbb{E}[\mathbb{E}[(v_n - \mathbb{E}[v_n|h_n])^2|h_n]]\\
&= \mathbb{E}[\mathbb{E}[v_n^2 - \mathbb{E}[v_n|h_n]^2 - 2(v_n - \mathbb{E}[v_n|h_n])\mathbb{E}[v_n|h_n]|h_n]]\\
&= \mathbb{E}[\mathbb{E}[v_n^2 - \mathbb{E}[v_n|h_n]^2|h_n]] - \mathbb{E}[\mathbb{E}[2(v_n - \mathbb{E}[v_n|h_n])\mathbb{E}[v_n|h_n]|h_n]]\\
&= \mathbb{E}[\mathbb{E}[v_n^2 - \mathbb{E}[v_n|h_n]^2|h_n]] - \mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[2(v_n - \mathbb{E}[v_n|h_n])|h_n]]\\
&= \mathbb{E}[\mathbb{E}[v_n^2 - \mathbb{E}[v_n|h_n]^2|h_n]] - \mathbb{E}[\mathbb{E}[v_n|h_n] \cdot 0]\\
&= \mathbb{E}[\mathbb{E}[v_n^2 - \mathbb{E}[v_n|h_n]^2|h_n]]\\
&\leq \mathbb{E}[\mathbb{E}[v_n^2|h_n]]\\
&= \mathbb{E}[v_n^2]
\end{aligned}$$

With probability one, using that $\mathbb{E}[v_n|h_n]$ is measurable with respect to h_n and that $\mathbb{E}[v_n|h_n] = \mathbb{E}[\mathbb{E}[v_n|h_n]|h_n]$. Similarly, for $k < n$, we get

$$\begin{aligned}
\mathbb{E}[x_n x_k] &= \mathbb{E}[v_n v_k - v_k\mathbb{E}[v_n|h_n] - \mathbb{E}[v_k|h_k]v_n + \mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]]\\
&= \mathbb{E}[v_n v_k] - \mathbb{E}[v_k\mathbb{E}[v_n|h_n]] - \mathbb{E}[\mathbb{E}[v_k|h_k]v_n] + \mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]]
\end{aligned}$$

In order to show that this is zero, we will first look at the first two terms:

$$\begin{aligned}
&\mathbb{E}[v_n v_k] - \mathbb{E}[v_k\mathbb{E}[v_n|h_n]]\\
&= \mathbb{E}[\mathbb{E}[v_n v_k|h_n]] - \mathbb{E}[v_k\mathbb{E}[v_n|h_n]]\\
&= \mathbb{E}[v_k\mathbb{E}[v_n|h_n]] - \mathbb{E}[v_k\mathbb{E}[v_n|h_n]]\\
&= 0,
\end{aligned}$$

where we use that since $k < n$, v_k is measurable with respect to $h_n \supset h_{k+1}$ and we can thus pull it out of the conditional expectation. For the second term we get:

$$\begin{aligned}
&\mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]] - \mathbb{E}[\mathbb{E}[v_k|h_k]v_n]\\
&= \mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]] - \mathbb{E}[\mathbb{E}[\mathbb{E}[v_k|h_k]v_n|h_n]]
\end{aligned}$$

$$
\begin{aligned}
&= \mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]] - \mathbb{E}[\mathbb{E}[v_n|h_n]\mathbb{E}[v_k|h_k]] \\
&= 0,
\end{aligned}
$$

since $\mathbb{E}[v_k|h_k]$ is measurable with respect to h_k and thus also with respect to $h_n \supset h_k$. With our premise, $\mathbb{E}[x_n x_k] = 0$ for $k \neq n$ and $\mathbb{E}[x_n^2] \leq \mathbb{E}[v_n^2]$, we get for any $m, n \in \mathbb{N}$

$$
\mathbb{E}[(\sum_{m \leq k \leq n} x_k)^2] = \sum_{m \leq k \leq n} \mathbb{E}[x_k^2] \leq \sum_{m \leq k \leq n} \mathbb{E}[v_k^2].
$$

Now, since $\mathbb{E}[v_k|h_k]$ is measurable with respect to h_n for $k < n$ and all v_k for $k < n$ are measurable with respect to that σ-algebra as well, we obtain that all x_k for $k < n$ are measurable as sums of measurable functions. But this implies that h_n is finer than the σ-algebra generated by $\{x_{n-1}, ..., x_1\}$ and we can apply the tower law to obtain

$$
\begin{aligned}
&\mathbb{E}[x_n|x_{n-1}, ..., x_1] \\
&= \mathbb{E}[v_n|x_{n-1}, ..., x_1] - \mathbb{E}[\mathbb{E}[v_n|h_n]|x_{n-1}, ..., x_1] \\
&= \mathbb{E}[v_n|x_{n-1}, ..., x_1] - \mathbb{E}[v_n|x_{n-1}, ..., x_1] \\
&= 0
\end{aligned}
$$

with probability one. Actually since conditioning on fewer variables can only make a $\sigma-$algebra coarser, we get for any $m < n \in \mathbb{N}$:

$$
\mathbb{E}[x_n|x_{n-1}, ..., x_m] = \mathbb{E}[\mathbb{E}[x_n|x_{n-1}, ..., x_1]|x_{n-1}, ..., x_m] = \mathbb{E}[0|x_{n-1}, ..., x_m] = 0
$$

with probability one. For any $m, n \in \mathbb{N}$, we can now define $u_m^n = \sum_{k=m}^{n+m} x_k$ and apply lemma A.0.3 to obtain

$$
P(\max_{n \leq v} |u_m^n| > \varepsilon) \leq \frac{1}{\varepsilon^2}\mathbb{E}[(u_m^v)^2].
$$

Rewriting now gives us

$$
P(\max_{n \leq v} |\sum_{k=m}^{n+m} x_k| > \varepsilon) \leq \frac{1}{\varepsilon^2}\mathbb{E}[(\sum_{k=m}^{n+v} x_k)^2] \leq \frac{1}{\varepsilon^2}\sum_{k=m}^{n+v} \mathbb{E}[v_k^2],
$$

and for v going to infinity (using continuouity of measure!):

$$
P(\sup_{n \geq m} |\sum_{k=m}^{n} x_k| > \varepsilon) \leq \frac{1}{\varepsilon^2}\sum_{k=m}^{\infty} \mathbb{E}[v_k^2].
$$

For $\epsilon > 0, \delta > 0$, we can always find $T \in \mathbb{N}$ such that $\frac{1}{\varepsilon^2}\sum_{k=T}^{\infty} \mathbb{E}[v_k^2] < \delta$ because of the convergence of the series. This implies

$$P(\sup_{n\geq T}|\sum_{k=0}^{n}x_k-\sum_{k=0}^{T-1}x_k|>\varepsilon)=P(\sup_{n\geq T}|\sum_{k=T}^{n}x_k|>\varepsilon)<\delta$$

and lemma A.0.2 gives us convergence of the series

$$\sum_{j=1}^{n}x_j=\sum_{j=1}^{n}v_j-\mathbb{E}[v_j|h_j]$$

with probability one. □

Now, we are able to prove the convergence of $x_{n+1}+\sum_{j=1}^{n}\alpha_jM(x_j)$ by augmenting the proof in [5] to account for stochastic α_t:

Lemma A.0.5. *For the conditions from theorem 4.2.4, the sequence*

$$x_{n+1}+\sum_{j=1}^{n}\alpha_jM(x_j)$$

converges to a real random variable with probability one.

Proof. First we want note that the random variables y_n, $M(x_n)$ and $\alpha_n(y_n-M(x_n))$ are bounded with probability one (the bound might depend on n!) and in particular the expectations of their absolute values stay finite, such that it makes sense to speak of their conditional expectations. This is because x_1 is bounded as a deterministic variable and $M(x_1)$ is thus bounded, which means that y_1 is bounded because $|y_1-M(x_1)|\leq K$ with probability one. Using $x_{n+1}=x_n-\alpha_ny_n$ with bounded α_n and repeating the argument, we inductively get boundedness of all y_n and $M(x_n)$ and combining this with the boundedness of α_n, $\alpha_n(y_n-M(x_n))$ is bounded as well.

Now, set $v_j:=\alpha_j(y_j-M(x_j))$ for y_j as in 4.2.4 and observe that

$$\mathbb{E}[v_j^2]=\mathbb{E}[\alpha_j^2(y_j-M(x_j))^2]\leq\mathbb{E}[a_j^2]K^2,$$

since both factors are positive and $|y_j-M(x_j)|\leq K$ with probability one. Since $\sum_{n=0}^{\infty}\alpha_n^2<C<\infty$ with probability one, and using positivity of the square terms and monotone convergence, we also get that $\sum_{n=0}^{\infty}\mathbb{E}[\alpha_n^2]=\mathbb{E}[\sum_{n=0}^{\infty}\alpha_n^2]<C$. This yields $\sum_{n=1}^{\infty}\mathbb{E}[(v_n^2)]<CK^2<\infty$. Using this, and that

$$M(x_n)=M(x_1-\sum_{k=0}^{n-1}\alpha_ky_k)$$

is a measurable function of $h_n=\{\alpha_n,...,\alpha_1,y_{n-1},...,y_1\}$ and therefore $v_n=\alpha_n(y_n-M(x_n))$ is h_{n+1}-measurable, we can apply lemma A.0.4 to v_n and h_n in order to see that $\sum_{j=1}^{n}v_j-\mathbb{E}[v_j|h_j]$ converges to some random variable with probability one.

Next, since α_n is h_n−measurable, we have that

$$\mathbb{E}[v_n|h_n] = \mathbb{E}[\alpha_n(y_n - M(x_n))|h_n] = \alpha_n \mathbb{E}[(y_n - M(x_n))|h_n] = \mathbb{E}[\alpha_n|h_n]\mathbb{E}[(y_n - M(x_n))|h_n].$$

In order to see that this is zero, we first note that $x_n = x_1 - \sum_{k=0}^{n-1} \alpha_k y_k$ is a function of h_n, such that $P(y_n|x_n)$ naturally induces a conditional measure of y_n given h_n by setting

$$P(y_n|h_n) = P(y_n|x_n(h_n))$$

which means that

$$\int_{-\infty}^{\infty} y \; dP(y|h_n) = \int_{-\infty}^{\infty} y \; dP(y|x_n) = M(x_n).$$

Now by lemma 2.1.1 property 7), we have that $M(x_n) = E[y_n|h_n]$ and

$$\mathbb{E}[(y_n - M(x_n))|h_n] = \mathbb{E}[y_n|h_n] - \mathbb{E}[\mathbb{E}[y_n|h_n]|h_n] = \mathbb{E}[y_n|h_n] - \mathbb{E}[y_n|h_n] = 0.$$

Therefore, $\mathbb{E}[v_n|h_n]$ is zero for arbitrary $n \in \mathbb{N}$ and

$$\sum_{j=1}^{n} \alpha_j(y_j - M(x_j)) = \sum_{j=1}^{n} v_j = \sum_{j=1}^{n} v_j - \mathbb{E}[v_j|h_n]$$

converges to some random variable with probability one for $n \to \infty$. Using

$$x_{n+1} = x_1 - \sum_{j=1}^{n} \alpha_j y_j,$$

we have

$$x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j) = x_1 - \sum_{j=1}^{n} \alpha_j y_j + \sum_{j=1}^{n} \alpha_j M(x_j) = x_1 - \sum_{j=1}^{n} \alpha_j (y_j - M(x_j))$$

and the statement, since x_1 is a constant and the other term converges with probability one. □

Next we can show that x_n converges. Here, the proof in [5] needs no major modification:

Lemma A.0.6. *For the conditions from theorem 4.2.4 holding, x_n converges with probability one.*

Proof. At first, we show that the sequence $(x_n)_{n\in\mathbb{N}}$ stays bounded: using that

$$|y_n - M(x_n)| \leq K$$

with probability one and $M(x_n) = x_n - \theta$ we obtain that for $x_n \geq \theta + K$ we have that $M(x_n) \geq K$ and therefore

$$y_n \geq 0.$$

For $x_n \leq \theta - K$, we get that $M(x_n) \leq -K$ and accordingly

$$y_n \leq 0$$

with probability one. This means that for $|x_n| \geq |\theta| + K \geq |\theta \pm K|$ we have that $|x_{n+1}| = |x_n - \alpha_n y_n| \leq |x_n|$. On the other hand, for $|x_n| \leq |\theta| + K$,

$$|y_n| \leq |M(x_n)| + K \leq |\theta| + K + |\theta| + K,$$

such that $|x_{n+1}|$ is bounded by at most $3K + 3|\theta|$ with probability one.
Now, assume that there is a nonzero probability of x_n not converging. This means that with probability $p > 0$, ω is such that we have

$$\lim_{n\to\infty} \inf x_n < \lim_{n\to\infty} \sup x_n,$$

while $x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j)$ converges to some $x(\omega) \in \mathbb{R}$ by lemma A.0.5. Fix any such ω, for which the conditions on α from lemma A.0.5 also hold. There are now two cases: either $\limsup x_n > \theta$ or $\limsup x_n \leq \theta$.

