List of Corrections

University of Mannheim

Bachelor Thesis

Reinforcement Learning

by

Felix Benning

born on the 27.11.1996 in Nürtingen
matriculation number 1501817

in the
Fakulty for Mathematics in Business and Economics
Supervisor: Prof. Dr. Leif Döring

Due Date: 11.05.2019

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

This thesis was not previously presented to another examination board and has not been published.

City, Date Signature

Contents

Introduction

Let us consider a system to be a intelligent, if it collects information from its sensors and transforms this information into an "appropriate" response. Then the question where this input comes from (eyes, ears, cameras, etc.), or what kind of input it is (visual, audio, etc.), is as irrelevant for this definition, as the question what a response is (motor activity, sound, etc.). An intelligent system is simply a function mapping inputs to "appropriate" outputs. The question what "appropriate" means, is in contrast a lot more important and difficult to answer. But if we assume that there exists a "correct" response, then this correct response is itself a function from the input space to the output space. Artificial intelligence is therefore simply an artificial implementation of this correct response function.

If we have complete knowledge of this function, we can encode this function as an executable program manually. But more often than not, we do not really understand the rules according to which the inputs are supposed to be transformed into outputs.

For this reason we might want a machine to "learn" these rules by itself., i.e. we want it to approximate the correct response function. There are two larger subcategories in this category of *machine learning*.

In *supervised learning* we do not know the correct response function, but we know the correct response for certain inputs intuitively. An example for this case is image classification of animals. There we do not know how the input (pixels of the picture) map to the output (name of the animal), but we can tell for a given picture what animal is depicted intuitively; i.e. we can provide examples of the correct response function to the learning algorithm. Supervised Learning is therefore concerned with generalization from examples. And while approximating a function with a finite sample of (error prone) function evaluations is a relatively old problem in numerical analysis and statistics, supervised learning is often associated with more recent approaches like artificial neuronal networks.

While we are still able to provide examples in supervised learning, *unsupervised learning* has to work with even less. This category includes clustering and principle component analysis, as well as *reinforcement learning*. Reinforcement learning is applied to problems, where we "know the correct response if we see it", but can not provide examples ourselves. This is for example the case with

walking. We know how walking looks like, but we have a hard time giving an example for the correct output signals sent to the motors in the leg. It is also used in cases where we do not know the correct response and have never seen it, but are able to compare different response functions. Examples for this range from games like chess to profit maximization of a company.

In general reinforcement learning is used in cases, where humans or the environment in general can rate the solution, and provide rewards for good outcomes. Similar to the conditioning of animals, desirable actions are *reinforced* with rewards.

But in order to write algorithms which maximize their rewards, we first need to formulate the relationship of possibly delayed rewards with actions (outputs) of the learning algorithm in certain states (given certain inputs). The model for this relationship – which virtually all modern reinforcement learning algorithms are based on – is the Markov Decision Process.

We will therefore introduce this model and its properties in the first chapter, continue with a review of common reinforcement algorithms in the second chapter, and explore the relation between the theory of stochastic approximation and reinforcement learning in the third chapter, which yields proofs of convergence to the optimal policy for a number of reinforcement algorithms.

Chapter 1

Reinforcement Learning Algorithms

1.1 Introduction

Dynamic Programming usually breaks down in the real world for two reasons:

1. The transition probabilities and immediate rewards are not known or hard to calculate.
2. The state and action space is too large to even compute one iteration of Dynamic Programming for every state-action tuple (e.g. possible positions and possible moves in every position in chess).

This is where *Reinforcement Learning* algorithms come in, which try to find solutions without having to sweep the entire state space. In this chapter based on Sutton and Barto (1998) we will examine advantages and disadvantages of various algorithms and discuss possible variations and extensions. In the next chapter we will show almost sure convergence of the basic algorithms introduced in this chapter and illustrate their connection to stochastic approximation.

FiXme: check outline of the plan

We separate their introduction and the convergence proofs, because – while the guarantee of almost sure convergence is reassuring – it is of little use for comparing various algorithms. Since one of the reasons for moving away from Dynamic Programming in the first place was, that we could not calculate the value function for the entire state space within a reasonable time frame. As the size of the state space is often too large for that. Therefore almost sure convergence should not be viewed as more than an entry requirement. For this reason most papers compare algorithms empirically on various example problems. And for some of the more complex algorithms convergence proofs simply do not exist yet.

So since the theoretical convergence properties are usually only ever an afterthought, it is more natural to introduce the various algorithms heuristically, explaining what specific problems they try to address with examples.

But let us ignore the second problem for a moment and consider the case where we only have the first problem (p and r unknown). Then we can learn from a sample $((X_t, A_t, Y_t, R_t), t \in \{0, \dots, T\})$ with

$$(Y_t, R_t) \sim \mathcal{P}(\cdot \mid X_t, A_t)$$

and with $Y_t = X_{t+1}$ if the sample is generated sequentially by a behaviour. But there is no reason not to allow this more general sample which might be useful in cases where you can jump around in the state space and try different transitions at will.

We can then use this batch of transitions to calculate estimators $\hat{p}, \hat{r}$ for the state transitions and immediate rewards $\hat{r}$. And use Dynamic Programming on these estimators.

Algorithm 1 Naive Batch Learning Algorithm

1. Generate the history $(X_t, A_t, Y_t, R_t, t \in \{0, \dots, T\})$

```
for (y, x, a) ∈ X × X × A do                    ▷ initialize variables
    rewards[x, a]←list()
    stateTransitions[y | x, a]← 0
    totalTransitions[x, a]← 0
end for
for t = 0, ..., T do
    rewards[X_t, A_t].append(R_{t+1})
    stateTransitions[Y_t | X_t, A_t]++
    totalTransitions[X_t, A_t]++
end for
for (x, a) ∈ X × A do
    r̂(x, a) ← average(rewards[x,a])
    for y ∈ X do
        p̂(y | x, a) ← stateTransitions[y | x, a]/totalTransitions[x, a]
    end for
end for
```

2. Use Dynamic Programming on $\hat{r}$, $\hat{p}$

FiXme: add code/ ref code Dynamic Programming

FiXme: numerical stability analysis?

If the batch was generated by an exploration policy we separated the *exploration* from the later *exploitation*. The methods which do that are often grouped under the term *Batch Reinforcement Learning* or *off-line* learning.

This method works fine, if the state space is small and one can sample enough observations for every state and action in a reasonable timeframe. But if our state space is too large for that, it is impractical to wait for this.

1.2 Monte Carlo

One idea to tackle larger state spaces is, that it is often unnecessary to know the value function in every state.

To visualize this idea it is useful to imagine the state space to be a room the agent has to navigate.

	γ^2	γ	G	
	γ^3			
	γ^4			
	γ^5			
	S			

The state space $\mathcal{X}$ are the tiles on the floor, the actions $\mathcal{A}$ are the adjacent tiles, where the next state is with probability one equal to the action, and the reward is 0 for every transition but the transition to the goal (G) where the reward is 1 and the game ends. The start state is S. The value of choosing a tile is indicated in grey on the tile.

It is often enough for the agent to know the action value function for the states on his path, without knowing the value of states in the corners. Since he can then follow these "breadcrumbs" to find the goal reliably again. And it will quickly stop walking in circles if it goes into the direction of the highest value next time.

This idea is the basis for Monte Carlo algorithms. To make notation more brief we will introduce the algorithm to calculate the value function, the action value function will be analogous. Consider an episodic MDP and a behaviour π then

$$\sum_{t=0}^{\infty} \gamma^t R_{t+1} = \sum_{t=0}^{T} \gamma^t R_{t+1}$$

is a bias free estimator for $V^\pi(X_0)$ for episode length T. As the definition was

$$V^\pi(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \,\middle|\, X_0 = x \right]$$

And because of the Markov property

$$\sum_{t=k}^{T} \gamma^t R_{t+1}$$

is a bias free estimator for $V^\pi(X_k)$. *First-visit Monte Carlo* uses the return after the first visit of a state $x \in \{X_1, \ldots, X_t\}$ as an estimator for $V^\pi(x)$.

Because of the strong law of large numbers, first-visit Monte Carlo algorithm causes the value function estimation $\hat{V}^\pi(x)$ to converge with probability 1 to

Algorithm 2 First-visit Monte Carlo

for $x \in \mathcal{X}$ **do** ▷ initializing
    Returns$(x) \leftarrow$ list()
    $V^\pi(x) \leftarrow 0$
end for
while true **do** (forever) ▷ learning
    Generate an episode $((X_t, R_{t+1}), t \in \{0, \dots, T\})$ with behaviour π
    for $x \in \{X_0, \dots, X_T\}$ **do**
        $k \leftarrow \min\{t \mid X_t = x\}$
        Returns(x).append$(\sum_{t=k}^{T} \gamma^t R_{t+1})$
        $V^\pi(x) \leftarrow$ average(Returns(x))
    end for
end while

$V^\pi(x)$ for every state $x \in \mathcal{X}$ which is visited with positive probability given behaviour π and starting distribution X_0.

