An overview of "*Reinforcement Learning: An Introduction (2nd edition)*"

Nichita Uțiu, Romanian Institute of Science and Technology, Cluj-Napoca, Romania

This is a collection of notes on Sutton and Barto's book *"Reinforcement Learning: An Introduction (2nd edition)"* from 2018. These are some of the main takeaways from the book structured on a per-chapter basis with a main focus on the theoretical aspects. These can be used as a support for the book.

Note: The numbers in round brackets in the headings indicate the chapters of the book the notes correspond to.

(1)

Part I - Tabular methods

(3) Finite Markov Decision Processes (MDPs)

A Markov decision process is the quadruplet (S, A, R, p) where S, A, R are the *state*, *action* and *reward* spaces respectively, while p is a function of 4 arguments:

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = t \mid S_{t-1} = s, A_{t-1} = a\} \tag{2}$$

where $s, s' \in S$, $r \in R$ and $a \in A$. The values are in the interval $[0, 1]$ and it has the following property:

$$\forall s \in S \forall a \in A(s) \left(\sum_{s' \in S} \sum_{r \in R} p(s', r \mid s, a) = 1 \right) \tag{3}$$

This means that for all state-action pairs (s, a), p represents a **probability distribution** of next states and rewards. Another commonly used notation is the **state-transition probability** $p(s'|s, a)$ which is just the marginal probability of transition to a next state:

$$p(s'|s, a) = \sum_{r \in R} p(s', r|s, a) \tag{4}$$

We can also compute the **expected reward** for a state-action pair:

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a) \tag{5}$$

As well as the expected reward **given the next step**:

$$r(s, a, s') = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, St = s'] = \sum_{r \in R} r \sum_{s' \in S} \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{6}$$

If we consider a task where the MDP has no terminal state and actions can be take from every state, we can develop the concept of a cumulative reward called **return** . More often than not, we add a **discount coefficient** $\gamma \in [0, 1]$ to the value of the return from timestep $t + 1$ and get the recursive formula:

$$G_t = R_t + \gamma G_{t+1} = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{7}$$

Note: The continuous formulation (episodes never end) is often desired in proofs, but many tasks are episodic in nature. To unify them, we can **consider the terminal state to be an absorbing state** where all action loop back deterministically with 0 reward.

Policies and value functions

By definition, **a policy is a conditional probability distribution** of actions to be taken give a certain state. It is written as $\pi(a|s)$. This definition gives rise to the concept of **state value function** and **state-action value function**. These simply denote the expected return in a given state, or state-action pair for an agent which samples actions according to some policy π.

$$\begin{aligned}
v_\pi(s) &\triangleq \mathbb{E}_\pi[G_t \mid S_t = s] = \sum_{a \in A(s)} \pi(s|a) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) && (8)\\
&= \sum_{a \in A(s)} \pi(s|a) q_\pi(s, a) && (9)\\
q_\pi &\triangleq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \sum_{s' \in S, r \in R} p(s', r|s, a)(r + \gamma \sum_{a' \in A} \pi(a'|s') q_\pi(s', a')) && (10)\\
&= \sum_{s' \in S, r \in R} p(s', r|s, a)(r + \gamma v_\pi(s')) && (11)
\end{aligned}$$

As we can see, both q_π and v_π can be written in terms of one another. They can also be written as a recursive function of themselves. This recursive relations are called **Bellman equations** (see page 59).

Given these expected values for returns, we can define a **partial ordering relation** on the set of policies:

$$\pi \geq \pi' \leftrightarrow \forall s \in S(v_\pi(s) \geq v_{\pi'}(s)) \tag{12}$$

This ordering implies there is a set of policies which are **optimal** (the greatest elements of the set of policies) and for which we have the following optimal state and state-action value function:

$$\begin{aligned}
v_*(s) &\triangleq \max_\pi v_\pi(s) \\
q_*(s, a) &\triangleq \max_\pi q_\pi(s, a)
\end{aligned} \tag{13}$$

Note: The optimal value functions also have Bellman equations with the non-optimal version of the function exchanged for the optimal one (q_* instead of q_π and v_* instead of q_π) and with maximization wrt. π.