We start with the first and fix $a, b \in \mathbb{R}$ with $a > \theta$ and

$$\lim_{n\to\infty} \inf x_n < a < b < \lim_{n\to\infty} \sup x_n.$$

Since α_n has to converge to zero because the sum of squares converges, we can choose $N_1 \in \mathbb{N}$ such that

$$\alpha_n \leq \min\{\frac{1}{3}, \frac{b-a}{6|\theta|}\}$$

for $n > N_1$. On the other hand, we can use the convergence of $x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j)$ to choose $N_2 \in \mathbb{N}$ with

$$|x_m + \sum_{j=1}^{m-1} \alpha_j M(x_j) - x_n - \sum_{j=1}^{n-1} \alpha_j M(x_j)| \leq \frac{b-a}{3}$$

for $m > n > N_2$.

Now, we choose m and n such that $m > n > \max\{N_1, N_2\} := N$, $x_n < a$, $x_m > b$ and such that $n < j < m$ implies $a \leq x_j \leq b$. Because of the choice of a and b above the $\liminf$ and below the $\limsup$, it is possible to fulfill the last condition by starting with an m with $x_m > b$ and an n with $x_n < a$ such that $m > n > N$ and iteratively replacing m and n with smaller/larger values while keeping $n < m$, until the values between x_n and x_m are between a and b. Now because

$$x_m - x_n + \sum_{j=n}^{m-1} \alpha_j M(x_j) \leq |x_m - x_n + \sum_{j=n}^{m-1} \alpha_j M(x_j)| \leq \frac{b-a}{3},$$

and since $x_j \geq a > \theta$ for $n < j < m$, we have that $M(x_j) \geq 0$ for those j and thus

$$x_m - x_n \leq \frac{b-a}{3} - \sum_{j=n}^{m-1} \alpha_j M(x_j) \leq \frac{b-a}{3} - \alpha_n M(x_n).$$

Now, if $\theta < x_n$, the last term of the sum would also become positive and we obtain $x_m - x_n \leq \frac{b-a}{3}$, which contradicts $x_n < a$, $x_m > b$. Thus we suppose that $\theta \geq x_n$ and calculate:

$$|M(x_n)| = |x_n - \theta| \leq |\theta| + |x_n| \leq |\theta| + (|\theta| + |\theta - x_n|) \leq 2|\theta| + (x_m - x_n),$$

using that $x_m > b > a > \theta \geq x_n$. Now we can use this to obtain

$$x_m - x_n \leq \frac{b-a}{3} - \alpha_n M(x_n) \leq \frac{b-a}{3} + \alpha_n(2|\theta| + (x_m - x_n))$$

and thus

$$(1 - \alpha_n)(x_m - x_n) \leq \frac{b-a}{3} + 2\alpha_n|\theta|$$

Together with $\alpha_n \leq \frac{b-a}{3\cdot(2|\theta|)}$ this yields

$$x_m - x_n \leq \frac{2(b-a)}{3(1-\alpha_n)} \leq b - a$$

since $\alpha_n \leq \frac{1}{3}$. This is a contradiction to $x_n < a$, $x_m > b$ and we are done with the case $\limsup x_n > \theta$.

For $\limsup x_n \leq \theta$ we fix $a, b \in \mathbb{R}$ with

$$\lim_{n\to\infty} \inf x_n < a < b < \lim_{n\to\infty} \sup x_n,$$

but do not further condition on a. We find $N_1 \in \mathbb{N}$ such that

$$\alpha_m \leq \min\{\frac{1}{3}, \frac{b-a}{6|\theta|}\}$$

for $m > N_1$ as well as $N_2 > N_1$ with

$$|x_m + \sum_{j=1}^{m-1} \alpha_j M(x_j) - x_n - \sum_{j=1}^{n-1} \alpha_j M(x_j)| < \frac{b-a}{3},$$

this time for $n > m > N_2$.
Next, we choose m and n such that $n > m > N_2$, $x_n < a$, $x_m > b$ and such that $m < j < n$

implies $a \leq x_j \leq b$. This time we have

$$x_m - x_n - \sum_{j=m}^{n-1} \alpha_j M(x_j) \leq |x_m - x_n - \sum_{j=m}^{n-1} \alpha_j M(x_j)| < \frac{b-a}{3},$$

which implies

$$x_m - x_n < \frac{b-a}{3} + \sum_{j=m}^{n-1} \alpha_j M(x_j) < \frac{b-a}{3} + \alpha_m M(x_m),$$

since $M(x_j)$ is negative for all $x_j \leq b < \theta$. If $x_m < \theta$ as well, we obtain the contradiction $x_m - x_n < \frac{b-a}{3}$ and are done. Now suppose, $x_m \geq \theta$:

$$x_m - x_n \leq \frac{b-a}{3} + \alpha_m |M(x_m)| \leq \frac{b-a}{3} + \alpha_m(|\theta| + |x_m|) \leq \frac{b-a}{3} + \alpha_m(|\theta| + |x_m - \theta| + |\theta|).$$

Since $x_m \geq \theta \geq \limsup x_n > a > x_n$, this implies

$$x_m - x_n \leq \frac{b-a}{3} + \alpha_m(|\theta| + (x_m - x_n) + |\theta|).$$

As before, together with $\alpha_m \leq \frac{b-a}{3 \cdot (2|\theta|)}$ this yields the contradiction to $x_n < a$, $x_m > b$:

$$x_m - x_n \leq \frac{2(b-a)}{3(1-\alpha_n)} \leq b - a,$$

using that $\alpha_n \leq \frac{1}{3}$. Therefore, x_n converges pointwise for almost every ω. By setting $x = \lim_{n\to\infty} x_n$ for ω for which this limit exists and zero else, the almost sure limit of x_n, x is in particular measurable as the pointwise limit of $x_n \chi_{\{(x_k)_{k\in\mathbb{N}} \text{ converges}\}}$, since the set $\{(x_k)_{k\in\mathbb{N}} \text{ converges}\}$ can be written as $\bigcap_{n=1}^{\infty} \bigcup_{M=0}^{\infty} \bigcap_{m=M}^{\infty} \{\|x_m - x_{M-1}\| < \frac{1}{n}\}$, where $\{\|x_m - x_{M-1}\| < \frac{1}{n}\}$ is clearly measurable. $\square$

We are now able to prove theorem 4.2.4, again essentially reusing the proof from [5]:

Proof. We already know by lemma A.0.6 that x_n converges to some random variable x with probability one. Now suppose, that $P(x \neq \theta) > 0$: Since the borel-σ-algebra on $\mathbb{R}$ is generated by the open intervals, we can find an open interval (ϵ_1, ϵ_2) with $P(x \in (\epsilon_1, \epsilon_2)) > 0$ and $\theta \notin (\epsilon_1, \epsilon_2)$. Now, we fix $\delta < P(x \in (\epsilon_1, \epsilon_2))$ and $\varepsilon > 0$ such that $\theta \notin (\epsilon_1 - \varepsilon, \epsilon_2 + \varepsilon)$. Then we can find $N \in \mathbb{N}$ such that $P(\sup_{n>N} |x_n - x| \leq \varepsilon) > 1 - \delta$. Then,

$$\begin{aligned}
&P(x_n \in (\epsilon_1 - \varepsilon, \epsilon_2 + \varepsilon) \text{ for all } n > N) \geq P(\{x \in (\epsilon_1, \epsilon_2)\} \cap \{\sup_{n>N} |x_n - x| \leq \varepsilon\} \\
&\geq P(x \in (\epsilon_1, \epsilon_2)) + P(\sup_{n>N} |x_n - x| \leq \varepsilon) - 1 \\
&> \delta + 1 - \delta - 1
\end{aligned}$$

$= 0.$

This means that with positive probability: $|M(x_n)| = |x_n - \theta| \geq d(\theta, (\epsilon_1 - \varepsilon, \epsilon_2 + \varepsilon)) =: c > 0$ for all $n > N$, while the sign of $M(x_n)$ stays constant. In those cases we have that $\sum_{j=1}^{n} \alpha_j M(x_j)$ can only converge, if $\sum_{j=1}^{n} \alpha_j$ does. But, by lemma A.0.6, x_n converges and by lemma A.0.5, $x_{n+1} + \sum_{j=1}^{n} \alpha_j M(x_j)$ does so as well, in both cases with probability one. This means that $\sum_{j=1}^{n} \alpha_j M(x_j)$ converges with probability one as well. On the other hand, $\sum_{t\in\mathbb{N}} \alpha_t = \infty$ with probability one and we have a contradiction. Therefore, x_n converges to θ with probability one. □

In order to prove theorem 4.2.1, we still need to show that the process $\tilde{\delta}_{t+1} = g_t(i)\tilde{\delta}_t(i) + f_t(i)(\|\tilde{\delta}_n\|_\infty + \|\Delta_n\|_\infty)$ from theorem 4.2.3 converges to zero. This follows from the following lemma proven in [3]:

Lemma A.0.7. *Let $x_0 \in \mathbb{R}^d$ be a pointwise non-negative random variable that is bounded in the sup norm by some $C \in \mathbb{R}$ with probability one. Define*

$$x_{t+1}(i) = g_t(i)x_t(i) + f_t(i)(\|x_t\|_\infty + \epsilon_t)$$

for some random processes on $[0,1]^n$, g_t and f_t, as well as random $0 \leq \epsilon_t \in \mathbb{R}$ converging to zero with probability one. Assume that $\lim_{n\to\infty} \prod_{t=k}^{n} g_t = 0$ for all $k \in \mathbb{N}$, where the product is taken pointwise and, that pointwise $f_t \leq \gamma(1 - g_t)$ for some fixed $\gamma \in [0,1)$; both with probability one. Then, x_t converges to 0 with probability one.

In order to prove this, we start with proving that the process without the ϵ_t-term converges to 0. Next, we prove a few general lemmata concerning inequality relations between different sequences, as well as how they relate to convergence. Then, we show that it is enough to show that a modified version of the process that is guaranteed to stay bounded will converge to zero and use this to show convergence.

As said, we start by showing convergence for the case without the last term.[3] states:

Lemma A.0.8. *Let $x_1 \in \mathbb{R}^d$ be a pointwise non-negative random variable that is bounded in the sup norm by some $C \in \mathbb{R}$ with probability one. Define*

$$x_{t+1}(i) = g_t(i)x_t(i) + f_t(i)\|x_t\|_\infty$$

for some random processes on $[0,1]^n$, g_t and f_t. Assume that $\lim_{n\to\infty} \prod_{t=k}^{n} g_t = 0$ for all $k \in \mathbb{N}$, where the product is taken pointwise and, that pointwise $f_t \leq \gamma(1 - g_t)$ for some fixed $\gamma \in [0,1)$; both with probability one. Then, x_t converges to 0 with probability one.

Proof. Fix $\epsilon > 0$ and $\delta > 0$. We want to show convergence using lemma A.0.1. This means we want to find an M such that we have that $\|x_t\|_\infty \leq \epsilon$ for all $t \geq M$ with probability of at least

$1-\delta$. In order to do this, we fix a sequence $(p_n)_{n\in\mathbb{N}}$ with $0 < p_n < 1$ for all $n \in \mathbb{N}$. Now with probability one:

$$x_{t+1}(i) = g_t(i)x_t(i)+f_t(i)\|x_t\|_\infty \leq g_t(i)\|x_t\|_\infty+f_t(i)\|x_t\|_\infty \leq (g_t(i)+\gamma(1-g_t(i)))\|x_t\|_\infty \leq \|x_t\|_\infty.$$

In particular, we obtain $\|x_{t+1}\|_\infty \leq \|x_t\|_\infty \leq C$ for all $t \in \mathbb{N}$. Now, by induction the process y_t defined by

$$y_{t+1}(i) = g_t(i)y_t(i) + \gamma(1-g_t(i))C$$

for $y_1 = x_1$ bounds x_t pointwise from above. On the other hand, one can easily see using telescoping sum that

$$\begin{aligned} y_{t+1}(i) &= \prod_{s=1}^{t} g_s(i)y_1(i) + \sum_{s=1}^{t}\gamma(1-g_s(i))C\prod_{k=s+1}^{t} g_k(i) \\ &= \prod_{s=1}^{t} g_s(i)y_1(i) + \gamma C\sum_{s=1}^{t}(\prod_{k=s+1}^{t} g_k(i) - \prod_{k=s}^{t} g_k(i)) \\ &= \prod_{s=1}^{t} g_s(i)y_1(i) + \gamma C(1-\prod_{s=1}^{t} g_s(i)). \end{aligned}$$

But since $\lim_{n\to\infty}\prod_{t=k}^{n} g_t = 0$ for all k with probability one and $x_1 = y_1$ is pointwise bounded with probability one, y_t converges to γC in every component with probability one. This yields $\limsup_{t\to\infty}\|x_t\|_\infty \leq \limsup_{t\to\infty}\|y_t\|_\infty = \gamma C$. Therefore, with probability one, $\sup_{t\geq n}\|x_t\|_\infty$ converges to some constant $c \leq \gamma C$ and by lemma A.0.1, we can choose $M_1 \in \mathbb{N}$ such that $\sup_{s\geq t}\|x_s\|_\infty \leq \frac{1+\gamma}{2}C > \gamma C$ and thus $\|x_t\|_\infty \leq \frac{1+\gamma}{2}C$ for all $t \geq M_1$ with probability of at least p_1.