Every-visit Monte Carlo uses the Return after every visit of a state x to estimate $V^\pi(x)$. Since this means that the tail of the rewards can be included in multiple returns, the returns are not independent from each other anymore which make proving convergence a little bit more difficult than simply applying the strong law of large numbers. But every visit Monte Carlo will turn out to be a special case of $TD(\lambda)$.

FiXme: will TD proof work?

The same method can be applied to learn the action value function Q^π. In this case we take the returns following a state *and* action as estimators for Q^π. But if the model is known and our problem is just a large state space calculating V^π is preferable, since $|\mathcal{X}| \leq |\mathcal{X} \times \mathcal{A}|$ means that the first algorithm needs fewer observations to converge and Q^π can be calculated with V^π given r and p.

1.2.1 From π to π^*

Let us assume exploring starts

$$\mathbb{P}(X_0 = x) > 0 \qquad \forall x \in \mathcal{X}$$

for a moment. Then Monte Carlo converges whatever policy we select. Similar to policy iteration (**??**) we can then alternate between calculating Q^{π_n} and selecting π_{n+1} as a greedy policy with regard to Q^{π_n}. This is referred to as generalized policy iteration (GPI). If we would wait for our Monte Carlo approximation of Q^{π_n} to converge, π_n would converge to π^* for the same reason as policy iteration converges. But remember we are doing Monte Carlo in the first place, because the state space is too large to wait for an evaluation of every state. Which means in practice algorithms alternate between choosing a greedy policy with regard to V^{π_n} and generating an episode with policy π_n,

averaging the estimates from this episode with the estimates of $V^{\pi_{n-1}}$. But since we have to assume

$$V^{\pi_{n-1}} \neq V^{\pi_n}$$

in general, using these old estimates is not bias free. It is easy to see that this procedure can only converge to π^* if it converges at all. Since there are only a finite number of deterministic stationary policies it would have to stay constant at some point, but for a constant policy Monte Carlo will converge, and then the policy can only stay constant if it is greedy with regards to its own value function, which forces it to be optimal (**??**). But it is still an open problem whether or not this alternating procedure converges at all (Sutton and Barto 2018, p. 99).

If we remove the assumption of exploring starts, we still need to ensure that every state is visited with positive probability for MC to converge. This requires the policy to do the exploring. There are multiple approaches to exploration which we will discuss later. We will discuss the most straightforward example here, under the assumption that we can calculate Q^{π_n} somehow.

FiXme: check claim

Definition 1.2.1. A stationary policy π is called

soft if it fulfils

$$\pi(a \mid x) > 0 \qquad \forall (x, a) \in \mathcal{X} \times \mathcal{A}$$

ε-soft if it fulfils

$$\pi(a \mid x) > \frac{\varepsilon}{|\mathcal{A}_x|} \quad \forall (x, a) \in \mathcal{X} \times \mathcal{A}$$

ε-greedy with regard to Q, if it selects the greedy action w.r.t. Q with probability $(1 - \varepsilon)$ and a (uniform) random action with probability ε, i.e.

$$\pi(a \mid x) = \begin{cases} (1 - \varepsilon) + \frac{\varepsilon}{|\mathcal{A}_x|} & a \text{ is greedy}^1 \\ \frac{\varepsilon}{|\mathcal{A}_x|} & a \text{ is not greedy} \end{cases}$$

Note that an ε-greedy policy is ε-soft.

An ε-soft policy π^* is called *ε-soft optimal* if

$$V^{\pi^*}(x) = \sup_{\pi \ \varepsilon\text{-soft}} (x) V^{\pi} =: \tilde{V}^*(x)$$

Proposition 1.2.2. *Generalized policy iteration with ε-greedy policies converges to a ε-soft optimal policy*

Proof. To use the same argument as with the standard policy iteration, we would need

$$T^*(V^{\pi_n}) = T^{\pi_{n+1}}(V^{\pi_n})$$

[1] w.l.o.g. only one greedy action

But this is not true. To circumvent this problem we "move the requirement of ε-softness into the environment". I.e. we define an MDP $\tilde{M}$ where

$$\tilde{\mathcal{P}}(\cdot \mid x, a) \coloneqq (1-\varepsilon)\mathcal{P}(\cdot \mid x, a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} \mathcal{P}(\cdot \mid x, b)$$

This means, that with probability ε the transition kernel will ignore the selected action and behave as if an uniformly random action was chosen.

We can transform ε-soft policies π from the old MDP to stationary policies $\tilde{\pi}$ of $\tilde{M}$ with a mapping f, where

$$f(\pi)(a \mid x) = \tilde{\pi}(a \mid x) \coloneqq \frac{\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon} \geq 0$$

And for every stationary policy $\tilde{\pi}$ of $\tilde{M}$ we can define the ε-soft policy π by

$$\pi(a \mid x) \coloneqq (1-\varepsilon)\tilde{\pi}(a \mid x) + \frac{\varepsilon}{|\mathcal{A}_x|}$$

which is the inverse mapping. Therefore f is a bijection between the ε-soft policies in the old MDP and all stationary policies in the new MDP. We now show that the Value functions V^π stay invariant with regard to this mapping. First note that

$$\begin{aligned}
\tilde{p}(y \mid x, a) &= \tilde{\mathcal{P}}(\{y\} \times \mathbb{R} \mid x, a) \\
&= (1-\varepsilon)p(y \mid x, a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} p(y \mid x, b)
\end{aligned}$$

and together with $q(B \mid x, a) \coloneqq \mathcal{P}(\mathcal{X} \times B \mid x, a)$ the complementary marginal distribution we can show

$$\begin{aligned}
\tilde{r}(x, a) &= \int t \; d\tilde{q}(t \mid x, a) = \int t \; d \left((1-\varepsilon) q(\cdot \mid x, a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} q(\cdot \mid x, b) \right)(t) \\
&= (1-\varepsilon) \int t \; dq(t \mid x, a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} \int t \; dq(t \mid x, a) \\
&= (1-\varepsilon) r(x, a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} r(x, b)
\end{aligned}$$

Therefore we know

$$\begin{aligned}
&\sum_{a\in\mathcal{A}_x} \tilde{\pi}(a \mid x)\tilde{r}(x,a) \\
&= \sum_{a\in\mathcal{A}_x} \left(\frac{\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon}\right) \left((1-\varepsilon)r(x,a) + \frac{\varepsilon}{|\mathcal{A}_x|}\sum_{b\in\mathcal{A}_x} r(x,b)\right) \\
&= \sum_{a\in\mathcal{A}_x} \left(\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}\right) r(x,a) + \frac{\varepsilon}{|\mathcal{A}_x|}\sum_{b\in\mathcal{A}_x} r(x,b) \underbrace{\sum_{a\in\mathcal{A}_x} \frac{\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon}}_{=1} \\
&= \sum_{a\in\mathcal{A}_x} \pi(a \mid x) r(x,a)
\end{aligned}$$

and similarly

$$\begin{aligned}
&\mathbb{P}^{\tilde{\pi}}(\tilde{X}_1 = y \mid \tilde{X}_0 = x) = \sum_{a\in\mathcal{A}_x} \tilde{\pi}(a \mid x)\tilde{p}(y \mid x,a) \\
&= \sum_{a\in\mathcal{A}_x} \left(\frac{\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon}\right) \left((1-\varepsilon)p(y \mid x,a) + \frac{\varepsilon}{|\mathcal{A}_x|}\sum_{b\in\mathcal{A}_x} p(y \mid x,b)\right) \\
&= \ldots = \sum_{a\in\mathcal{A}_x} \pi(a \mid x) p(y \mid x,a) = \mathbb{P}^{\pi}(X_1 = y \mid X_0 = x)
\end{aligned}$$

The last two equations together imply

$$\begin{aligned}
\tilde{T}^{\tilde{\pi}} V^{\pi}(x) &= \sum_{a\in\mathcal{A}_x} \tilde{\pi}(a \mid x)\tilde{r}(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathbb{P}^{\tilde{\pi}}(\tilde{X}_1 = y \mid X_0 = x) V^{\pi}(y) \\
&= \sum_{a\in\mathcal{A}_x} \pi(a \mid x) r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathbb{P}^{\pi}(X_1 = y \mid X_0 = x) V^{\pi}(y) \\
&= T^{\pi} V^{\pi}(x) = V^{\pi}(x)
\end{aligned}$$