---

(4) Dynamic Programming (DP)

A lot of RL algorithms revolve around a framework called **General Policy Iteration (GPI)** which involves 2 steps:

1. **Evaluation:** Taking a policy π and estimating the value of v_π. The obtained estimate is often called V.
2. **Improvement:** Given the estimate V, improve the policy π.

If we were to write the Bellman equation for every state in S, we would have a system of $|S|$ linear equations and unknowns which could be solved, however, given the size of the state space this is often inconvenient.

Iterative Policy Evaluation

DP does the first step of GPI by **sweeping through the set of all states** and computing for each one the expected return using the Bellman equations and updating the approximate value function. **This sweeping process is repeated** until the maximum change over all the states is lower than a threshold θ (see page 75 for algorithm).

From this point of view, DP requires knowledge of the full model of the environment (ie. $p(s', r|s, a)$). This assumption is a strong one and is not met for many environments used in practice.

Policy improvement theorem

The policy improvement theorem states that, considering the partial ordering relation defined between policies, given 2 deterministic policies π and π' such that $\forall s \in S$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{14}$$

Then $\pi' \geq \pi$ as per the definition of the partial ordering relation over policies. If there is equality, this means that the policies are equal and that **they are both optimal** (see page 78 for theorem and proof).

This theorem has the consequence that the **greedy policy constructed wrt. to another policy is at least as good as that policy**. This gives rise to a process of **policy improvement** by taking the greedy policy wrt. an approximate value function.

The two steps of GPI described before (evaluation and improvement) are therefore the processes of evaluating/approximating the value function given a policy and then updating the policy to be greedy wrt. that value function. **GPI can describe the vast majority of RL algorithms**.

Note: DP methods are **proven to converge** to the optimal policy **if all states are updated an infinity of times in the limit**. As long as this condition is satisfied the states can be updated in any order. This means one **can update them asynchronously and in parallel** (see section 4.5 on page 85).

Policy and value iteration

Until now we've only discussed policy improvement in the framework of DP. The main incarnation of this is **the policy iteration algorithm which alternates between estimation and improvement** (see page 80).

However, this process can be simplified by merging estimation and improvement into a single step defined by the following update rule (ie. **value iteration**):

$$v(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v(s')\right] \tag{15}$$

Devising an algorithm based on this rule is straightforward. **Instead of doing 2 sweeps, we just repeatedly do this combined step** (see page 83). Although slightly different, this algorithm still fits into the GPI family, the difference being that the alternation takes place at a per-state level.

---

(5) Monte Carlo (MC) methods

One of the main limitations of DP algorithms is that they need full knowledge of the dynamics of the environment. MC methods relax this constraint and are part of the family of **model-free methods**. We do however **consider that we only apply MC methods to episodic tasks** in this chapter.

MC methods **revolve around sampling episodes given a policy**. The most basic evaluation algorithm (see page 92) revolves around sampling episodes (sequences of $S_0, A_0, R_1, S_1, A_1, R_2, \dots S_{T-1}, A_{T-1}, R_T$) given a policy π (possibly stochastic), and **computing the average return for each state or state-action pair**. It is often more useful to estimate Q-values rather than state values because from Q-values we can directly infer a greedy policy.

Since the **estimators for the value of each state are independent** and are not a function of other estimates, **MC methods do not bootstrap by definition**.

Note: We can take into consideration just the first visit of the state or all visits of the states in a sampled episode when computing the average returns. These two methods are intuitively named **first-visit** and **every-visit MC**.

Monte Carlo estimation for action values (w/ or w/o exploring starts)

One **can use MC for estimation in the GPI framework and come up with MC control** (ie. using sampled episodes to estimate the true value function of the policy). However, unlike DP methods which consider all possible trajectories, MC methods sample according to a policy, meaning that they are biased and that they **do not maintain exploration**. This has the result that **unless we guarantee exploration, MC methods are not guaranteed to converge** asymptotically to the real value function.

- One method to solve this is to uniformly sample the starting states and start from them if the environment allows. This is called **MC with exploring starts** (see page 99).
- Another method is to **only explore a subset of stochastic policies** for which the probability of any action is non-zero. These are called ε**-soft policies** and we use a subset called ε**-greedy** policies for improvement (see page 101 for more details). ε-greedy policies are policies which are greedy with a probability of $1 - \varepsilon$ and sample uniform otherwise.