Next, assume that for $k \leq n \in \mathbb{N}$ we have found M_k such that

$$\|x_t\|_\infty \leq (\frac{1+\gamma}{2})^k C =: C_k$$

for all $t > M_k$ with probability $\prod_{n=1}^{k} p_n$. Restricting the analysis to events ω for which this inequality holds, by induction the process defined by

$$z_{M_k} = x_{M_k}$$

and

$$z_{t+1}(i) = g_t(i)z_t(i) + \gamma(1-g_t(i))C_k$$

for $t \geq M_k$ bounds x_t from above pointwise, with probability one (conditional on the inequality holding). But this process converges to γC_k with probability one, and thus we can find an index M_{k+1}, such that $\|x_t\|_\infty \leq \frac{1+\gamma}{2}C_k = C_{k+1}$ for all $t \geq M_{k+1}$ with probability of at least

p_{k+1}, again conditional on ω being such that the inequality holds for all previous k. But by the definition of conditional probabilities, this means that $\|x_t\|_\infty \leq \frac{1+\gamma}{2} C_k = C_{k+1}$ for all $t \geq M_{k+1}$ with probability of at least $p_{k+1} \prod_{n=1}^{k} p_n = \prod_{n=1}^{k+1} p_n$. Since $\gamma < 1$, C_k converges to zero and there exists a $N \in \mathbb{N}$ such that $C_k \leq \varepsilon$ for all $k > N$. We can then choose our p_k such that $\prod_{n=1}^{N} p_n \geq 1 - \delta$. Then for $t \geq M_k(p_1, ..., p_n)$ we have $\|x_t\|_\infty \leq \epsilon$ with probability of at least $1 - \delta$. Therefore, x_n converges to zero. □

Next, we show that under certain Lipschitz conditions it is possible to replace a "noise" term in the recursion equation defining a stochastic process by its limit without altering convergence behavior. We start with a simple inequality proven in [4]:

Lemma A.0.9. *Let k, a, b be real sequences and assume*

$$k_{n+1} \leq a_n k_n + b_n$$

as well as $a_n \geq 0$ for all $n \in N$. Then

$$k_{n+1} \leq k_0 \prod_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j.$$

Proof. By assumption we have $k_1 \leq a_0 k_0 + b_0$. We proceed by induction assuming the inequality for n. Then,

$$k_{n+2} \leq a_{n+1}(k_0 \prod_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j) + b_{n+1}$$

with the right hand side equal to

$$k_0 \prod_{i=0}^{n+1} a_i + \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n+1} a_j + b_{n+1} \prod_{j=n+2}^{n+1} a_j$$

with the last product being empty. Thus, by simplyifying we obtain

$$k_{n+2} \leq k_0 \prod_{i=0}^{n+1} a_i + \sum_{i=0}^{n+1} b_i \prod_{j=i+1}^{n+1} a_j$$

and we are done. □

Next, as in [4], we use the inequality to obtain a bound for the difference between the process with and without the noise term:

Lemma A.0.10. *Let X and Y be Banach spaces and U_n be maps from $X \times Y$ to X for all $n \in \mathbb{N}$. Let $y_n \in Y$ denote the elements of a convergent sequence with limit $y \in Y$. Let $\|U_k(x, y) - U_k(x', y)\| \leq L_k^1 \|x - x'\|$ and $\|U_k(x, y) - U_k(x, y')\| \leq L_k^2 \|y - y'\|$ for all $x, x' \in X$ and $y, y' \in Y$*

and constants L_k^1, L_k^2. Define

$$x_{n+1} := U_n(x_n, y)$$

and

$$z_{n+1} := U_n(z_n, y_n)$$

for $x_0 = z_0 \in X$. Then

$$\|z_{n+1} - x_{n+1}\| \leq \sum_{s=0}^{n} \|y_s - y\| L_s^2 \prod_{t=s+1}^{n} L_t^1.$$

Proof. Define $k_n = \|z_n - x_n\|, a_n = L_n^1$ and $b_n = L_n^2 \|y_n - y\|$. Since $x_0 = z_0$, $k_0 = 0$ and it is enough to show $k_{n+1} \leq a_n k_n + b_n$ and then use lemma A.0.9.

$$\begin{aligned} k_{n+1} &= \|z_{n+1} - x_{n+1}\| = \|U_n(z_n, y) - U_n(x_n, y_n)\| \\ &\leq \|U_n(z_n, y) - U_n(x_n, y)\| + \|U_n(x_n, y) - U_n(x_n, y_n)\| \\ &\leq L_k^1 \|z_n - x_n\| + L_k^2 \|y - y_n\| \\ &= L_k^2 \|y - y_n\| + L_k^1 k_n \\ &= b_n + a_n k_n \end{aligned}$$

□

In order to use this bound for proving convergence, we show that by reweighing a converging sequence by a double sequence with certain properties, we get a converging sequence. Following [4] we have:

Lemma A.0.11. *Let $(a_{n,m})_{n,m \in \mathbb{N}, m \leq n}$ be a collection of non-negative numbers, such that $a_{n,m} \xrightarrow{n \to \infty} 0$ for all $m \in \mathbb{N}$ and*

$$\sum_{m=0}^{n} a_{n,m} \xrightarrow{n \to \infty} c$$

for some $c \in \mathbb{R}$. Let Δ_n denote the elements of a sequence converging to $\Delta \in \mathbb{R}$. Then

$$b_n := \sum_{m=0}^{n} a_{n,m} \Delta_m$$

converges to $c\Delta$.

Proof. We set $c_n = c - \sum_{m=0}^{n} a_{n,m}$ and $H_K = \sup_{n \geq K} |\Delta_n - \Delta|$ and first prove a bound:

$$\begin{aligned} |b_n - c\Delta| &= |\sum_{m=0}^{n} a_{n,m} \Delta_m - (c_n + \sum_{m=0}^{n} a_{n,m}) \Delta| \\ &= |\sum_{m=0}^{n} a_{n,m} \Delta_m - \sum_{m=0}^{n} a_{n,m} \Delta - c_n \Delta| \end{aligned}$$

$$\begin{aligned}
&\leq (\sum_{m=0}^{n} a_{n,m}|\Delta_m - \Delta|) + |c_n\Delta| \\
&= (\sum_{m=0}^{K-1} a_{n,m}|\Delta_m - \Delta| + \sum_{m=K}^{n} a_{n,m}|\Delta_m - \Delta|) + |c_n\Delta| \\
&\leq (H_0 \sum_{m=0}^{K-1} a_{n,m} + H_K \sum_{m=K}^{n} a_{n,m}) + |c_n\Delta| \\
&\leq (H_0 \sum_{m=0}^{K-1} a_{n,m} + H_K \sum_{m=0}^{n} a_{n,m}) + |c_n\Delta| \\
&\leq (H_0 \sum_{m=0}^{K-1} a_{n,m} + CH_K) + |c_n\Delta|
\end{aligned}$$

for some constant C, as $\sum_{m=0}^{n} a_{n,m}$ is bounded as a converging sequence. Now to show convergence, fix $\varepsilon > 0$. Because Δ_n converges to Δ, we can choose K with $CH_K \leq \frac{\varepsilon}{2}$. On the other hand, we can choose $N > K$ such that $H_0 \sum_{m=0}^{K-1} a_{n,m} + |c_n\Delta| \leq \frac{\varepsilon}{2}$ for all $n \geq N$, since c_n and $a_{n,m}$ converge to 0 in n and thus $H_0 \sum_{m=0}^{K-1} a_{n,m} + |c_n\Delta|$ converges to 0 for fixed K as a finite sum of sequences converging to 0. Therefore, for all such $n \geq N : |b_n - c\Delta| \leq \varepsilon$ and we have convergence, since ε was chosen arbitrarily. □

Next, as in [4], we combine the two previous results to see that under the right conditions, we can in fact replace a noise term by its limit:

Lemma A.0.12. *Let x_n, y_n, z_n, L_n^1 and L_n^2 be as in lemma A.0.10, but additionally assume that $L_n^1 \leq 1$ for all $n \in \mathbb{N}$. Also assume that there exists $C > 0$ such that $L_n^2 \leq C(1 - L_n^1)$ for all $n \in \mathbb{N}$ and that $\prod_{n=k}^{\infty} L_n^1 = 0$ for all $k \in \mathbb{N}$. Then, $\|z_n - x_n\|$ converges to 0.*

Proof. Define $\Delta_n = \|y_n - y\|$, converging to 0. By lemma A.0.10 we have

$$\|z_{n+1} - x_{n+1}\| \leq \sum_{m=0}^{n} \|y_m - y\| L_m^2 \prod_{t=m+1}^{n} L_t^1 \leq \sum_{m=0}^{n} \Delta_m C(1 - L_m^1) \prod_{t=m+1}^{n} L_t^1.$$

Now, notice that $b_{n,m} := C(1 - L_m^1) \prod_{t=m+1}^{n} L_t^1$ fullfills the requirements on $a_{n,m}$ in lemma A.0.11: firstly since $L_n^1 \leq 1$, $b_{n,m}$ is non-negative. Since $\prod_{t=m+1}^{\infty} L_t^1 = 0$, we have $b_{n,m} \xrightarrow{n\to\infty} 0$. Lastly,

$$\lim_{n\to\infty} \sum_{m=0}^{n} b_{n,m} = C \lim_{n\to\infty} \sum_{m=0}^{n} (1 - L_m^1) \prod_{t=m+1}^{n} L_t^1 = C \lim_{n\to\infty} \sum_{m=0}^{n} (\prod_{t=m+1}^{n} L_t^1 - \prod_{t=m}^{n} L_t^1).$$

After reducing the telescoping sum and plugging in our assumption that $\prod_{n=k}^{\infty} L_n^1 = 0$, we obtain

$$\lim_{n\to\infty} \sum_{m=0}^{n} b_{n,m} = C \lim_{n\to\infty} (\prod_{t=n+1}^{n} L_t^1 - \prod_{t=0}^{n} L_t^1) = C(1 - \lim_{n\to\infty} \prod_{t=0}^{n} L_t^1) = C.$$

Now, we can apply lemma A.0.11 to show that $\sum_{m=0}^{n} \Delta_m C(1 - L_m^1) \prod_{t=m+1}^{n} L_t^1$ converges to $C \cdot 0 = 0$ and therefore $\|z_{n+1} - x_{n+1}\|$ is bounded by a sequence that converges to 0 and converges to 0. ☐

Now, in order to show that it is enough to consider a modified bounded process, we introduce homogeneous processes, of which the considered process is an easy example, as well as two auxillary definitions:

Definition A.0.13. *A stochastic iterative process on* $\mathbb{R}^d$

$$x_{t+1} = G_t(x_t, \epsilon_t)$$

for $x_0 \in \mathbb{R}^d$*, a sequence of real random variables (noise)* ϵ *and random measurable functions* $G_t : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ *is called homogeneous, if each* G_t *is homogeneous. This means that for all* $x \in \mathbb{R}^d, \epsilon \in \mathbb{R}$ *and* $\beta > 0$ *we have:*

$$\beta G_n(x, \epsilon) = G_n(\beta x, \beta \epsilon).$$

For $G := (G_n)_{n \in \mathbb{N}}$ *we also write the sequence* $(x_n)_{n \in \mathbb{N}}$ *generated by such a process with start value* x_0 *and a some random sequence of noise* ϵ *as* $(x_n(G, x_0, \epsilon))_{n \in \mathbb{N}}$.

We say that G *is insensitive to finite perturbations of* ϵ*, if for arbitrary* x_0 *and* ϵ *changing finitely many* ϵ_t *does not affect the convergence of* $(x_n(G, x_0, \epsilon))_{n \in \mathbb{N}}$ *to zero (with probability one, it converges for the changed sequence, if and only if it converges for the original one). We say that* G *is insensitive to scaling of* ϵ *by numbers smaller than one, if for arbitrary* x_0 *and* ϵ *convergence to zero of* $(x_n(G, x_0, \epsilon))_{n \in \mathbb{N}}$ *implies convergence zo zero of the process with* ϵ *replaced by* $c\epsilon$ *for* $0 < c < 1$ *with probability one.*

For a homogeneous process, multiplying the start values and the noise by a positive constant thus yields the original process multiplied by said constant:

Lemma A.0.14. *For a homogeneous process* $(x_n(G, x_0, \epsilon))_{n \in \mathbb{N}}$ *and* $S > 0$*, we have* $Sx_n = z_n$*, for the homogeneous process generated by the same* G *with the noise and start values scaled by* S*,* $(z_n(G, Sx_0, S\epsilon))_{n \in \mathbb{N}}$.