Since the fixpoint is unique it follows that

$$V^{\tilde{\pi}} = V^{\pi}$$

Since f is bijective this implies

$$\sup_{\tilde{\pi}\in\tilde{\Pi}_S} V^{\tilde{\pi}}(x) = \sup_{\pi \ \varepsilon\text{-soft}} V^{\pi}(x)$$

as well as

$$\begin{aligned} Q^{\tilde{\pi}}(x,a) &= \tilde{r}(x,a) + \gamma \sum_{y \in \mathcal{X}} \tilde{p}(y \mid x,a) V^{\pi}(y) \\ &= (1-\varepsilon) \left(r(x,a) + \gamma \sum_{y \in \mathcal{X}} p(y \mid x,a) V^{\pi}(y) \right) \\ &\qquad + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} \left(r(x,b) + \gamma \sum_{y \in \mathcal{X}} p(y \mid x,b) \right) \\ &= (1-\varepsilon) Q^{\pi}(x,a) + \frac{\varepsilon}{|\mathcal{A}_x|} \sum_{b \in \mathcal{A}_x} \left(r(x,b) + \gamma \sum_{y \in \mathcal{X}} p(y \mid x,b) \right) \end{aligned}$$

Which implies that

$$\arg\max_{a \in \mathcal{A}_x} Q^{\tilde{\pi}}(x,a) = \arg\max_{a \in \mathcal{A}_x} Q^{\pi}(x,a)$$

Therefore greedy with regard to Q^{π} and $Q^{\tilde{\pi}}$ is the same. Let π_n be an ε-soft policy, and let π_{n+1} be ε-greedy with regard to Q^{π_n}. Then $\tilde{\pi}_{n+1} \coloneqq f(\pi_{n+1})$ is greedy w.r.t. $Q^{\tilde{\pi}_n}$

$$\begin{aligned} \tilde{\pi}_{n+1}(a \mid x) = f(\pi_{n+1})(a \mid x) &= \frac{\pi(a \mid x) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon} \\ &= \begin{cases} \frac{\left((1-\varepsilon) + \frac{\varepsilon}{|\mathcal{A}_x|}\right) - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon} = 1 & a \text{ is greedy} \\ \frac{\frac{\varepsilon}{|\mathcal{A}_x|} - \frac{\varepsilon}{|\mathcal{A}_x|}}{1-\varepsilon} = 0 & a \text{ is not greedy} \end{cases} \end{aligned}$$

This finishes our proof

$$\sup_{\pi \ \varepsilon\text{-soft}} V^{\pi}(x) = \sup_{\tilde{\pi} \in \tilde{\Pi}_S} V^{\tilde{\pi}}(x) = \lim_{n \to \infty} V^{\tilde{\pi}_n}(x) = \lim_{n \to \infty} V^{\pi_n}(x) \qquad \square$$

1.2.2 Shortcomings of Monte Carlo

Sutton and Barto provide a very instructive example for the weakness of Monte Carlo. Recall that Monte Carlo tries to estimate V^{π}. So the example assumes a given behaviour and tries to evaluate the value of a situation.

Example 1.2.3 (Driving Home)**.** Let us assume that John Doe works in city A and drives to his home in city B after work every day. Since he wants to get home as quickly as possible, the value of the state is inversely proportional to the time it will take him from a certain position home. This means that estimating the value of a certain state is equivalent to estimating the time left to drive. The longer John drives this route the better he will get at estimating

the time it will take him to drive certain sections of the road. If there is a delay in an earlier section he has a good estimate for the remaining time once he cleared the obstruction. Now imagine he is driving home from a doctors appointment from city A. As soon as he enters the highway to city B, he is able to use his experience driving this route to estimate the remaining time quite precisely.

The Monte Carlo algorithm on the other hand *never* uses existing value estimations to estimate the value of a different state. If John Doe uses Monte Carlo estimates to guess the remaining time from the doctor, the accuracy of his estimates will only ever increase with the times he actually starts driving from the doctor's office.

Since Monte Carlo has more possible "starting points" or generally earlier points in the chain in case of estimating Q^π, it is worse in this case. An extreme example would be two actions in the same state which have the same effect. Even though Monte Carlo might have a good estimate of the value of the first action it will start completely anew for the second action.

FiXme: illustrations?

1.3 Temporal Difference Learning TD

Temporal Difference learning (TD) first proposed by Sutton (1988) addresses this weakness of Monte Carlo algorithms. For a sequence $(X_t, A_t, R_{t+1}, t \in \mathbb{N}_0)$ realized with a stationary behaviour π, we know that

FiXme: realized?

$$\begin{aligned}
&\mathbb{E}^\pi[R_{t+1} + \gamma V(X_{t+1}) \mid X_t = x] \\
&= \mathbb{E}^\pi[r(x, A_0) \mid X_0 = x] + \gamma \sum_{y \in \mathcal{X}} \mathbb{P}^\pi(X_1 = y \mid X_0 = x) V(y) \\
&= T^\pi V(x) \qquad (1.1)
\end{aligned}$$

The random variable $R_{t+1} + \gamma V(X_{t+1})$ is therefore a bias free estimator for $T^\pi V(X_t)$. But because of its variance we can not simply update $V(X_t)$. Instead TD moves the estimate into the direction of this observation. To get an intuition for why this makes sense consider this example.

Example 1.3.1 (Unwinding the Arithmetic Mean)**.**

$$\begin{aligned}
\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i &= \frac{n-1}{n}\frac{1}{n-1}\sum_{i=1}^{n-1} X_i + \frac{1}{n}X_n \\
&= \left(1 - \frac{1}{n}\right)\overline{X}_{n-1} + \frac{1}{n}X_n \\
&= \overline{X}_{n-1} + \underbrace{\frac{1}{n}}_{\text{“learning rate”}}\underbrace{(X_n - \overline{X}_{n-1})}_{\text{“direction”}}
\end{aligned}$$

But Temporal Difference Learning tries to do Dynamic Programming (i.e. replacing V with $T^\pi V$, which would require a learning rate of 1) at the same time as trying to reduce the error of the estimate of $T^\pi V$, where the learning rate should be something like $1/n$. The smaller the learning rate, the more of the old estimate we retain. This has the disadvantage that old estimates are (in a way) previous steps in dynamic programming, which are further away from V^π. Larger learning rates on the other hand retain more of the variance of the latest random variable in the estimator. So one might for example want to have larger steps at the beginning until the dynamic programming part converges and then start to focus more on the variance reduction part. This implies that a wider range of learning rates might be sensible. We therefore define

$$V_{n+1}(x) \coloneqq \begin{cases} V_n(x) + \alpha_n(x)[R_{n+1} + \gamma V_n(X_{n+1}) - V_n(x)] & x = X_n \\ V_n(x) & x \neq X_n \end{cases} \tag{1.2}$$

where α_n is called the *learning rate*. As one might want to increase the learning rates for rarely visited areas of the MDP, and decrease it for well known parts, it makes sense to allow it to be a function of the history $((X_t, A_t, R_{t+1}), t \in \{0, \dots, n\})$ and thus allow it to be a random variable. We will omit this possible dependency for now and reconsider it in chapter 2.

For a deterministic stationary behaviour π, we can show the analogue

$$\begin{aligned} &\mathbb{E}^\pi[R_{t+1} + \gamma Q(X_{t+1}, A_{t+1}) \mid X_t = x, A_t = a] \\ &= r(x,a) + \gamma \sum_{y \in \mathcal{X}} p(y \mid x, a) Q(y, \pi(y)) \\ &= T^\pi Q(x,a) \end{aligned}$$

FiXme: Do I need to generalize the def?

Note that we could easily extend this to all stationary policies (c.f. **??**, **??**), by defining

$$T^\pi Q(x,a) \coloneqq r(x,a) + \gamma \sum_{(y,b) \in \mathcal{X} \times \mathcal{A}} \mathbb{P}^\pi[X_{t+1} = y, A_{t+1} = b \mid X_t = x, A_t = a] Q(y,b)$$

To distinguish the TD algorithm which calculates Q from the one which calculates V, Sutton and Barto call the first one *Sarsa*. The name stems from the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ where S_t is their notation for the state.

Similar to the Monte Carlo algorithm we use generalized policy iteration to approach π^*. But since we do not need to wait for an episode to finish to create new estimates, we can alternate even more quickly between calculating Q^{π_n} and selecting π_{n+1}. The most extreme form is now updating the policy before every transition in every time step. This is what is called *on-line* learning in contrast to the *off-line* mentioned in the introduction.

Batch Reinforcement Learning is often used synonymously with off-line learning. But as batch learning algorithms were designed to utilize the information of the existing batch as efficiently as possible, there were some attempts to

Algorithm 3 On-line Sarsa (Sutton and Barto 1998)

Assume there is a mapping Ψ assigning an action value function Q a policy π (for example mapping Q on an ε-greedy policy w.r.t. Q).

initialize Q(x,a)
while True **do** (for every episode)
    initialize x as a realization of X_0
    choose a according to $\Psi(Q)(\cdot \mid x)$
    repeat(for each step n of the episode)
        do a, observe reward R and next state y
        choose b according to $\Psi(Q)(\cdot \mid y)$
        $Q(x,a) \leftarrow Q(x,a) + \alpha_n(x,a)[R + \gamma Q(y,b) - Q(x,a)]$
        $x \leftarrow y$; $a \leftarrow b$;
    until x is terminal
end while

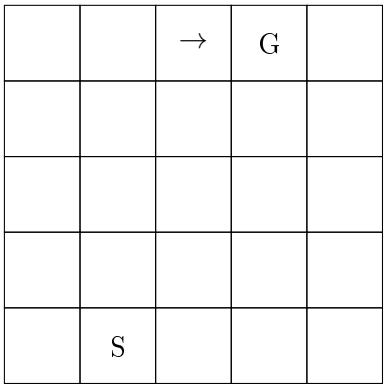
Where α_n is the learning rate.

modify them in order to provide intermediate results to advise action selection, transforming them into on-line algorithms. Lange et al. (2012) suggest to refer to these algorithms as *growing batch reinforcement learning*. We will discuss an example in 1.4 after explaining the the need for it.

1.3.1 Shortcomings of TD

In a way TD is the opposite of Monte Carlo. While Monte Carlo did not use existing estimates of the value function, TD *only* uses existing estimates. This can cause problems as well.

Recall our room navigation example and assume that we initialize the TD algorithm with the value function $Q \equiv 0$. Then the algorithm picks an action, walks into that direction, gets reward zero and updates Q in this place to zero. This continues until it reaches the goal where it updates the state and action that led it to the goal with a positive number. Which makes this action the greedy action (indicated by the arrow).



But in all the previous states TD still does not behave better than before. It will take a couple more episodes to backpropagate the reward of the goal to the Q values at the start. This is quite bad in comparison to Monte Carlo which only takes one episode to leave a trail of breadcrumbs to the goal. One way to tackle this

problem is to try to find a compromise between Monte Carlo and TD which $TD(\lambda)$ tries to do. The other approach is growing batch learning.

1.4 Growing Batch Learning

FiXme: section instead of subsection? chance place? - after TD(lambda), after Q-learning?

One reason for TD's problems is, that it uses sampled transitions only once and then throws them away. This is not so much a problem when the MDP is cheap to sample. But in a lot of real world applications experiences cost more time than most computations, can be expensive (causing damage to hardware) and/or outright dangerous (e.g. self-driving cars). For this reason it is preferable if the algorithm learns as much as possible from a given batch of experiences (transitions). To serve as a comparison, recall our initial naive batch learning algorithm (Algorithm 1). First it estimates the model parameters; after that it uses these estimates multiple times in every Dynamic Programming step. TD's estimates, on the other hand, are baked into the current estimate of V (or Q), which means that older samples are baked into older instances of V which can not really be reused as they also represent previous instances of the "dynamic programming part" of TD (and maybe also previous policies in the GPI case with Q).

So how could we modify this batch learning algorithm (using the entire batch of transitions) to be on-line, i.e. provide provisional estimates and incorporate additional information – growing batches?

First we can notice that we estimate $\hat{r}$ by calculating the algorithmic mean, which we can unwind as in 1.3.1. But note that this increases the number of operations for calculating the mean of n elements from

$$n-1 \text{ additions } +1 \text{ division} = n$$

to

$$(n-1) \times (1 \text{ addition} + 1 \text{ subtraction } +1 \text{ division}) = 3n-3$$

basically tripling the number of operations.

The calculation of $\hat{p}$ is already partly on-line, as the increment in line 8 and 9 can be done in every transition. Recalculation of $\hat{p}$ in every step would entail an iteration through all the $y \in \mathcal{X}$ whichever followed a particular (x, a) and might thus be better left for the time that $\hat{p}$ is actually required.

To accelerate Dynamic Programming we could use the previous estimation of V (Q) as a starting point the next time we use DP. But DP still requires us to go over the entire state space (state-action space) for just one update, and most model parameter stay the same anyway so more local updates are warranted.

The approach of algorithms like "Dyna" are basically hybrids of a "pure" on-line learning algorithm like TD (or – as we will later see – Q learning i.e. Dyna-Q) and a model based learning algorithm which runs in the background –

ideally (for performance) as a separate process – and updates the value function with "localized" dynamic programming, i.e.

$$V^\pi(x) \leftarrow \sum_{a \in \mathcal{X}_x} \pi(a \mid x) \left(\hat{r}(x,a) + \sum_{y \in \mathcal{X}} \hat{p}(y \mid x, a) V^\pi(y) \right) \tag{1.3}$$

or

$$Q^\pi(x,a) \leftarrow \hat{r}(x,a) + \sum_{y \in \mathcal{X}} \sum_{b \in \mathcal{A}_y} \hat{p}(y \mid x, a) \pi(b \mid y) Q^\pi(y,b) \tag{1.4}$$

This leaves us with the question, which states to prioritize for updates. The simplest form of Dyna just picks these states randomly. From our example in 1.3.1 it should be quite obvious, that this is not the most efficient way to backpropagate rewards. But it should not be completely disregarded as it has one of, if not *the* smallest overhead of possible selection algorithms.

A second approach is *prioritized sweeping* where the absolute difference between the sample of $T^\pi V(X_t)$, $R_{t+1} + \gamma V(X_{t+1})$ and $V(X_t)$ (c.f. (1.2)) needs to be larger than a certain threshold for this state to be put into the update queue. The model learning part then updates all the states which lead to this state, and inserts theses states into the queue themselves if their change is large too.

FiXme: write down algorithm?

Another approach is *trajectory sampling* where a trajectory of state is generated by applying the current policy to the model ($\hat{p}$) of the real MDP. Then the algorithm could update the value functions along this chain of states, ideally backwards (to avoid the problem that TD has). This focuses the update on parts of the MDP which are actually relevant as they are possible to reach with the current policy. Just like it makes more sense to analyse different chess games as a joint string of moves instead of analysing random positions which might never come up in a real game. Trajectory sampling can also be combined with the other approach to improve TD. By using the learning algorithm of the mixture $TD(\lambda)$ of TD and Monte Carlo on these simulated trajectories.

The approaches we discussed here so far are all *model based learning*, which could leave the impression that (growing) batch learning is synonymous with model based learning in contrast to *model free learning* methods like Monte Carlo and TD, which do not actually estimate the transition probabilities and skip right to the value functions. But there are also model free growing batch learning methods.

One such example is *experience replay* coined by Lin (1992) which simply creates backups of past transitions and plays them back to the learning algorithms in a possibly different (e.g. backwards) order. The fact, that playing back a random transition is equivalent to sampling the model ($\hat{p}, \hat{r}$), motivates why this has the desired effect if the selection of past transition is not unbalanced. Experience replay can be warranted if the sums in (1.3) or (1.4) are

too large, i.e. when too many different states y can follow a state x. This is sometimes described as a high *branching factor*, where this factor could be defined as the average number of branches. Experience replay essentially trades memory (backing up all transitions) for computation speed, as it does not need to sum over all branches following a state.

Sutton and Barto (1998) call the model based learning methods "planning", and use the term "decision time planning" for algorithms which try to improve the knowledge of $Q(X_t, a)$ for all actions a, just before selecting the action A_t, by exploring the branches of the state based on its current model (essentially applying some localized Dynamic Programming to its model to aid the decision).