Proof. This is obviously true for $n = 0$. Per induction:

$$\begin{aligned} Sx_{n+1} &= SG_n(x_n, \epsilon_n) \\ &= G_n(Sx_n, S\epsilon_n) \\ &= G_n(z_n, S\epsilon_n) \\ &= z_{n+1} \end{aligned}$$

☐

We now use this to show that we can replace a homogeneous process by a bounded modification and still expect the same convergence behavior towards zero, as done in [3]:

Lemma A.0.15. *Let $(x_n(G, x_0, \epsilon))_{n\in\mathbb{N}}$ be a homogeneous process with G insensitive to finite perturbations of ϵ and to scaling of ϵ by numbers smaller than one. Assume the modified process with $y_0 = x_0$ and*

$$y_{n+1} = \begin{cases} G(y_n, \epsilon_n) \textit{ for } \|G(y_n, \epsilon_n)\|_\infty < T \\ T\frac{G(y_n,\epsilon_n)}{\|G(y_n,\epsilon_n)\|_\infty} \textit{ else} \end{cases}$$

for $T > 0$ converges to zero with probability one. Then x_n converges to zero with probability one as well.

Proof. For arbitrary noise δ and a start value α, we denote the n-th element of the homogeneous process $(x_n(G, \alpha, \delta))_{n\in\mathbb{N}}$ as $x_n(\alpha, \delta)$ and the n-th element of the modified process as $y_n(\alpha, \delta)$. At first, we want to show, that

$$y_n(x_0, \epsilon) = x_n(d_n x_0, c_n\epsilon)$$

for a sequence $d \subset (0, 1]^{\mathbb{N}}$ and a sequence of real sequences c with $c_n \subset (0, 1]^{\mathbb{N}}$ for all $n \in \mathbb{N}$ and $c_n(k) = 1$ for $k \geq n$ where the product $c_n\epsilon$ is to be read pointwise. We start by setting $c_0(i) = 1$ for all $i \in \mathbb{N}$ and $d_0 = 1$ and note that the equation holds for $n = 0$, since $y_0 = x_0$. We proceed by an recursive construction: Assume the equation holds for $n \in \mathbb{N}$. Define

$$S_n = \begin{cases} 1 \text{ for } \|G(y_n, \epsilon_n)\|_\infty < T \\ \frac{T}{\|G(y_n,\epsilon_n)\|_\infty} \text{ else} \end{cases} \leq 1.$$

Then using the equation for n and homogeneity:

$$\begin{aligned} y_{n+1}(x_0, \epsilon) &= S_n G_n(y_n(x_0, \epsilon), \epsilon_n) \\ &= G_n(S_n y_n(x_0, \epsilon), S_n\epsilon_n) \\ &= G_n(S_n x_n(d_n x_0, c_n\epsilon), S_n\epsilon_n). \end{aligned}$$

By lemma A.0.14, we can thus rewrite

$$y_{n+1}(x_0, \epsilon) = G_n(x_n(S_n d_n x_0, S_n c_n\epsilon), S_n\epsilon_n).$$

Defining $d_{n+1} = S_n d_n$ and $c_{n+1}(i) = S_n c_n(i)$ for $i \leq n$; $c_{n+1}(i) = 1$ else, we obtain $y_{n+1}(x_0, \epsilon) = x_{n+1}(d_{n+1}x_0, c_{n+1}\epsilon)$ by noting that both y_{n+1} and x_{n+1} do not depend on ϵ_k for $k > n$.

Now fix $0 < \delta < T$. Since $\lim_{n\to\infty} y_n(x_0, \epsilon) = 0$ by assumption, we can find $M \in \mathbb{N}$ such that for all $n > M$ $P(\|y_n(x_0, \epsilon)\|_\infty < \delta) > 1 - \delta$.

Again, we restrict the further analysis to events ω with $\|y_n(x_0, \epsilon)\|_\infty < \delta$ for all $n > M$: For those, we have $S_n = 1$ for $n > M$, since rescaling leaves the norm of $y_n(x_0, \epsilon)$ at T, and thus because of $\delta < T$ no rescaling happens for $n > M$. Accordingly, c is a constant sequence of sequences for $n > M$. Furthermore, those sequences c_n only have finitely many elements that are not equal to one. For the same reason, d is constant for $n > M$ as well. Now for those n:

$$y_n(x_0, \epsilon) = x_n(d_{M+1}x_0, c_{M+1}\epsilon)$$

and thus $x_n(d_{M+1}x_0, c_{M+1}\epsilon)$ converges to 0. Because G is insensitive to finite perturbations of ϵ, $x_n(d_{M+1}x_0, \epsilon) = d_{M+1}x_n(x_0, \frac{\epsilon}{d_{M+1}})$ converges to zero as well. But $d_{M+1} \leq 1$ by construction and we can use that the process is insensitive to scaling of ϵ by numbers smaller than one to get convergence of $x_n(x_0, \epsilon)$ to zero (for the respective events with measure larger than $1 - \delta$). Since $\delta > 0$ was arbitrary, we actually get convergence of $x_n(x_0, \epsilon)$ to zero with probability one. □

We are now finally able to prove theorem A.0.7, as in [3]:

Proof. We will first show that x fulfills the conditions for lemma A.0.15 and then consider a rescaled version of the process: It is obvious that $G_t(x, \varepsilon) := g_t(i)x + f_t(i)(\|x\|_\infty + \epsilon)$ is homogeneous. In order to show that those G_t are insensitive to finite perturbations of ϵ, assume that for a fixed event ω $\tilde{\epsilon}$ is such that $\tilde{\epsilon}_t \neq \epsilon_t$ for only finitely many $t \in \mathbb{N}$. Define

$$y_{t+1}(i) := g_t(i)y_t(i) + f_t(i)(\|y_t\|_\infty + \tilde{\epsilon}_t)$$

for $y_0 = x_0$. Then,

$$\begin{aligned}\Delta_{t+1}(i) := |x_{t+1}(i) - y_{t+1}(i)| &= |g_t(i)(x_t(i) - y_t(i)) + f_t(i)(\|x_t\|_\infty - \|y_t\|_\infty + \epsilon_t - \tilde{\epsilon}_t)| \\ &\leq g_t(i)\Delta_t(i) + f_t(i)(|\|x_t\|_\infty - \|y_t\|_\infty| + |\epsilon_t - \tilde{\epsilon}_t|) \\ &\leq g_t(i)\Delta_t(i) + f_t(i)(\|\Delta_t\|_\infty + |\epsilon_t - \tilde{\epsilon}_t|).\end{aligned}$$

Since ϵ and $\tilde{\epsilon}$ only differ for finitely many indexes, we get that

$$\Delta_{t+1}(i) \leq g_t(i)\Delta_t(i) + f_t(i)\|\Delta_t\|_\infty$$

for large enough t. By lemma A.0.8, the sequence defined by replacing the inequality with an equality converges to zero, and since it bounds $\Delta_t(i)$, we get that x and y have the same convergence properties towards zero and G is therefore insensitive to finite perturbations of ϵ. Next, we want to show that G is insensitive to scaling of ϵ by numbers smaller than one: Let $0 < c < 1$ be arbitrary and assume that x converges to zero. Then, because $\varepsilon > 0$ by assumption,

$$x_{t+1}(i) = g_t(i)x_t(i) + f_t(i)(\|x_t\|_\infty + \epsilon_t) \geq g_t(i)x_t(i) + f_t(i)(\|x_t\|_\infty + c\epsilon_t),$$

and the process

$$z_{t+1}(i) := g_t(i)z_t(i) + f_t(i)(\|z_t\|_\infty + c\epsilon_t)$$

with $z_0 = x_0$ is bounded in every component by x_t for each $t \in \mathbb{N}$ and thus converges to zero as well, which proves that x is insensitive to scaling of ϵ by numbers smaller than one.

Now, we denote with $\tilde{x}$ the process obtained by rescaling x with bound C playing the role of T in lemma A.0.15 and note, that it is now sufficient to prove that $\tilde{x}$ converges to zero with probability one. In order to prove this, we will largely repeat the proof of lemma A.0.8:
Fix $\epsilon > 0$ and $\delta > 0$. Also fix a sequence $(p_n)_{n\in\mathbb{N}}$ with $0 < p_n < 1$ for all $n \in \mathbb{N}$.

We have that $\|\tilde{x}_t\|_\infty \leq C$ for all $t \in \mathbb{N}$. Now, by induction the process y_t defined by

$$y_{t+1}(i) = g_t(i)y_t(i) + \gamma(1 - g_t(i))(C + \epsilon_t)$$

for $y_1 = x_1$ bounds $\tilde{x_t}$ pointwise from above with probability one.
This time,

$$\begin{aligned} y_{t+1}(i) &= \prod_{s=1}^{t} g_s(i)y_1(i) + \sum_{s=1}^{t} \gamma(1 - g_s(i))(C + \epsilon_t) \prod_{k=s+1}^{t} g_k(i) \\ &= \prod_{s=1}^{t} g_s(i)y_1(i) + \gamma(C + \epsilon_t)\sum_{s=1}^{t}(\prod_{k=s+1}^{t} g_k(i) - \prod_{k=s}^{t} g_k(i)) \\ &= \prod_{s=1}^{t} g_s(i)y_1(i) + \gamma(C + \epsilon_t)(1 - \prod_{s=1}^{t} g_s(i)). \end{aligned}$$

By lemma A.0.12, y converges to γC in every component with probability one: Fix $\omega \in \Omega$ such that all of the probability one conditions in the requirements of the lemma hold and define

$$U_t(x, \epsilon)(i) := g_t(i)x + \gamma(1 - g_t(i))(C + \epsilon_t)$$

Then, using the notation from lemma A.0.10 and interpreting $U_t(.,.)$ as a collection of functions mapping from $\mathbb{R}$ to $\mathbb{R}$ instead of a function from $\mathbb{R}^d$ to $\mathbb{R}^d$, we have $L_t^1(i) = g_t$ and $L_t^2(i) = (1 - g_t)$. Thus the conditions for lemma A.0.12 are trivially fulfilled by our assumptions on g, componentwise and since ϵ_t converges to zero, we get that $y_{t+1} = U_t(y_t, \epsilon_t)$ has the same componentwise limit as

$$\tilde{y}_{t+1} = U_t(\tilde{y}_t, 0) = g_t\tilde{y}_t + \gamma(1 - g_t)C,$$

which was shown to converge to γC in every component in the proof of lemma A.0.8.

This yields $\limsup_{t\to\infty}\|\tilde{x}_t\|_\infty \leq \limsup_{t\to\infty}\|y_t\|_\infty = \gamma C$. Therefore, with probability one, $\sup_{t\geq n}\|\tilde{x}_t\|_\infty$ converges to some constant $c \leq \gamma C$ and by lemma A.0.1, we can choose $M_1 \in \mathbb{N}$ such that

$$\sup_{s\geq t}\|\tilde{x}_s\|_\infty \leq \frac{1+\gamma}{2}C > \gamma C$$

and thus $\|\tilde{x}_t\|_\infty \leq \frac{1+\gamma}{2}C$ for all $t \geq M_1$ with probability of at least p_1.
Next, assume that for $k \leq n \in \mathbb{N}$ we have found M_k such that

$$\|\tilde{x}_t\|_\infty \leq (\frac{1+\gamma}{2})^k C =: C_k$$

for $t \geq M_i$ with probability $\prod_{n=1}^{k} p_n$. Restricting the analysis to events ω for which this inequality holds, by induction the process defined by

$$z_{M_k} = \tilde{x}_{M_k}$$

and

$$z_{t+1}(i) = g_t(i)z_t(i) + \gamma(1 - g_t(i))(C_{i+1} + \epsilon_t)$$

for $t \geq M_i$ bounds x_t from above pointwise, with probability one (conditional on the inequality holding). But this process converges to γC_{k+1} with probability one, and thus we can find an index M_{k+1}, such that $\|\tilde{x}_t\|_\infty \leq \frac{1+\gamma}{2}C_k = C_{k+1}$ for all $t \geq M_{k+1}$ with probability of at least p_{k+1}, again conditional on ω being such that the inequality holds for k. By the definition of conditional probabilities, this means that $\|\tilde{x}_t\|_\infty \leq \frac{1+\gamma}{2}C_k = C_{k+1}$ for all $t \geq M_{k+1}$ with probability of at least $p_{k+1}\prod_{n=1}^{k} p_n = \prod_{n=1}^{k+1} p_n$. Since $\gamma < 1$, C_k converges to zero and there exists a $N \in \mathbb{N}$ such that $C_k \leq \varepsilon$ for all $k > N$. But we can choose our p_k such that $\prod_{n=1}^{N} p_n \geq 1 - \delta$. Then for $t \geq M_k$ we have $\|\tilde{x}_t\|_\infty \leq \epsilon$ with probability of at least $1 - \delta$. □

This gives us the convergence to zero of the process $\tilde{\delta}_{t+1} = g_t(i)\tilde{\delta}_t(i) + f_t(i)(\|\tilde{\delta}_t\|_\infty + \|\Delta_t\|_\infty)$ in theorem 4.2.3 with Δ_t playing the role of ϵ_t and thus completes the proof of that theorem.