1.5 TD(λ) – Mixing Monte Carlo and TD

To try and mix TD and Monte Carlo, recall that Monte Carlo uses

$$\hat{V}^{\infty}(X_t) \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T$$

as the estimator for $V^{\pi}(X_t)$ where T is the length of the episode. And since all these estimates get averaged we can unwind this mean (1.3.1)

$$V_n(X_t) = V_{n-1}(X_t) + \frac{1}{n}(\hat{V}_n^{\infty}(X_t) - V_{n-1}(X_t))$$

On the other hand, recall that TD uses

$$\hat{V}^{(1)}(X_t) \coloneqq R_{t+1} + \gamma V(X_{t+1})$$

as the estimator for $T^{\pi}V(X_t)$ which in turn can be interpreted as an estimator for $V^{\pi}(X_t)$. TD then adjusts its estimates as follows

$$V_n(X_t) = V_{n-1}(X_t) + \alpha_n(\hat{V}_n^{(1)}(X_t) - V_{n-1}(X_t))$$

1.5.1 n-Step Return

FiXme: "corrected n-step truncated return" + explanation?

This leads to the *n-step return* which we define as

$$\hat{V}^{(n)}(X_t) \coloneqq R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(X_{t+n})$$

Since rewards in the terminal state are by convention zero, we know for $n \geq T-t$

$$\hat{V}^{(n)}(X_t) - \hat{V}^{\infty}(X_t) = \gamma^n V(X_{t+n}) = \gamma^n V(X_T)$$

If the value function correctly assigns value 0 to the terminal state X_T, then they are already equal for all $n \geq T - t$, otherwise it converges. This allows us to interpret Monte-Carlo as ∞-step return.

Lemma 1.5.1. *The n-step return has the* error reduction property

$$\left\|\mathbb{E}^\pi[\hat{V}^{(n)}(X_t) \mid X_t = \cdot] - V^\pi\right\|_\infty \leq \gamma^n \|V - V^\pi\|$$

Proof. We inductively prove that

$$\mathbb{E}^\pi[\hat{V}^{(n)}(X_t) \mid X_t = \cdot] = (T^\pi)^n V$$

at which point the claim follows from the contraction property of T^π. The induction basis is just the TD case, c.f. (1.1). The induction step follows

$$\begin{aligned}
&\mathbb{E}^\pi[\hat{V}^{(n)}(X_t) \mid X_t = x] \\
&= \mathbb{E}^\pi[R_{t+1} + \gamma \hat{V}^{(n-1)}(X_{t+1}) \mid X_t = x] \\
&\overset{\text{A.1.1}}{=} \mathbb{E}^\pi[R_{t+1} \mid X_t = x] \\
&\quad + \gamma \sum_{y \in \mathcal{X}} \mathbb{P}^\pi(X_{t+1} = y \mid X_t = x) \underbrace{\mathbb{E}^\pi[\hat{V}^{(n-1)}(X_{t+1}) \mid X_t = x, X_{t+1} = y]}_{\overset{\text{Markov+ind.}}{=} (T^\pi)^{n-1} V(y)} \\
&= \mathbb{E}^\pi[r(x, A_0) \mid X_0 = x] + \gamma \sum_{y \in \mathcal{X}} \mathbb{P}^\pi(X_{t+1} = y \mid X_t = x)(T^\pi)^{n-1} V(y) \\
&= (T^\pi)^n V(x)
\end{aligned}$$

□

1.5.2 TD(λ) Forward View

In that sense any kind of convex combination of n-step returns are estimates for V^π with a minimal error reduction of γ. This is true in particular for the geometric average

$$\hat{V}^\lambda(X_t) \coloneqq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{V}^{(n)}(X_t)$$

Why is the geometric average interesting? Well imagine incrementing over the time steps after t and at every step deciding, whether to call it a day (using the current value function as a replacement for the rest of the returns) or to continue. Since any time looks equally good or bad to stop, there is not really a reason to pick any particular step. So why not continue at every step with equal probability λ? Let N be the associated random variable which represents the stop time. Then assuming a realized MDP, denoting the realized chain of states by x_t, we observe

$$\mathbb{E}[\hat{V}^{(N)}(x_t)] = \sum_{n=1}^{\infty} \mathbb{P}(N = n) \hat{V}^{(n)}(x_t) = \sum_{n=1}^{\infty} (1 - \lambda) \lambda^{n-1} \hat{V}^{(n)}(x_t) = \hat{V}^\lambda(x_t)$$

At this point it might be helpful to note, that for $\lambda = 0$ we get our 1-step basic TD estimate. For this reason TD is the special case TD(0). To continue this

definition for $\lambda = 1$, we expand the estimation as follows

$$\begin{aligned}
\hat{V}^{\lambda}(X_t) &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\hat{V}^{(n)}(X_t) \\
&= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\left(\sum_{j=0}^{n-1}\gamma^j R_{t+1+j} + \gamma^n V(X_{t+n})\right) \\
&\overset{\text{Fub.}}{=} \sum_{j=0}^{\infty}\gamma^j R_{t+1+j}(1-\lambda)\underbrace{\sum_{n=j+1}^{\infty}\lambda^{n-1}}_{=\sum_{n=j}^{\infty}\lambda^n=\left(\frac{1}{1-\lambda}-\sum_{n=0}^{j-1}\lambda^n\right)} + (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V(X_{t+n}) \\
&= \sum_{j=0}^{\infty}\gamma^j R_{t+1+j}\left(1-(1-\lambda)\frac{1-\lambda^j}{1-\lambda}\right) + (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V(X_{t+n}) \\
&= \sum_{j=0}^{\infty}\gamma^j R_{t+1+j}\lambda^j + (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V(X_{t+n}) \qquad (1.5) \\
&\xrightarrow{\lambda\to 1} \sum_{j=0}^{\infty}\gamma^j R_{t+1+j}
\end{aligned}$$

This is simply the return after t. The expected Return is guaranteed to be well defined by Assumption **??**, but by now it is probabily sensible to tighten this assumption to certain boundedness.

FiXme: check claim, should I change assumption 1 in the first place?

The resulting algorithm (1.6) is called TD(λ). The notation (1.7) will be useful later.

$$V_{t+1}(X_t) = V_t(X_t) + \alpha_t(X_t)\underbrace{[\hat{V}_t^{\lambda}(X_t) - V_t(X_t)]}_{=:\Delta V_t^{\lambda}(X_t)} \qquad (1.6)$$

$$= V_0(X_t) + \sum_{k=0}^{t}\alpha_k(X_k)\Delta V_k^{\lambda}(X_k)\mathbb{1}_{X_k=X_t} \qquad (1.7)$$

The index t indicates that $\hat{V}^{\lambda}$ uses V_t for its estimation of the remaining return. TD(0) is then just TD and TD(1) is every-visit Monte Carlo for appropriate α_t. To be more precise Sutton and Barto (1998) call this the *forward view* of TD(λ). This stems from the fact, that we update the value of V in the place x_t with *future* returns, which forces us to wait for the end of the episode just like with Monte Carlo. For this reason this forward view of TD(λ) is not actually the version of TD(λ) which would actually be implemented. The forward view is simply easier to handle in convergence proofs, which is why it is also called the theoretical view of TD(λ).

1.5.3 TD(λ) Backward View

To motivate the *backward view* of TD(λ), we transform $\Delta V_t^\lambda(X_t)$ using (1.5)

$$\begin{aligned}
\Delta V_t^\lambda(X_t) &= \sum_{n=0}^{\infty}(\gamma\lambda)^n R_{t+1+n} + (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\gamma^n V_t(X_{t+n}) - V_t(X_t) \\
&= \sum_{n=0}^{\infty}(\gamma\lambda)^n R_{t+1+n} + \sum_{n=1}^{\infty}(\lambda\gamma)^{n-1}\gamma V_t(X_{t+n}) - \sum_{n=0}^{\infty}(\lambda\gamma)^n V_t(X_{t+n}) \\
&= \sum_{n=0}^{\infty}(\gamma\lambda)^n \left[R_{t+1+n} + \gamma V_t(X_{t+1+n}) - V_t(X_{t+n})\right] \\
&= \sum_{n=t}^{\infty}(\gamma\lambda)^{n-t} \left[R_{n+1} + \gamma V_t(X_{n+1}) - V_t(X_n)\right] \\
&\approx \sum_{n=t}^{\infty}(\gamma\lambda)^{n-t} \underbrace{\left[R_{n+1} + \gamma V_n(X_{n+1}) - V_n(X_n)\right]}_{=\Delta V_n^0(X_n)}
\end{aligned}$$

Where $\Delta V_n^0(X_n)$ is simply the update direction of TD(0). Now why does this approximation make sense?

First of all, V_{n+1} stays the same to V_n in every place but X_n. Assuming that the sequence of states wanders about a little bit, the first few V_n will all be equal to V_t. And once the sequence of states comes back around, the hope is that $n-t$ would be large enough, for $(\gamma\lambda)^{n-t} \approx 0$. Additionally, the difference between V_n and V_t is small, even for repeated states, if the learning rate α_n is sufficiently small.

Now to the question, why is this approximation useful for us? Well TD(0) is already on-line, as the updates happen with every step. So we could just "broadcast" the update deltas $\Delta_n^0(X_n)$ back to previous states, which can then get updated in the same direction. How much they get updated depends on the number of visits and the time between visits. Imagine the state X_t is never visited again afterwards. Then over time it receives the $\Delta_n^0(X_n)$ following X_t which get added to $V_n(X_t)$ with the factor $(\gamma\lambda)^{n-t}$. Then they eventually sum to something akin to $\Delta V_t^\lambda(X_t)$. The backwards view essentially tries to assign discounted credit/blame to previous states. Reducing λ translates to a faster fading of the *eligibility* for credit/blame. In the same manner the factor

$$e_n(x) = \sum_{k=0}^{n}\alpha_k(X_k)(\gamma\lambda)^{n-k}\mathbb{1}_{X_k=x}$$

is called *eligibility trace* of state x for $\Delta V_n^0(X_n)$.

Using (1.7) we will now show that, assuming the same V_0, the estimate V_T of the the forward view will be the same as the estimate generated by summing

over the TD(0) deltas $\Delta V_n^0(X_n)$ multiplied with the eligibility trace.

$$\begin{aligned}
V_T(x) - V_0(x) &= \sum_{k=0}^{T-1} \alpha_k(X_k) \mathbb{1}_{X_k=x} \Delta V_k^\lambda(X_k) \\
&\approx \sum_{k=0}^{T-1} \alpha_k(X_k) \mathbb{1}_{X_k=x} \sum_{n=k}^{\infty} (\gamma\lambda)^{n-k} \Delta V_n^0(X_n) \\
&= \sum_{k=0}^{T-1} \alpha_k(X_k) \mathbb{1}_{X_k=x} \sum_{n=k}^{T-1} (\gamma\lambda)^{n-k} \Delta V_n^0(X_n) \\
&\overset{\text{Fub.}}{=} \sum_{n=0}^{T-1} \Delta V_n^0(X_n) \sum_{k=0}^{n} \alpha_k(X_k) (\gamma\lambda)^{n-k} \mathbb{1}_{X_k=x} \\
&= \sum_{n=0}^{T-1} \Delta V_n^0(X_n) e_n(x)
\end{aligned}$$

Note that this could easily be adapted for non episodic MDPs by replacing T with ∞, where V_∞ is the estimate of the value function in the limit.

Algorithm 4 On-line TD(λ) (Backward View)

```
Initialize V
e(x) ← 0   ∀x ∈ 𝒳
while True do
    Initialize x as a realization of X_0
    repeat(for every step n of the episode)
        a ← action sampled from π(· | x)
        Take action a, observe reward R, next state x'
        Δ ← R + γV(x') − V(x)
        e(x) ← e(x) + α_n(x)
        for y ∈ 𝒳 do
            V(y) ← V(y) + e(y)Δ
            e(y) ← γλe(y)
        end for
        x ← x'
    until x is terminal
end while
```

Just like with Monte Carlo and TD the action value (Q) version Sarsa(λ) is basically the same, and generalized policy iteration would again be used to find an optimal policy.

Note that I moved the α_n into the eligibility trace to allow for a changing learning rate over n. Sutton and Barto (1998) place the learning rate outside the eligibility trace which forces it to be constant in order to translate to the forward view.

1.5.4 True Online TD(λ)

1.6 Q-learning

Q-learning is virtually the same as Sarsa, just that Q-learning tries to skip policy iteration, by trying to approximate Q^* directly, instead of Q^π. As it does not approximate the value function of the policy it uses currently, it is an *off-policy* algorithm, while Monte Carlo and TD are *on-policy* algorithms. Instead of

$$Q(X_t, A_t) \leftarrow Q(X_t, A_t) + \alpha[R_{t+1} + \gamma Q(X_{t+1}, A_{t+1}) - Q(X_t, A_t)]$$

here the update rule is

$$Q(X_t, A_t) \leftarrow Q(X_t, A_t) + \alpha[R_{t+1} + \gamma \max_{b \in \mathcal{A}_{X_t}} Q(X_{t+1}, b) - Q(X_t, A_t)]$$

because the expected value equals

$$\begin{aligned}
&\mathbb{E}[R_{t+1} + \gamma \max_{b \in \mathcal{A}_{X_t}} Q(X_{t+1}, b) \mid X_t = x, A_t = a] \\
&= r(x, a) + \gamma \sum_{y \in \mathcal{X}} p(y \mid x, a) \max_{b \in \mathcal{A}_x} Q(y, b) \\
&= T^* Q(x, a)
\end{aligned}$$

instead of $T^\pi Q$ like in the case of Sarsa.

1.6.1 Shortcomings of Q-learning

Due to its heritage, it displays the same weakness as TD (1.3.1), which we can fix with model learning algorithms like Dyna-Q, as hinted at in 1.4. But the approach of TD(λ) generalizes only to some degree to Q-learning.

To create n-step estimates, the algorithm would need to actually pick the greedy action in every step. But since we also need to do some exploration this can not be guaranteed. Watkin's Q(λ) therefore limits itself to n-steps with n smaller than the number of steps until the next exploratory action. Peng's Q(λ) essentially pretends this problem does not exist and uses histories with non-greedy actions too. This means it "converges to something between Q^* and Q^π", the hope is that it would converge to Q^* if the policy becomes more and more greedy over time. While there is no proof that Peng's Q(λ) converges, most studies have shown it to perform "significantly better" than Watkin's empirically (Sutton and Barto 1998, p. 184). So we are now again at a point, where we simply do not know whether it converges or not.

FiXme: Quote papers, write down pseudo algos?

But at first glance Q-learnings bigger flaw seems to be its tendency to overestimate itself. As it directly estimates Q^* it's greedy actions assume that the optimal policy will be played afterwards. But given an ε-greedy policy

that is not always the case. Sutton and Barto's example is a Cliff Edge, where the shortest Path is right beside the cliff, but a "safer path" is only a little bit longer. Q-learning tries to walk the short path but plunges down the cliff relatively often because of the ε part of its ε-greedy policy. While Sarsa with GPI converges to walking the safer path since it estimates Q^π account for the exploratory part of the policy. But this flaw could also be addressed with more sophisticated exploration policies.

1.7 Exploration

Until now, the only policy we discussed, which tries to tackle the exploration vs. exploitation trade-off, is the ε-greedy policy. This section is meant to give an overview over different approaches to exploration. For on-policy algorithms, convergence is tricky to discuss, as was already the case for the ε-greedy policy. For off-policy Algorithms, in particular Q-learning, the exploration policy only needs to guarantee that every state action pair is visited often enough (i.e. infinitely many times in the limit). But to allow for the use of a Q(λ) algorithm, the greedy actions need to be picked on most occasions.

1.7.1 Optimism

An extremely simple approach is, to initialize the estimate of the action value function Q higher than $R/(1-\gamma)$ (c.f. Assumption **??**). Then unexplored state action pairs have this over-optimistic value, and exploring states lowers their value down towards their actual value. The algorithm will therefore always select the least explored actions until expectations are lowered. While it is very easy to show convergence of a simple greedy policy iteration, it is virtually useless in our setting of large state and action spaces, since it will only exploit its knowledge once it thoroughly convinced itself that there are no better actions.

1.7.2 Boltzmann Exploration

The ε-greedy policy is also known as *semi-uniform random exploration*, because it selects a random action uniformly, when exploration is rolled with probability ε. Boltzmann exploration does not just differentiate between exploratory actions and greedy actions, but weights actions depending on the estimation of their value

$$\pi(a \mid x) = \frac{\exp(Q(x,a)/T)}{\sum_{b \in \mathcal{A}_x} exp(Q(x,b)/T)}$$

where $T > 0$ is a "temperature" parameter, which can be used to change the algorithms tendency for exploration. Decreasing T, decreases exploration. For

$x^* = \max_{i=1,\dots,n} x_i$ we can see that in limit

$$\begin{aligned}
\exp(x_i/T) &= \exp\left(\frac{x^*}{T} + \frac{x_i - x^*}{T}\right) = \exp(x^*/T)\exp((x_i - x^*)/T) \\
\implies \lim_{T\to 0} \frac{\exp(x_i/T)}{\sum_{i=1}^n \exp(x_i/T)} &= \lim_{T\to 0} \frac{\exp(x_i/T)}{\exp(x^*/T)} \frac{1}{\sum_{i=1}^n \exp((x_i - x^*)/T)} \\
&= \lim_{T\to 0} \frac{\exp((x_i - x^*)/T)}{\sum_{i=1}^n \exp((x_i - x^*)/T)} \\
&= \begin{cases} 0 & x_i \leq x^* \\ \frac{1}{\#\{j|x_j = x^*\}} & x_i = x^* \end{cases}
\end{aligned}$$

While increasing T, increases exploration

$$\lim_{T\to\infty} \frac{\exp(x_i/T)}{\sum_{i=1}^n \exp(x_i/T)} = \frac{1}{n}$$

1.7.3 Directed Exploration

Boltzmann exploration fixes most of the suicidal tendencies that the ε-greedy policy expresses in our cliff edge example (in 1.6.1). But since Boltzmann exploration still assigns every action a positive probability, it is still considered an *undirected* exploration method. The following methods are *directed* exploration methods.