B Code

The code is also available at https://github.com/flodorner/Local-Options .

agents.py

```
import numpy as np


class policy_agent_deterministic:
    def __init__(self, policy, num_act):
        # Implements a simple agent that carries out a determinstic policy.
        # policy is a dict that maps states to actions. If the policy is not defined, the agent acts randomly.
        self.policy = policy
        self.num_act = num_act

    def act(self, obs):
        try:
            return self.policy[obs]
        except:
            return np.random.choice(np.arange(self.num_act))


class policy_agent:
    # Implements a simple agent that carries out a policy.
    # policy is a dict that maps states to distributions over actions (in form of a list of probabilities).
    def __init__(self, policy, num_act):
        self.policy = policy
        self.num_act = num_act

    def act(self, obs):
        return np.random.choice(np.arange(self.num_act), p=self.policy[obs])
```

```
import numpy as np
from copy import deepcopy
import random
from functions import rand_argmax
from agents import policy_agent
from wrapper import partial_env_A
from functions import one_hot


def Tabular_q(env, episodes, num_act, episode_length=np.inf, epsilon=0.05,
              alpha=lambda v, t: 0.1, gamma=0.99, eval_interval=np.inf, Qs=None, init=0, soft_end=False,
              Q_trafo=lambda x: x):
    # Q-lerning. Returns Q-values as a dict.
    # Alpha is a map from visit count and the elapsed time to the learning rate. eval_interval determines
    # after how many episodes the greedy policy is evaluated and the return printed. Qs allows for the initialization
    # of Q-values with a dictionary. If Qs is None, init allows for constant initialization at the value init.
    # soft end determines, how terminal states are treated. If soft_end is true, transitions to terminal states still
    # update on the Q-value of the next state. Q_trafo is the scalarization function that determines the action
    # selection in the multi-objective case.
    vs = {}
    if Qs is None:
        Qs = {}
    else:
        Qs = deepcopy(Qs)
    for i in range(episodes):
        obs_new = env.reset()
        if obs_new not in Qs.keys():
            Qs[obs_new] = [init for i in range(num_act)]
        if obs_new not in vs.keys():
            vs[obs_new] = [init for i in range(num_act)]
        done = False
        t = 0
        while done is False and t < episode_length:
            if obs_new not in Qs.keys():
                Qs[obs_new] = [init for i in range(num_act)]
            if obs_new not in vs.keys():
                vs[obs_new] = [init for i in range(num_act)]

            if np.random.uniform() > epsilon:
                act_new = rand_argmax(Q_trafo(Qs[obs_new]))
            else:
                act_new = np.random.choice(np.arange(num_act))

            if t > 0:
                error = (rew + gamma * Qs[obs_new][np.argmax(Q_trafo(Qs[obs_new]))] - Qs[obs][act])
                Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                vs[obs][act] = vs[obs][act] + 1

            obs = obs_new
            act = act_new

            if hasattr(env, 'dynamic') and env.dynamic is True:
                obs_new, rew, done, _ = env.step(act, Qs)
            else:
                obs_new, rew, done, _ = env.step(act)

            if done is True:
                if soft_end is False:
                    error = (rew - Qs[obs][act])
                    Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                    vs[obs][act] = vs[obs][act] + 1
                else:
                    if obs_new not in Qs.keys():
                        Qs[obs_new] = [init for i in range(num_act)]
                    error = (rew + gamma * Qs[obs_new][np.argmax(Q_trafo(Qs[obs_new]))] - Qs[obs][act])
                    Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                    vs[obs][act] = vs[obs][act] + 1

            t = t + 1

        if i % eval_interval == (-1) % eval_interval:
            obs = env.reset()
            total = 0
            if obs not in Qs.keys():
                Qs[obs] = [init for i in range(num_act)]
            done = False
            s = 0
            while done is False and s < episode_length:
                act = rand_argmax(Q_trafo(Qs[obs]))
                obs_new, rew, done, _ = env.step(act)
                if obs_new not in Qs.keys():
                    Qs[obs_new] = [init for i in range(num_act)]
                obs = obs_new
                total = total + rew
                s = s + 1
            print(i, total)
    return Qs


def SARSA(env, episodes, num_act, policy_agent, episode_length=np.inf, epsilon=0.05,
          alpha=lambda v, t: 0.1, gamma=0.99, Qs=None, init=0, soft_end=False):
    # SARSA. Returns Q-values as a dict.
    # Alpha is a map from visit count and the elapsed time to the learning rate. Qs allows for the initialization
    # of Q-values with a dictionary. If Qs is None, init allows for constant initialization at the value init.
    # soft end determines, how terminal states are treated. If soft_end is true, transitions to terminal states still
    # update on the Q-value of the next state.
    vs = {}
    if Qs is None:
        Qs = {}
    else:
        Qs = deepcopy(Qs)
    for i in range(episodes):
        obs_new = env.reset()
        done = False
        t = 0
        while done is False and t < episode_length:
            if obs_new not in Qs.keys():
                Qs[obs_new] = [init for i in range(num_act)]
            if obs_new not in vs.keys():
                vs[obs_new] = [0 for i in range(num_act)]

            resample = False

            if np.random.uniform() > epsilon:
                act_new = policy_agent.act(obs_new)
            else:
                act_new = np.random.choice(np.arange(num_act))
                resample = True

            if t > 0:
                if resample == True:
                    act_target = policy_agent.act(obs_new)
                else:
                    act_target = act_new

                error = (rew + gamma * Qs[obs_new][act_target] - Qs[obs][act])
                Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                vs[obs][act] = vs[obs][act] + 1

            obs = obs_new
            act = act_new
            obs_new, rew, done, _ = env.step(act)
            if done is True:
                if soft_end is False:
                    error = (rew - Qs[obs][act])
                    Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                    vs[obs][act] = vs[obs][act] + 1
                else:
                    act_target = policy_agent.act(obs_new)
                    error = (rew + gamma * Qs[obs_new][act_target] - Qs[obs][act])
                    Qs[obs][act] = Qs[obs][act] + alpha(vs[obs][act], t) * error
                    vs[obs][act] = vs[obs][act] + 1
            t = t + 1
    return Qs


def Tdzero(env, episodes, num_act, policy_agent, episode_length=np.inf, epsilon=0.05,
           alpha=lambda v, t: 0.1, gamma=0.99, Vs=None, init=0, soft_end=False):
    # Td(0). Returns V-values as a dict.
    # Alpha is a map from visit count and the elapsed time to the learning rate. Vs allows for the initialization
    # of V-values with a dictionary. If Vs is None, init allows for constant initialization at the value init.
    # soft end determines, how terminal states are treated. If soft_end is true, transitions to terminal states still
    # update on the Q-value of the next state.
    vs = {}
    if Vs is None:
        Vs = {}
    else:
        Vs = deepcopy(Vs)
    for i in range(episodes):
        obs_new = env.reset()
        done = False
        resample = False
        t = 0
        while done is False and t < episode_length:
            if obs_new not in Vs.keys():
                Vs[obs_new] = init
            if obs_new not in vs.keys():
                vs[obs_new] = 0

            if t > 0:
                if resample is False:
                    error = (rew + gamma * Vs[obs_new] - Vs[obs])
                    Vs[obs] = Vs[obs] + alpha(vs[obs], t) * error
                    vs[obs] = vs[obs] + 1
                else:
                    resample = False

            if np.random.uniform() > epsilon:
                act_new = policy_agent.act(obs_new)
            else:
                act_new = np.random.choice(np.arange(num_act))
                resample = True

            obs = obs_new
            act = act_new

            obs_new, rew, done, _ = env.step(act)

            if done is True:
                if soft_end is False:
                    error = (rew - Vs[obs])
                    Vs[obs] = Vs[obs] + alpha(vs[obs], t) * error
                    vs[obs] = vs[obs] + 1
                else:
                    error = (rew + gamma * Vs[obs_new] - Vs[obs][act])
                    Vs[obs] = Vs[obs] + alpha(vs[obs], t) * error
                    vs[obs] = vs[obs] + 1
            t = t + 1
    return Vs


def get_F(env, episodes, num_act, policy, entries, exits, c=1, episode_length=np.inf, epsilon=0.05,
          alpha=lambda x, y: 0.1, gamma=0.99, evaluation="SARSA", re_evaluation_factor=0.25):
    # Returns the affinely linear operator F (the local option model) for a policy on a subset of the state space
    # defined by its entry and exits states under that policy, using the indirect method.
    # If evaluation is "SARSA", SARSA is used to approximate F, else td(0). c determines the bonus reward for exiting
    # that is used to calculate F. The larger c, the more accurate the method. re_evaluation_factor specifies, how many
    # episodes are to spend on learning with bonus rewards after the initialization without bonus rewards.

    if isinstance(next(iter(policy.values())), int):
        policy = {key: one_hot(policy[key], num_act) for key in policy}
        # entries to A, exits out of A
    B = np.zeros(len(entries))
    W = np.zeros((len(entries), len(exits)))
    x = np.zeros(len(exits))

    env_x = partial_env_A(env, entries, exits, values=x, gamma=gamma)

    if evaluation is "SARSA":
        Qs = SARSA(env_x, episodes, num_act, policy_agent(policy, num_act), episode_length=episode_length,
                   epsilon=epsilon,
                   alpha=alpha, gamma=gamma)
        for i, entry in enumerate(entries):
            Q = np.array(Qs[entry])
            V = np.average(Q, weights=policy[entry], axis=0)
            B[i] = V
    else:
        Vs = Tdzero(env_x, episodes, num_act, policy_agent(policy, num_act), episode_length=episode_length,
                    epsilon=epsilon,
                    alpha=alpha, gamma=gamma)
        for i, entry in enumerate(entries):
            V = Vs[entry]
            B[i] = V

    for k in range(len(exits)):
        x[k] = c
        x[:k] = 0
        env_x = partial_env_A(env, entries, exits, values=x, gamma=gamma)
        if evaluation is "SARSA":
            Qs_new = SARSA(env_x, int(episodes * re_evaluation_factor), num_act, policy_agent(policy, num_act),
                           episode_length=episode_length, epsilon=epsilon,
                           alpha=alpha, gamma=gamma, Qs=Qs)
            for i, entry in enumerate(entries):
                Q = np.array(Qs_new[entry])
                V = np.average(Q, weights=policy[entry], axis=0)
                V_pred = B[i]
                W[i, k] = (V - V_pred) / c

        else:
            Vs_new = Tdzero(env_x, int(episodes * re_evaluation_factor), num_act, policy_agent(policy, num_act),
                            episode_length=episode_length, epsilon=epsilon,
                            alpha=alpha, gamma=gamma, Vs=Vs)
            for i, entry in enumerate(entries):
                V = Vs_new[entry]
                V_pred = B[i]
                W[i, k] = (V - V_pred) / c

    return lambda v, i: B[i] + sum([v[j] * W[i, j] for j in range(len(v))]), W, B


def nested_key_else_zero(d, i, j):
    try:
        return d[i][j]
    except:
        return 0


def learn_matrices(env, episodes, policy_agent, entries, exits, episode_length=np.inf, gamma=0.99):
    # Returns the affinely linear operator F (the local option model) for a policy on a subset of the state space
    # defined by its entry and exits states under that policy, by approximating a transition model and direct
    # calculation.
    env_x = partial_env_A(env, entries, exits, values=np.zeros(len(exits)))
    reward_dict = {}
    transition_dict = {}
    transition_dict_terminal = {}
    for i in range(episodes):
        obs = env.reset()
        done = False
        t = 0
        while done is False and t < episode_length:
            t = t + 1
            act = policy_agent.act(obs)
            obs_new, rew, done, _ = env_x.step(act)
            try:
                s = sum([transition_dict[obs][key] for key in transition_dict[obs]])
                reward_dict[obs] = rew / s + (s - 1) * reward_dict[obs] / s
            except:
                reward_dict[obs] = rew
            if done is False:
                try:
                    transition_dict[obs][obs_new] = transition_dict[obs][obs_new] + 1
                except:
                    try:
                        transition_dict[obs][obs_new] = 1
                    except:
                        transition_dict[obs] = {}
                        transition_dict[obs][obs_new] = 1
            else:
                try:
                    transition_dict_terminal[obs][obs_new] = transition_dict_terminal[obs][obs_new] + 1
                except:
                    try:
                        transition_dict_terminal[obs][obs_new] = 1
                    except:
                        transition_dict_terminal[obs] = {}
                        transition_dict_terminal[obs][obs_new] = 1

            obs = obs_new

    index_array = {i: j for i, j in enumerate(reward_dict)}
```

```
R = np.array([reward_dict[index_array[i]] for i in range(len(index_array))])

P = []
E = []
for j in range(len(index_array)):
    e = [nested_key_else_zero(transition_dict_terminal, index_array[j], exits[i]) for i in range(len(exits))]
    p = [nested_key_else_zero(transition_dict, index_array[j], index_array[i]) for i in range(len(index_array))]
    z = (sum(p) + sum(e))
    p = np.array(p) / z
    e = np.array(e) / z
    P.append(p)
    E.append(e)
P = np.array(P)
E = np.array(E)

B = np.matmul(np.linalg.inv((np.eye(P.shape[0]) - gamma * P)), R)
W = np.matmul(np.linalg.inv((np.eye(P.shape[0]) - gamma * P)), gamma * E)

indexes = []
for key in index_array:
    if index_array[key] in entries:
        indexes.append(key)
indexes.sort()

indexes = np.array(indexes)
B = B[indexes]
W = W[indexes]

# selection !!!

return lambda v, i: B[i] + sum([v[j] * W[i, j] for j in range(len(v))]), W, B
```