Interval Estimation

Kaelbling (1993) suggested to use confidence intervals in more simple decision problems like the n-armed bandit. In the n-armed bandit the agent has to decide between n options which have stochastic returns. Sampling these options allows us to estimate their means and construct confidence intervals around them. Choosing the action with the highest upper bound of its confidence interval accounts for the trade-off between picking actions with high mean, and less explored actions with large confidence intervals. Consequently modifications where proposed which would allow for the construction of confidence intervals around the $Q(x, a)$ values in model based learning (e.g. Wiering and Schmidhuber 1998; Strehl and Littman 2008). It is probably futile trying to apply interval estimation to pure on-line learning, since at least much of the early variation in the Q values comes from "dynamic programming" and not stochastic uncertainty.

The basic idea behind model based interval estimation (MBIE) is, to create confidence intervals around $\hat{r}$ and $\hat{p}$. This is relatively easy for $\hat{r}(x, a)$ as it is simply a mean of reward samples. And assuming boundedness of rewards we can use Hoeffding's inequality. To avoid the boundedness assumption we

could use for example, that independently identical distributed (iid) X_i have the property

$$\mathrm{Var}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\mathrm{Var}[X_1] \quad \text{and} \quad \hat{\sigma}_n \coloneqq \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2} \to \sigma_{X_1}$$

so we could use $\hat{\sigma}_n/\sqrt{n}$ as an estimator for the standard deviation of $\overline{X}$. And since $\overline{X}$ is approximately normal distributed because of the central limit theorem, the standard deviation is enough to construct a simple confidence interval. We denote this confidence interval with

$$CI(r_{x,a}) \coloneqq (\hat{r}(x,a) - \varepsilon^r, \hat{r}(x,a) + \varepsilon^r)$$

In the same spirit Strehl and Littman (2008) define

$$CI(p_{x,a}) \coloneqq \left\{ p \in [0,1]^{|\mathcal{X}|} : \sum_{y \in \mathcal{X}} p(y) \text{ and } \|p - \hat{p}(\cdot \mid x,a)\|_1 \leq \varepsilon^p \right\}$$

with an appropriately selected ε^p. This is of course less of a confidence *interval* and should maybe rather be called a confidence set.

Their idea is, that if we take the "upper limit" out of both "intervals" we get something akin to the upper limit $\tilde{Q}$ of the "confidence interval" for Q which would satisfy the recursion

$$\tilde{Q}(x,a) = \max_{R \in CI(r_{x,a})} R + \max_{p \in CI(p_{x,a})} \gamma \sum_{y \in \mathcal{X}} p(y) \max_{b \in \mathcal{A}_y} \tilde{Q}(y,b)$$

They show that this recursion is still a contraction, which means we could calculate $\tilde{Q}$ with the same (local) dynamic programming ideas which we use in model based learning. While the question whether or not this construct is in fact a confidence interval around Q is not very pressing (since we are not interested in confidence intervals per se but rather exploration heuristics), the computational burden of a second recursion on top of the main recursion is more of a problem. Especially since this recursion includes two more maximizations, and while the first one is trivial once ε^r is calculated, the second one is not. Whether or not this is acceptable depends on the computational resources available and on the expensiveness of samples from the MDP.

Alternatively one could try to simplify the estimation. Wiering and Schmidhuber (1998) simply forego the recursion and only use bounds around $\hat{r}$ and $\hat{p}$, ignoring the estimation error in the Q values of the following states. Other approaches simply award exploration bonuses based on the frequency of tries, or on how recent the last try was.

FiXme: Dyna-Q+ ?

Another problem is, that interval estimation allows the upper limit of the confidence interval around the optimal action to be lower than the average

value of another action with positive probability. This unlikely case results in the algorithm never trying these optimal actions again. This is called sticking (Kaelbling 1993, p. 61). Last but not least there is the question on how to initialize the size of the intervals when no samples (or too few) are available. Wiering and Schmidhuber (1998) suggest to begin with Boltzmann exploration, and switch to interval estimation later.

Bayesian Exploration

Dearden et al. (1998) suggest to select actions based on the probability that they are optimal, conditional on the samples collected. But just like with every Bayesian approach, calculating conditional probabilities requires a prior distribution of the Q values. This results in more parameters with which the algorithm can be tuned. Therefore the authors warn that this could have benefitted the performance of this algorithm in their empirical comparison. And while their algorithm was quite competitive, they also note that it is very computationally expensive.

1.7.4 Intrinsic Rewards

Most exploration approaches struggle, when the rewards for actions have a long delays (i.e. when rewards are sparse). Recall our room navigation example again, and imagine a second goal $\tilde{G}$ closer to the start with a smaller reward. Then the algorithm will most likely stumble into the secondary goal first and the action values around this secondary goal will relatively quickly be updated to lead towards this goal.

In case of the ε-greedy policy, the occasional exploratory actions will only cause a one step deviation from the shortest path to this secondary goal, and the algorithm will use its knowledge about the surroundings to quickly walk back towards this goal using subsequent greedy actions. While finding the larger goal, requires multiple subsequent exploratory actions, leading away from the secondary goal.

			G	
	↓			
↓	←	↓		
$\tilde{G}$	←	←	←	
↑	S	←		

Boltzmann exploration does not perform much better, as the actions leading to the secondary goal will be assigned higher probability, so finding the larger goal still requires a chain of unlikely actions.

Since estimating confidence intervals around Q values without samples is not really possible, the performance of interval estimation depends on how it handles unknown states. If it uses Boltzmann exploration in the beginning,

it will obviously struggle too. And setting high ranges for the initial interval estimations results in very similar behaviour to optimistic initialization of the Q values. As the algorithm would then explore until the entire state space is explored.

A naive fix to this problem would be to favour chains of exploratory actions (i.e. increase the likelihood of an exploratory action following another), or similar adjustments. An out-of-the-box solution is, to modify the existing rewards, artificially making them less sparse.

Curiosity-Driven Exploration

Inspired by human curiosity, intrinsic rewards for actions which do not immediately result in extrinsic reinforcement were suggested. There are mostly two types of intrinsic motivation algorithms. One rewards reaching "novel" states, encouraging the agent to stay in unexplored territory before walking back to known extrinsic rewards. Where "novelty" could be a constant minus the number of visits, or anything else one might come up with.

In the second approach the agent builds a world model which tries to predict the next state x_{t+1} given state x_t and action a_t. This world model would be a function approximation algorithm (e.g. an artificial neuronal net) which would be trained on the samples generated. Surprising this world model results in an additional reward for the reinforcement algorithm. This encourages the agent to take actions which it does not understand (i.e. which the world model does not predict). But in this basic form, the agent would get addicted to white noise, i.e. situations where the next state is completely unpredictable but also not necessarily meaningful.

FiXme: Citation of earlier work

Pathak et al. (2017) recently came up with a modification to fix this problem. They argue that the unpredictability is due to the fact, that these unpredictable state changes are not influenced by the agents actions. So they suggest a filter Φ for the state space, which maps the state to a smaller feature space, which only represents the influenceable state of the world. This feature map Φ is attained by training a neuronal net.

FiXme: diagram here

This neuronal net consists of two copies of a neuronal net with input neurons x_t and x_{t+1} respectively, which have fewer output neurons. These two outputs $\Phi(x_t)$ and $\Phi(x_{t+1})$ are then fed into a common net which has $\hat{a}_t$ as output neurons. Similar to autoencoders compressing information with a bottleneck, this neuronal net has to compress x_t and x_{t+1} to the middle layer $\Phi(x_t)$ and $\Phi(x_{t+1})$. But since it does not need to recover the original from this compression but rather a_t, it is meant to throw away the information about x_t which does not help to predict a_t, i.e. throw away parts of the state space which can not be influenced by an action.