environments.py

```python
import numpy as np


def generate_rewards(n, actions):
    #generates a reward model by uniformly sampling rewards for transitions in an n-dimensional state space.
    return np.random.uniform(size=(n, actions, n))


def generate_transitions(n, actions):
    # generates a transition model by uniformly sampling n-dimensional vectors for each action in every one of n states
    # and normalizing.
    M = np.zeros((actions, n, n))
    for act in range(actions):
        for start in range(n):
            p = np.random.uniform(size=(n))
            p = p / sum(p)
            M[act][start] = p
    return M


class MDP:
    # Implements and MDP from a transition model M, a reward model R and a start distribution ps. If ps is a number,
    # this is interpreted as a delta-distribution supported at that number.
    def __init__(self, M, R, ps=0):
        self.episodes = 0
        self.M = M
        self.R = R
        self.ps = ps

    def reset(self):
        if isinstance(self.ps, int):
            self.state = self.ps
        else:
            self.state = np.random.choice(np.arange(len(self.ps)), p=self.ps)
        self.episodes += 1
        return self.state

    def step(self, act):
        new_state = np.random.choice(np.arange(len(self.M[act])), p=self.M[act][self.state])
        reward = self.R[new_state][act][self.state]
        self.state = new_state

        done = False

        # detect terminal?!

        return self.state, reward, done, None

    def uniform_policy_operator(self, start_B, gamma=0.99):

        P = np.mean(self.M, axis=0)
        R = np.mean(self.R, axis=1)
        R = np.diagonal(np.matmul(P, R))
        R = R[:start_B]

        E = P[:start_B, start_B:]
        P = P[:start_B, :start_B]

        """
        print( np.linalg.norm( np.linalg.inv( np.eye(P.shape[0]) - 0.99*P ),ord=np.inf ) )
        print(np.linalg.norm(np.eye(P.shape[0]) -
                             np.matmul(
                                 np.linalg.inv( np.eye(P.shape[0]) - 0.99*P ),
                                                                ( np.eye(P.shape[0]) - 0.99*P )),
                            ord=np.inf)*np.linalg.norm(np.eye(P.shape[0]) - 0.99*P,ord=np.inf))
        """

        B_True = np.matmul(np.linalg.inv((np.eye(P.shape[0]) - gamma * P)), R)
        W_True = np.matmul(np.linalg.inv((np.eye(P.shape[0]) - gamma * P)), gamma * E)
        return W_True, B_True


def random_MDP(n, acts, B_start):
    #Generates a random Mdp and calculates the option model for a uniform policy on the states between 0 and B_start-1
    M = generate_transitions(n, acts)
    R = generate_rewards(n, acts)
    env = MDP(M, R)
    W, B = env.uniform_policy_operator(B_start)
    return env, W, B


class robot():
    #The robot MDP (see readme for a graphical description)
    def __init__(self):
        self.state = (16, 0, 0)
        self.episodes = 0
        self.steps = 0

    def reset(self):
        self.state = (16, 0, 0)
        self.episodes += 1
        return self.state

    def step(self, act):
        self.steps += 1
        rew = 0
        if act == 0:
            if self.state[0] > 1:
                self.state = (self.state[0] - 1, self.state[1], self.state[2])
                rew += 1
            elif self.state[0] > 0:
                None
            else:
                # stop charging
                self.state = (-self.state[0], self.state[1], self.state[2])
        if act == 1:
            if self.state[0] > 4:
                self.state = (self.state[0] - 5, self.state[1] + 1, self.state[2])
                if self.state[1] > 1:
                    rew += 10
                    self.state = (self.state[0], self.state[1] - 1, self.state[2])
            elif self.state[0] > 0:
                None
            else:
                # stop charging
                self.state = (-self.state[0], self.state[1], self.state[2])
        if act == 2:
            if self.state[0] > 4:
                self.state = (self.state[0] - 5, self.state[1], self.state[2] + 1)
                if self.state[2] == 3:
                    rew += 100
            elif self.state[0] > 0:
                None
            else:
                # stop charging
                self.state = (-self.state[0], self.state[1], self.state[2])
        # Implement this consistently!!!
        if act == 3:
            if self.state[0] > 0 or (self.state[0]==0 and sum([self.state[1],self.state[2]])>0):
                self.state = (-self.state[0], 0, 0)
            elif self.state[0] < -11:
                self.state = (self.state[0] - 3, 0, 0)
                self.state = (max((self.state[0], -16)), 0, 0)
            elif self.state[0] < -1:
                self.state = (self.state[0] - 4, 0, 0)
            else:
                self.state = (self.state[0] - 9, 0, 0)
        return self.state, rew, False, None
```

functions.py

```
import numpy as np
import random


def rand_argmax(x):
    # Returns an index that is randomly sampled from the indices with maximial entries in an array.
    return random.choice(np.argwhere(x == np.max(x)))[0]


def Qstar_to_values(Qs):
    #Takes a dict of Q* values and returns a dict of V* values.
    out = {}
    for key in Qs:
        out[key] = np.max(Qs[key])
    return out


def Qs_to_policy(Qs, Q_trafo=lambda x: x):
    # Takes a dict of Q values and returns the greedy policy as dict
    policy = {}
    for key in Qs:
        policy[key] = np.argmax(Q_trafo(Qs[key]))
    return policy


def one_hot(i, n):
    # Takes a number i and a length n and returns the one-hot encoding for the number of the length
    a = np.zeros(n)
    a[i] = 1
    return a

def Multi_Qs_to_F(policy, Qs, exits):
    # Returns the operator F, given a policy, Q-values and a list of exit states.
    coeffs = []
    for entry in exits:
        try:
            Q = np.array(Qs[entry])
        except:
            Q = [[0 for i in exits] for i in range(len(Qs[list(Qs.keys())[0]]))]
        try:
            V = np.average(Q, weights=policy[entry], axis=0)
        except:
            V = Q[policy[entry]]
        if np.all(V == 0):
            V = [0 for i in exits]
        coeffs.append(V)

    coeffs = np.array(coeffs)
   W = coeffs[:, 1:]
   B = coeffs[:, 0]
    return lambda v, i: coeffs[i][0] + sum([v[j] * coeffs[i][j + 1] for j in range(len(coeffs[i]) - 1)]), W, B


def Multi_Vs_to_F(policy, Vs, exits):
    # Returns the operator F, given a policy, V-values and a list of exit states.
    coeffs = []
    for entry in exits:
       V = np.array(Vs[entry])
        coeffs.append(V)

    coeffs = np.array(coeffs)
   W = coeffs[:, 1:]
   B = coeffs[:, 0]
    return lambda v, i: coeffs[i][0] + sum([v[j] * coeffs[i][j + 1] for j in range(len(coeffs[i]) - 1)]), W, B
```

```
import numpy as np
import scipy.stats as stat
from matplotlib import pyplot as plt


def mean_confidence_interval(data, confidence=0.95):
    # Computes confidence interval (assuming data is normally distributed)
    a = 1.0 * np.array(data)
    n = len(a)
    m, se = np.mean(a), stat.sem(a)
    h = se * stat.t.ppf((1 + confidence) / 2., n - 1)
    return m, m - h, m + h


def compareplot(axis, arrays, names, colors, title="", xlabel="x",
                ylabel="y", ylim=(0, 10), save=False):
    # gets an array as x_axis and a list of arrays of arrays as entries. Calculates the 95% confidence interval assuming
    # normality over the second axis of the array and plots all entries with confidence bands.
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    if ylim is not None:
        ax.set_ylim(ylim[0], ylim[1])
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
    ax.set_title(title)

    for i, array in enumerate(arrays):
        means, upper, lower = [], [], []
        for item in array:
            m, mp, mm = mean_confidence_interval(item)
            means.append(m)
            upper.append(mm)
            lower.append(mp)
        ax.plot(axis, means, colors[i])
        ax.fill_between(axis, lower, upper, facecolor=colors[i], interpolate=True, alpha=0.2, label="_nolegend_")
    ax.legend(names)
    if save is False:
        plt.show()
    else:
        plt.savefig(save)


def compareplot_Q(axis, arrays, names, colors, title="", xlabel="x",
                ylabel="y", ylim=(0, 10), save=False):
    # gets an array as x_axis and a list of arrays of arrays as entries. Calculates the 95% confidence interval assuming
    # normality over the second axis of the array and plots all entries with confidence bands.
    # Also plots the true Q-values for the initial state of the robot MDP.
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    if ylim is not None:
        ax.set_ylim(ylim[0], ylim[1])
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
    ax.set_title(title)

    for i, array in enumerate(arrays):
        means, upper, lower = [], [], []
        for item in array:
            m, mp, mm = mean_confidence_interval(item)
            means.append(m)
            upper.append(mm)
            lower.append(mp)
        ax.plot(axis, means, colors[i])
        ax.fill_between(axis, lower, upper, facecolor=colors[i], interpolate=True, alpha=0.2, label="_nolegend_")
    ax.legend(names)

    ax.plot(axis, [129 for i in axis], "b--")
    ax.plot(axis, [93.3 for i in axis], "--",color="orange")
    ax.plot(axis, [142.2 for i in axis], "r--")
    ax.plot(axis, [115.2 for i in axis], "k--")
    if save is False:
        plt.show()
    else:
        plt.savefig(save)
```

wrapper.py

```
import numpy as np
import random


class partial_env_A:
    def __init__(self, env, entries, exits, values=None, gamma=0.99, track_exits=False, randomstart=True):
        # Implements a local MDP with single on the part of the state space confined by entries and exits.
        # values specifies the additional rewards that are obtained by reaching each of the exit states.
        # If track_exits is True, the distribution of exits is saved as a parameter and can be used to initialize
        # the complementary local MDP on the rest of the state space. If randomstart is True, the start state is sampled
        # from the entries. This requires that the original MDP can be reset to this state.
        self.env = env

        self.entries = entries
        self.exits = exits

        self.gamma = gamma
        self.dynamic = False
        self.track_exits = track_exits
        if track_exits is True:
            self.exit_distribution = {}

        if values is None:
            self.values = [0 for i in range(len(self.exits))]
        else:
            self.values = values

    def reset(self):
        self.env.reset()
        if self.randomstart:
            self.env.state = random.choice(self.entries)
        if self.track_exits is True:
            self.last_entry = self.env.state
        return self.env.state

    def step(self, act, Qs=None):
        obs, rew, done, _ = self.env.step(act)
        if obs in self.exits:
            i = self.exits.index(obs)
            done = True
            rew = rew + self.gamma * self.values[i]
            if self.track_exits is True:
                try:
                    self.exit_distribution[self.last_entry][i] = self.exit_distribution[self.last_entry][i] + 1
                except:
                    self.exit_distribution[self.last_entry] = np.zeros(len(self.exits))
                    self.exit_distribution[self.last_entry][i] = self.exit_distribution[self.last_entry][i] + 1

        return obs, rew, done, None


class partial_env_A_multireward:
    def __init__(self, env, entries, exits, track_exits=False, randomstart=True, gamma=0.99):
        # Implements a local MDP with multiple rewards on the part of the state space confined by entries and exits.
        # The first reward is the original one and the other rewards are proxy rewards given for reaching the respective
        # exits. If track_exits is True, the distribution of exits is saved as a parameter and can be used to initialize
        # the complementary local MDP on the rest of the state space. If randomstart is True, the start state is sampled
        # from the entries. This requires that the original MDP can be reset to this state.
        self.env = env

        self.entries = entries
        self.exits = exits
        self.dynamic = False
        self.track_exits = track_exits
        self.randomstart = randomstart
        self.gamma = gamma
        if track_exits is True:
            self.exit_distribution = {}

    def reset(self):
        self.env.reset()
        if self.randomstart:
            self.env.state = random.choice(self.entries)
        if self.track_exits is True:
            self.last_entry = self.env.state
        return self.env.state
    def step(self, act, Qs=None):
        obs, rew, done, _ = self.env.step(act)
        rews = np.zeros(len(self.exits) + 1)
        rews[0] = rew
        if obs in self.exits:
            i = self.exits.index(obs)
            done = True
            rews[i + 1] = self.gamma
            if self.track_exits is True:
                try:
                    self.exit_distribution[self.last_entry][i] = self.exit_distribution[self.last_entry][i] + 1
                except:
                    self.exit_distribution[self.last_entry] = np.zeros(len(self.exits))
                    self.exit_distribution[self.last_entry][i] = self.exit_distribution[self.last_entry][i] + 1

        return obs, rews, done, None


class partial_env_B:
    def __init__(self, env, entries, exits, value_map, actions, gamma=0.99, entry_distribution=None):
        # Implements a local MDP on the part of the state space confined by entries and exits.
        # Value map gets the Qs for the entry states and an index i and returns the ith entry of
        # the operator F applied to the entry Qs and is used to give bonus rewards for transitions to the exits.
        # Theoretically, this might be doable by a neural net that maps states to affine functions and applies
        # them to the Qs, instead. entry_distribution can be used to define a different start distribution than in the
        # original MDP. This is necessary, if the original MDP starts outside of the treated part.

        self.env = env

        self.entries = entries
        self.exits = exits

        self.gamma = gamma
        self.dynamic = True
        self.value_map = value_map
        self.entry_distribution = entry_distribution
        self.actions = actions

    def reset(self):
        self.env.reset()
        if self.entry_distribution is not None:
            self.env.state = np.random.choice(self.entries, p=self.entry_distribution)
        return self.env.state

    def step(self, act, Qs):
        obs, rew, done, _ = self.env.step(act)
        if obs in self.exits:
            i = self.exits.index(obs)
            done = True
            for s in self.entries:
                try:
                    Qs[s]
                except:
                    Qs[s] = [0 for i in range(self.actions)]
            value = self.value_map([max(Qs[s]) for s in self.entries], i)
            rew = rew + self.gamma * value
        return obs, rew, done, None
```

```
import numpy as np
from matplotlib import pyplot as plt

from agents import policy_agent
from algorithms import Tabular_q, SARSA, get_F, learn_matrices, Tdzero
from environments import random_MDP
from functions import Multi_Qs_to_F, Multi_Vs_to_F
from misc import compareplot
from wrapper import partial_env_A_multireward

np.random.seed(0)

Ws=[]
Bs=[]
for i in range(10000):
    env, W_True, B_True = random_MDP(50, 4, 46)
    Ws.append(np.linalg.norm(W_True,ord=np.inf))
    Bs.append(np.linalg.norm(B_True,ord=np.inf))
B50=np.mean(Bs)
W50=np.mean(Ws)

Ws=[]
Bs=[]
for i in range(10000):
    env, W_True, B_True = random_MDP(10, 4, 6)
    Ws.append(np.linalg.norm(W_True,ord=np.inf))
    Bs.append(np.linalg.norm(B_True,ord=np.inf))
B10=np.mean(Bs)
W10=np.mean(Ws)




random_dict = {key: [0.25, 0.25, 0.25, 0.25] for key in range(100)}
random = policy_agent(random_dict, 4)

k = 25
cs = [0.5, 1, 10, 100, 1000, 10000]

QW = []
VW = []
QB = []
VB = []

for c in cs:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        F, W, B = get_F(env, 1000, 4, random_dict
                        , np.arange(6), [6, 7, 8, 9], c=c, episode_length=100, epsilon=0,
                        alpha=lambda x, y: 0.1, evaluation="SARSA", re_evaluation_factor=1)
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        F, W, B = get_F(env, 1000, 4, random_dict
                        , np.arange(6), [6, 7, 8, 9], c=c, episode_length=100, epsilon=0,
                        alpha=lambda x, y: 0.1, evaluation="td", re_evaluation_factor=1)
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

sWs = []
sBs = []
for i in range(k):
    env, W_True, B_True = random_MDP(10, 4, 6)
    b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
    Qs = SARSA(b, 1000, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: 0.1)
    F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(6))
    sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
    sBs.append(np.linalg.norm(B - B_True, ord=np.inf))

QdW = sWs
QdB = sBs

sWs = []
sBs = []
for i in range(k):
    env, W_True, B_True = random_MDP(10, 4, 6)
    b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
    Vs = Tdzero(b, 1000, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: 0.1)
    F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(6))
    sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
    sBs.append(np.linalg.norm(B - B_True, ord=np.inf))

VdW = sWs
VdB = sBs

compareplot(np.log10(cs), [VW, QW, [VdW for i in cs], [QdW for i in cs],[[W10] for i in cs]],
            ["V_indirect", "Q_indirect", "V_direct", "Q_direct","True_W"],
            ["b", "orange", "lightblue", "navajowhite", "red"],
            title="Random_MDP(10,4,6)_1000_episodes", xlabel="log_10_c", ylabel="error_in_W", ylim=None)

compareplot(np.log10(cs), [VB, QB, [VdB for i in cs], [QdB for i in cs],[[B10] for i in cs]],
            ["V_indirect", "Q_indirect", "V_direct", "Q_direct","True_B"],
            ["b", "orange", "lightblue", "navajowhite", "red"],
            title="Random_MDP(10,4,6)_1000_episodes", xlabel="log_10_c", ylabel="error_in_B", ylim=None)


alpha = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1]
probW = []
QW = []
VW = []
probB = []
QB = []
VB = []
for a in alpha:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 8, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Vs = Tdzero(b, 1000, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: a)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 8, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Qs = SARSA(b, 1000, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: a)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

compareplot(np.log10(alpha), [VW, QW ,[[W10] for i in alpha]],
            ["V", "Q","True_W"], ["b", "orange", "red"],
            title="Random_MDP(10,4,6)_1000_episodes", xlabel="log_10_alpha", ylabel="error_in_W", ylim=None)

compareplot(np.log10(alpha), [VB, QB,[[B10] for i in alpha]],
            ["V", "Q","True_B"], ["b", "orange", "red"],
            title="Random_MDP(10,4,6)_1000_episodes", xlabel="log_10_alpha", ylabel="error_in_B", ylim=None)

episodes = (np.arange(5)) * 500 + 100
probW = []
QW = []
VW = []
probB = []
```

```
QB = []
VB = []
for e in episodes:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        F, W, B = learn_matrices(env, e, random, np.arange(6), [6, 7, 8, 9])
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    probW.append(sWs)
    probB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Qs = SARSA(b, e, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: 0.1)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Vs = Tdzero(b, e, 4, random, episode_length=100, epsilon=0, alpha=lambda x, y: 0.1)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

compareplot(episodes, [VW, QW, probW, [[W10] for i in episodes]],
            ["V", "Q", "Prob","True_W"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha=0.1", xlabel="episodes", ylabel="error_in_W", ylim=None)

compareplot(episodes, [VB, QB, probB,[[B10] for i in episodes]],
            ["V", "Q", "Prob","True_B"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha=0.1", xlabel="episodes", ylabel="error_in_B", ylim=None)


def alpha(v, t):
    return 1 / (v + 1)


episodes = (np.arange(5)) * 500 + 100
probW = []
QW = []
VW = []
probB = []
QB = []
VB = []
for e in episodes:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        F, W, B = learn_matrices(env, e, random, np.arange(6), [6, 7, 8, 9])
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    probW.append(sWs)
    probB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Qs = SARSA(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
```

```
            sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Vs = Tdzero(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

compareplot(episodes, [VW, QW, probW, [[W10] for i in episodes]],
            ["V", "Q", "Prob","True_W"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha_decay", xlabel="episodes", ylabel="error_in_W", ylim=None)

compareplot(episodes, [VB, QB, probB, [[B10] for i in episodes]],
            ["V", "Q", "Prob","True_B"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha_decay", xlabel="episodes", ylabel="error_in_B", ylim=None)


def alpha(v, t):
    return 5 / (v + 5)


episodes = (np.arange(5)) * 500 + 100
probW = []
QW = []
VW = []
probB = []
QB = []
VB = []
for e in episodes:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        F, W, B = learn_matrices(env, e, random, np.arange(6), [6, 7, 8, 9])
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    probW.append(sWs)
    probB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Qs = SARSA(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(10, 4, 6)
        b = partial_env_A_multireward(env, np.arange(6), [6, 7, 8, 9])
        Vs = Tdzero(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(6))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

compareplot(episodes, [VW, QW, probW, [[W10] for i in episodes]],
            ["V", "Q", "Prob","True_W"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha_decay_5", xlabel="episodes", ylabel="error_in_W", ylim=None)

compareplot(episodes, [VB, QB, probB, [[B10] for i in episodes]],
            ["V", "Q", "Prob","True_B"], ["b", "orange", "k","red"],
            title="Random_MDP(10,4,6)_alpha_decay_5", xlabel="episodes", ylabel="error_in_B", ylim=None)


def alpha(v, t):
    return 1 / (v + 1)


episodes = (np.arange(5) + 1) * 2000
probW = []
QW = []
VW = []
probB = []
QB = []
VB = []
for e in episodes:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        F, W, B = learn_matrices(env, e, random, np.arange(46), [46, 47, 48, 49])
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    probW.append(sWs)
    probB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        Qs = SARSA(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(46))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        Vs = Tdzero(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(46))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

compareplot(episodes, [VW, QW, probW, [[W50] for i in episodes]],
            ["V", "Q", "Prob","True_W"], ["b", "orange", "k","red"],
            title="Random_MDP(50,4,46)_alpha_decay", xlabel="episodes", ylabel="error_in_W", ylim=None)

compareplot(episodes, [VB, QB, probB, [[B50] for i in episodes]],
            ["V", "Q", "Prob","True_B"], ["b", "orange", "k","red"],
            title="Random_MDP(50,4,46)_alpha_decay", xlabel="episodes", ylabel="error_in_B", ylim=None)


def alpha(v, t):
    return 10 / (v + 10)


episodes = (np.arange(5) + 1) * 2000
probW = []
QW = []
VW = []
probB = []
QB = []
VB = []
for e in episodes:
    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        F, W, B = learn_matrices(env, e, random, np.arange(46), [46, 47, 48, 49])
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    probW.append(sWs)
    probB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        Qs = SARSA(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Qs_to_F(random_dict, Qs, np.arange(46))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    QW.append(sWs)
    QB.append(sBs)

    sWs = []
    sBs = []
    for i in range(k):
        env, W_True, B_True = random_MDP(50, 4, 46)
        b = partial_env_A_multireward(env, np.arange(46), [46, 47, 48, 49])
        Vs = Tdzero(b, e, 4, random, episode_length=100, epsilon=0, alpha=alpha)
        F, W, B = Multi_Vs_to_F(random_dict, Vs, np.arange(46))
        sWs.append(np.linalg.norm(W - W_True, ord=np.inf))
        sBs.append(np.linalg.norm(B - B_True, ord=np.inf))
    VW.append(sWs)
    VB.append(sBs)

compareplot(episodes, [VW, QW, probW ],
            ["V", "Q", "Prob"], ["b", "orange", "k"],
            title="Random_MDP(50,4,46)_alpha_decay_10", xlabel="episodes", ylabel="error_in_W", ylim=None)

compareplot(episodes, [VB, QB, probB],
            ["V", "Q", "Prob"], ["b", "orange", "k"],
            title="Random_MDP(50,4,46)_alpha_decay_10", xlabel="episodes", ylabel="error_in_B", ylim=None)
```