The world model then has to predict $\Phi(x_{t+1})$ instead of x_{t+1}, which means that its error will not increase, when it does not predict non-influenceable changes like white noise as these are filtered out by Φ.

This set up proved extremely effective in empirical tests. It even learned to play some games it was tested on without any extrinsic reward at all.

It might be argued, that this feature map Φ filters out too many things, like actions by other actors, which are not influenced by the agent itself but influence the agent. If they influence the agent though, and the agent could or should react to these actions, then these events will be relevant for predicting the agents actions. Therefore they would not be filtered out. Only if the agent can not or has no incentive to respond to these events is it irrelevant for predicting the agents actions and will be filtered. And since the agent could not or does not need to react to these event anyway, no harm is done.

1.8 Function Approximation

Chapter 2

Stochastic Approximation – Convergence Proofs

Appendix A

Appendix

A.1 Basic Probability Theory

Lemma A.1.1.

(i) $\mathbb{P}(A \cap B \mid C) = \mathbb{P}(A \mid B \cap C)\mathbb{P}(B \mid C)$

(ii) $\mathbb{P}(A \mid C) = \sum_{n\in\mathbb{N}} \mathbb{P}(A \mid B_n \cap C)\mathbb{P}(B_n \mid C)$ *for* $\mathbb{P}\left(\biguplus_{n\in\mathbb{N}} B_n\right) = 1$

(iii) $\mathbb{E}[X \mid C] = \sum_{n\in\mathbb{N}} \mathbb{E}[X \mid C \cap B_n]\mathbb{P}(B_n \mid C)$ *for* $\mathbb{P}\left(\biguplus_{n\in\mathbb{N}} B_n\right) = 1$

Proof. (i)

$$\mathbb{P}(A \cap B \mid C) = \frac{\mathbb{P}(A \cap B \cap C)}{\mathbb{P}(B \cap C)} \frac{\mathbb{P}(B \cap C)}{\mathbb{P}(C)} = \mathbb{P}(A \mid B \cap C)\mathbb{P}(B \mid C)$$

(ii)

$$\begin{aligned}\mathbb{P}(A \mid C) &= \mathbb{P}\left(A \cap \biguplus_{n\in\mathbb{N}} B_n \;\middle|\; C\right) = \sum_{n\in\mathbb{N}} \mathbb{P}(A \cap B_n \mid C) \\ &\overset{\text{(i)}}{=} \sum_{n\in\mathbb{N}} \mathbb{P}(A \mid B_n \cap C)\mathbb{P}(B_n \mid C)\end{aligned}$$

(iii)

$$\begin{aligned}\mathbb{E}[X \mid C] &= \frac{1}{\mathbb{P}(C)} \int_C X d\mathbb{P} = \sum_{n\in\mathbb{N}} \frac{1}{\mathbb{P}(C)} \int_{C\cap B_n} X d\mathbb{P} \\ &= \sum_{n\in\mathbb{N}} \frac{\mathbb{P}(C \cap B_n)}{\mathbb{P}(C)} \frac{1}{\mathbb{P}(C \cap B_n)} \int_{C\cap B_n} X d\mathbb{P} \\ &= \sum_{n\in\mathbb{N}} \mathbb{E}[X \mid C \cap B_n]\mathbb{P}(B_n \mid C)\end{aligned}$$

□

Lemma A.1.2. *Let $(\Omega, \mathcal{A}, \mu)$ be a measure space and a function f exists with*

$$f\colon \Omega \to \mathbb{R} \text{ injective and measureable,}$$
$$f^{-1}\colon f(\Omega) \to \Omega \text{ measureable.}$$

Then for X Ω-valued random variable and Y $f(\Omega)$-valued random variable

$$\mathbb{P}_{f\circ X} = \mathbb{P}_Y \iff \mathbb{P}_X = \mathbb{P}_{f^{-1}\circ Y}$$

Proof. "$\Leftarrow$" Let $A \in \mathcal{B}(\mathbb{R})$, then w.l.o.g. $A \subseteq f(\Omega)$ otherwise

$$\mathbb{P}_{f\circ X}(A) = \mathbb{P}(A \cap f(\Omega)) + \underbrace{\mathbb{P}_{f\circ X}(A \cap f(\Omega)^{\complement})}_{=0} = \ldots = \mathbb{P}_Y(A)$$

Thus $f \circ f^{-1}(A) = A$ holds, which finishes this direction with

$$\begin{aligned}\mathbb{P}_{f\circ X}(A) &= \mathbb{P}(X^{-1} \circ f^{-1}(A)) = \mathbb{P}_X(f^{-1}(A)) \\ &= Pr_{f^{-1}\circ Y}(f^{-1}(A)) = \mathbb{P}(Y^{-1} \circ f \circ f^{-1}(A)) \\ &= \mathbb{P}_Y(A)\end{aligned}$$

"$\Rightarrow$" Let $A \in \mathcal{A}$, then

$$\begin{aligned}\mathbb{P}_X(A) &= \mathbb{P}(X^{-1} \circ f^{-1} \circ f(A)) = \mathbb{P}_{f\circ X}(f(A)) \\ &= \mathbb{P}_Y(f(A)) = \mathbb{P}(Y^{-1} \circ f(A)) \\ &= \mathbb{P}_{f^{-1}\circ Y}(A)\end{aligned}$$

□

Definition A.1.3 (Pseudo-inverse)**.** Let F be a cumulative distribution function, then

$$F^{\leftarrow}(y) \coloneqq \inf\{x \in \mathbb{R} : F(x) \geq y\}$$

is called the *Pseudo-inverse* of F.

Lemma A.1.4. *Let F be a cdf, then*

(i) $F^{\leftarrow}(y) \leq x \iff y \leq F(x)$

(ii) $U \sim \mathcal{U}(0,1) \implies F^{\leftarrow}(U) \sim F$

Proof. (i) "$\Rightarrow$"

$$\begin{aligned}y &\leq \inf_{x \in \{z \in \mathbb{R} : F(z) \geq y\}} F(x) \overset{\text{F right-continuous}}{=} F(\inf\{z \in \mathbb{R} : F(z) \geq y\}) \overset{\text{def.}}{=} F(F^{\leftarrow}(y)) \\ &\leq F(x)\end{aligned}$$

Where the last inequality follows from the assumption $F^{\leftarrow}(y) \leq x$ and F being non decreasing.
"$\Leftarrow$" Follows simply from the fact that x is included in the set of the infimum.

$$y \leq F(x) \implies F^{\leftarrow}(y) = \inf\{z \in \mathbb{R} : F(z) \geq y\} \leq x$$

(ii) is a simple corollary from (i)

$$\mathbb{P}(F^{\leftarrow}(U) \leq x) \overset{\text{(i)}}{=} \mathbb{P}(U \leq F(x)) = F(x)$$

□

A.2 Analysis

Bibliography

Dearden, Richard, Nir Friedman, and Stuart Russell (1998). "Bayesian Q-Learning". In: p. 8.

Kaelbling, Leslie Pack (1993). *Learning in Embedded Systems*. MIT press.

Lange, Sascha, Thomas Gabel, and Martin Riedmiller (2012). "Batch Reinforcement Learning". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Adaptation, Learning, and Optimization. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 45–73. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_2. URL: https://doi.org/10.1007/978-3-642-27645-3_2 (visited on 02/22/2019).

Lin, Long-Ji (May 1, 1992). "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". In: *Machine Learning* 8.3, pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699.

Pathak, Deepak et al. (July 2017). "Curiosity-Driven Exploration by Self-Supervised Prediction". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). Honolulu, HI, USA: IEEE, pp. 488–489. ISBN: 978-1-5386-0733-6. DOI: 10.1109/CVPRW.2017.70.

Strehl, Alexander L. and Michael L. Littman (Dec. 1, 2008). "An Analysis of Model-Based Interval Estimation for Markov Decision Processes". In: *Journal of Computer and System Sciences*. Learning Theory 2005 74.8, pp. 1309–1331. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2007.08.009.

Sutton, Richard S. (Aug. 1, 1988). "Learning to Predict by the Methods of Temporal Differences". In: *Machine Learning* 3.1, pp. 9–44. ISSN: 1573-0565. DOI: 10.1007/BF00115009.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. Cambridge, Mass. [u.a.]: MIT Press. xviii+322. ISBN: 978-0-262-19398-6.

— (2018). *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press. 526 pp. ISBN: 978-0-262-03924-6.

Wiering, Marco and Jürgen Schmidhuber (1998). "Efficient Model-Based Exploration". In: *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. Vol. 6, pp. 223–228.