Experiment2.py

```
import numpy as np
from matplotlib import pyplot as plt

from agents import policy_agent_deterministic
from algorithms import Tabular_q, SARSA, learn_matrices, Tdzero
from environments import robot
from functions import Multi_Qs_to_F, Qs_to_policy
from misc import compareplot, compareplot_Q
from wrapper import partial_env_A_multireward, partial_env_B

N = 100
epsnum = [10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5), int(10 ** 4), int(10 ** 4.5),
          int(10 ** 5)]

def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []
epslen = []

for e in epsnum:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    epslen_temp = []
    for k in range(N):
        env = robot()
        Qs = Tabular_q(env, e, 4, episode_length=20, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf, soft_end=True)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
        epslen_temp.append(env.steps)
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)
    epslen.append(epslen_temp)

compareplot_Q(np.log10(epsnum), [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="log_10_episodes", ylabel="Q-value",
            ylim=(0, 150))
compareplot(np.log10(epsnum), [epslen],
            ["Q-learning"], ["b"],
            title="Number_of_interactions_with_the_environment", xlabel="log_10_episodes",
            ylabel="Interactions", ylim=(0, 201000 * 10))
print(Q2)

env = robot()


def alpha(v, t):
    return 10 / (v + 10)


a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)

policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs = SARSA(a, 1000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
base = a.env.steps
F, W, B = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])

b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                    entry_distribution=None)

epsnum = [10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5), int(10 ** 4), int(10 ** 4.5),
          int(10 ** 5)]


def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []
epslen = []

for e in epsnum:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    epslen_temp = []
    for k in range(N):
        env = robot()
        b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                          entry_distribution=None)
        Qs = Tabular_q(b, e, 4, episode_length=20, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf)
        epslen_temp.append(b.env.steps)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)
    epslen.append(epslen_temp)

compareplot_Q(np.log10(epsnum), [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Expert_Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="log_10_episodes", ylabel="Q-value",
            ylim=(0, 150))

compareplot(np.log10(epsnum), [epslen, np.array(epslen) + base],
            ["Without_learning_of_the_Expert_model", "Including_learning_of_the_Expert_model"], ["b", "black"],
            title="Number_of_interactions_with_the_environment", xlabel="log_10_episodes",
            ylabel="Interactions", ylim=(0, 201000 * 10))
print(Q2)

env = robot()


def alpha(v, t):
    return 10 / (v + 10)


a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)

policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs = SARSA(a, 1000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
F1, W1, B1 = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])
base = a.env.steps
policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 1,
               (-13, 0, 0): 1,
               (-14, 0, 0): 1, (-15, 0, 0): 1, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs = SARSA(a, 1000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
F2, W2, B2 = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])
base += a.env.steps
```

```
policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 1, (-10, 0, 0): 1, (-11, 0, 0): 1, (-12, 0, 0): 1,
               (-13, 0, 0): 1,
               (-14, 0, 0): 1, (-15, 0, 0): 1, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)
Qs = SARSA(a, 1000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
F3, W3, B3 = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])
base += a.env.steps


def F(v, i):
    return max([F1(v, i), F2(v, i), F3(v, i)])


b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                  entry_distribution=None)

epsnum = [10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5), int(10 ** 4), int(10 ** 4.5),
          int(10 ** 5)]


def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []
epslen = []

for e in epsnum:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    epslen_temp = []
    for k in range(N):
        env = robot()
        b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                          entry_distribution=None)
        Qs = Tabular_q(b, e, 4, episode_length=20, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
        epslen_temp.append(b.env.steps)
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)
    epslen.append(epslen_temp)

compareplot_Q(np.log10(epsnum), [Q0, Q1, Q2, Q3],
              ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
              title="Multiexpert_Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="log_10_episodes", ylabel="Q-value",
              ylim=(0, 150))
compareplot(np.log10(epsnum), [epslen, np.array(epslen) + base],
            ["Without_learning_of_the_Expert_models", "Including_learning_of_the_Expert_models"], ["b", "black"],
            title="Number_of_interactions_with_the_environment", xlabel="log_10_episodes",
            ylabel="Interactions", ylim=(0, 201000 * 10))

print(Q2)

env = robot()


def alpha(v, t):
    return 10 / (v + 10)


a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)

policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
                (-14, 0, 0): 1, (-15, 0, 0): 1, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs = SARSA(a, 1000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
F, W, B = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])

b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                  entry_distribution=None)

epsnum = [10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5), int(10 ** 4), int(10 ** 4.5),
          int(10 ** 5)]


def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []

for e in epsnum:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    for k in range(N):
        Qs = Tabular_q(b, e, 4, episode_length=20, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)

compareplot_Q(np.log10(epsnum), [Q0, Q1, Q2, Q3],
              ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
              title="Expert_Q-learning_with_suboptimal_expert", xlabel="log_10_episodes", ylabel="Q-value", ylim=(0, 150))
print(Q2)

env = robot()


def alpha(v, t):
    return 10 / (v + 10)


a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)

policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs_Sarsa = SARSA(a, 100000, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)

noise = [1e-4, 1e-3, 1e-2, 1e-1, 10 ** (-0.9), 10 ** (-0.8), 10 ** (-0.7), 10 ** (-0.6), 10 ** (-0.5)]

Q0 = []
Q1 = []
Q2 = []
Q3 = []

for std in noise:
    Qs_noisy = {
        key: [(Q + np.random.normal(scale=std, size=len(Q)) if type(Q) == np.ndarray else 0) for Q in Qs_Sarsa[key]] for
    key
        in Qs_Sarsa}
    F, W, B = Multi_Qs_to_F(policy_dict, Qs_noisy, [(-i, 0, 0) for i in range(17)])
    b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                      entry_distribution=None)
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    for k in range(N):
        Qs = Tabular_q(b, int(10 ** 4.5), 4, episode_length=10, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)

compareplot_Q(np.log10(noise), [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Expert_Q-learning_for_the_Robot_MDP_with_perturbed_F", xlabel="log_10_std", ylabel="Q-value",
            ylim=(0, 150))
print(Q2)


def alpha(v, t):
    return 10 / (v + 10)


env = robot()


def exit(vector):
    return vector[0] + vector[-1]


def Q_trafo(Qs):
    Q = [exit(Qs[i]) if isinstance(Qs[i], np.ndarray) else 0 for i in range(len(Qs))]
    return Q


policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)
Qs = SARSA(a, 1000, 4, policy, episode_length=100, epsilon=0, alpha=alpha, gamma=0.9)
F, W, B = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])

epsnum = [10 ** 1, int(10 ** 1.5), 10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5)]
epslens = []
Diffs = []
for e in epsnum:
    Diff = []
    epslen = []
    for i in range(int(N / 4)):
        env = robot()
        a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)],
                                      gamma=0.9)
        Qs = Tabular_q(a, e, 4, episode_length=20, epsilon=0.25, alpha=alpha, gamma=0.9, Q_trafo=Q_trafo)
        base = a.env.steps
        epslen.append(base)
        policy = Qs_to_policy(Qs, Q_trafo)
        policy_dict = {}
        for j in [(-i, 0, 0) for i in range(17)]:
            try:
                policy_dict[j] = policy[j]
            except:
                policy_dict[j] = 0
        F1, W1, B1 = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])
        Diff.append(np.linalg.norm(W - W1))

    epslens.append(epslen)
    Diffs.append(Diff)

compareplot(np.log10(epsnum), [Diffs],
            ["Difference_in_W"], ["b"],
            title="Q-learning_for_approximating_F", xlabel="log_10_episodes", ylabel=r'Error_($||\cdot||_{\infty}$)',
            ylim=(0, 4))
compareplot(np.log10(epsnum), [Diffs],
            ["Q_learning"], ["b"],
            title="Error_in_expert_model_with_learnt_policy_(fixed_goal)", xlabel="log_10_episodes",
            ylabel=r'Error_($||\cdot||_{\infty}$)', ylim=(0, 4))
compareplot(np.log10(epsnum), [epslens, [[2000000]for i in epslens]],
            ["Learning_expert_policy_for_fixed_goal","Q-learning(total)"], ["b","orange"],
            title="Number_of_interactions_with_the_environment", xlabel="log_10_episodes",
            ylabel="Interactions", ylim=(-10000, 201000 * 10))


def alpha(v, t):
    return 10 / (v + 10)


env = robot()


def exit(vector):
    return vector[0] + vector[-1]


def Q_trafo(Qs):
    Q = [exit(Qs[i]) if isinstance(Qs[i], np.ndarray) else 0 for i in range(len(Qs))]
    return Q


policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)
Qs = SARSA(a, 1000, 4, policy, episode_length=100, epsilon=0, alpha=alpha, gamma=0.9)
F, W, B = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])

epsnum = [10 ** 1, int(10 ** 1.5), 10 ** 2, int(10 ** 2.5), 10 ** 3, int(10 ** 3.5)]
Diffs = []
for e in epsnum:
    Diff = []
    for i in range(int(N / 4)):
        a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)],
                                      gamma=0.9)
        Qs = SARSA(a, e, 4, policy, episode_length=5, epsilon=0, alpha=alpha, gamma=0.9)
        F1, W1, B1 = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])
        Diff.append(np.linalg.norm(W - W1))

    Diffs.append(Diff)

compareplot(np.log10(epsnum), [Diffs],
            ["Difference_in_W"], ["b"],
            title="SARSA_for_approximating_F", xlabel="log_10_episodes", ylabel=r'Error_($||\cdot||_{\infty}$)',
            ylim=(0, 4))
compareplot(np.log10(epsnum), [Diffs],
            ["SARSA"], ["b"],
            title="Error_in_expert_model_using_a_fixed_policy", xlabel="log_10_episodes",
            ylabel=r'Error_($||\cdot||_{\infty}$)', ylim=(0, 4))

eps = [0.05, 0.1, 0.15, 0.2, 0.25, 0.5, 0.75, 0.8, 0.85, 0.9 , 0.95 , 1.0]


def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []

for e in eps:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    for k in range(N):
        env = robot()
        Qs = Tabular_q(env, 10 ** 3, 4, episode_length=20, epsilon=e,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf, soft_end=True)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)

compareplot_Q(eps, [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="epsilon", ylabel="Q-value", ylim=(0, 150))

print(Q2)

env = robot()


def alpha(v, t):
    return 10 / (v + 10)


a = partial_env_A_multireward(env, [(-i, 0, 0) for i in range(17)], [(i + 1, 0, 0) for i in range(16)], gamma=0.9)

policy_dict = {(0, 0, 0): 3, (-1, 0, 0): 3, (-2, 0, 0): 3, (-3, 0, 0): 3, (-4, 0, 0): 3, (-5, 0, 0): 3, (-6, 0, 0): 3,
               (-7, 0, 0): 3, (-8, 0, 0): 3, (-9, 0, 0): 3, (-10, 0, 0): 3, (-11, 0, 0): 3, (-12, 0, 0): 3,
               (-13, 0, 0): 3,
               (-14, 0, 0): 3, (-15, 0, 0): 3, (-16, 0, 0): 1}
policy = policy_agent_deterministic(policy_dict, 4)

Qs = SARSA(a, 1000, 4, policy, episode_length=20, epsilon=0, alpha=alpha, gamma=0.9)
F, W, B = Multi_Qs_to_F(policy_dict, Qs, [(-i, 0, 0) for i in range(17)])

b = partial_env_B(env, [(i + 1, 0, 0) for i in range(16)], [(-i, 0, 0) for i in range(17)], F, 4, gamma=0.9,
                  entry_distribution=None)

eps = [0.05, 0.1, 0.15, 0.2, 0.25, 0.5, 0.75, 0.8, 0.85, 0.9 , 0.95 , 1.0]


def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []

for e in eps:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    for k in range(N):
        Qs = Tabular_q(b, 10 ** 3, 4, episode_length=20, epsilon=e,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)

compareplot_Q(eps, [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Expert_Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="epsilon", ylabel="Q-value",
            ylim=(0, 150))
print(Q2)

epslength = [2, 2 * int(10 ** (0.5)), 2 * int(10 ** 1), 2 * int(10 ** (1.5)), 2 * int(10 ** 2), 2 * int(10 ** 2.5),
             2 * int(10 ** 3)]
```

```
def alpha(v, t):
    return 10 / (v + 10)


Q0 = []
Q1 = []
Q2 = []
Q3 = []
epslen = []

for e in epslength:
    Q0_temp = []
    Q1_temp = []
    Q2_temp = []
    Q3_temp = []
    epslen_temp = []
    for k in range(N):
        env = robot()
        Qs = Tabular_q(env, 10 ** 3, 4, episode_length=e, epsilon=0.25,
                       alpha=alpha, gamma=0.9, eval_interval=np.inf, soft_end=True)
        start_values = Qs[(16, 0, 0)]
        Q0_temp.append(start_values[0])
        Q1_temp.append(start_values[1])
        Q2_temp.append(start_values[2])
        Q3_temp.append(start_values[3])
        epslen_temp.append(env.steps)
    Q0.append(Q0_temp)
    Q1.append(Q1_temp)
    Q2.append(Q2_temp)
    Q3.append(Q3_temp)
    epslen.append(epslen_temp)

compareplot_Q(np.log10(np.array(epslength)), [Q0, Q1, Q2, Q3],
            ["Action_0", "Action_1", "Action_2", "Action_3"], ["b", "orange", "red", "black"],
            title="Q-learning_for_the_Robot_MDP_with_gamma=0.9", xlabel="log10_episode_length", ylabel="Q-value",
            ylim=(0, 150))
compareplot(np.log10(np.array(epslength)), [epslen],
            ["Q-learning"], ["b"],
            title="Number_of_interactions_with_the_environment", xlabel="log_10_episode_length",
            ylabel="Interactions", ylim=(0, 201000 * 10))

print(Q2)

#Optimal policy
env = robot()
total = 0
steps = 0
env.reset()
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
for j in range(3000):
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
print(total)
#Action 3 followed by by optimal policy
total = 0
steps = 0
env.reset()
obs, rew, done, _ = env.step(3)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
for j in range(3000):
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
print(total)

#Action 0 followed by by optimal policy
total = 0
steps = 0
env.reset()
obs, rew, done, _ = env.step(0)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
```

```
total += rew * 0.9 ** steps
steps += 1
for j in range(3000):
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
print(total)

#Action 1 followed by by optimal policy
total = 0
steps = 0
env.reset()
obs, rew, done, _ = env.step(1)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(3)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(3)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
obs, rew, done, _ = env.step(2)
print(obs)
total += rew * 0.9 ** steps
steps += 1
for j in range(3000):
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(3)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
    obs, rew, done, _ = env.step(2)
    total += rew * 0.9 ** steps
    steps += 1
print(total)

#Qs:
#0: 128.9996538917452
#1: 93.31174768708226
#2: 142.2218376574947
#3: 115.1996885025707
```