Off-policy prediction via Importance Sampling

So far the methods described have been **on-policy** methods, meaning that the policy improved (**target policy**) and the one that decides the actions (**behavior policy**) were the same. If the 2 policies were different, the method would be called **off-policy**.

One way to evaluate target policies wrt. a different behavior one is via Importance Sampling. This consists in weighting the sampled trajectory's return to compensate for the different probabilities of taking the actions wrt. to the behavior policy as opposed to the target one:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{16}$$

If we were to use the unweighted averaged returns of the values, we would be estimating the value function of the behavior policy b: $\mathbb{E}_b[G_t|s] = v_b(s)$. However, **by averaging the weighted returns, with the importance sampling ratios we get an estimator of the values for** π:

$$\mathbb{E}_b[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s) \tag{17}$$

For simplicity's sake, **the book considers that all the sampled episodes are concatenated into a long one** and that we let $T(t)$ denote the first termination timestep after t (the lowest timestep greater or equal to t which is an episode termination). Also, we let $\mathcal{T}(s)$ be the set of timesteps in which s was visited. Given these notations, we can define an estimate of v_π:

$$V(s) \triangleq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{|\mathcal{T}(s)|} \tag{18}$$

This is called **ordinary importance sampling**. It still has uses in more advanced algorithms, but it has the disadvantage that it provides a very high variance estimate. A lower variance version called **weighted importance sampling** exists:

$$V(s) \triangleq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}} \tag{19}$$

Note: One can implement MC prediction efficiently using cumulative sums to compute averages, rather than memorizing all the transitions of an episode (see section 4.6, specifically page 110).

Off-policy Monte Carlo Control

In order to implement off-policy MC control, we can use importance sampling in **conjunction with an ε-soft policy** (ie. ε-greedy wrt. the Q-function) to preserve exploration (see page 111 for algorithm). Exploring starts can be used if they are an option.

Note:

- **Ordinary importance sampling has infinite variance if the sampling ratios are unbounded**, however, it's an unbiased estimator! **Weighted sampling is, on the other hand, biased** but has very low variance (see figure 5.3 and example 5.5 from page 106 for a comparison).
- As an implementation note, MC methods do not need to memorize rewards and states. Cumulative sums and counts can be kept instead (see page 110).
- The lack of bootstrapping in MC methods makes them more robust against environments which violate the Markovian property.
- Other more advanced methods such as **discount-aware importance sampling** and **per-decision importance sampling** can be used to further lower the variance of the estimates (see sections 5.8 and 5.9 - we skipped over these details).

---

(6) Temporal Difference (TD) Learning

Like MC methods, TD learning does not need a model, **but unlike MC, it does bootstrap**. The general update formula for TD estimation is the following:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{20}$$

This kind of update is called a **TD(0) update or single-step TD update**. The term $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called a **TD error** and is multiplied by a **step-size** α to move the estimates in the direction of the real value function. Based on the Bellman equations from the DP chapter, we know this estimate leads to the real value function.

For batch TD and MC (updating on a series of episodes at once), MC is the **least-squares estimator** of the data whereas **TD(0) estimates the values of the maximum likelihood MDP which generates the data**.

TD methods are sound and converge to v_π if the step size decreases according to the stochastic approximation conditions. However, there is no proof of whether TD or MC converges faster. However, **TD methods seem to empirically converge faster on stochastic tasks.**

Another key difference between MC and TD methods is that **TD methods continually learn during the episode** using only single transitions, whereas MC requires several episode samples.

Note: It is also immediately clear **that there is a strong connection between gradient ascent/descent algorithms and TD methods**.

SARSA

Sarsa is an **on-policy TD(0) control method which estimates Q-values**. Sarsa takes 2 actions using a policy, observes the states for both of them, and the reward for the first (the name is actually an acronym for this sequence: **S**tate **A**ction **R**eward State Action). The TD error is then used to learn the Q-values:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \tag{21}$$

As with on-policy MC, we need to use a soft policy to ensure convergence (see page 130 for algorithm).

Q-Learning

Q-learning is a very similar algorithm which uses an update rule very similar to Sarsa, **but it tries to estimate the optimal policy by bootstrapping as if it took an action using the greedy policy**:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)] \tag{22}$$

This sort of bootstrapping **makes Q-learning an off-policy method** which tries to estimate q_* yet takes actions according to π (see page 131 for algorithm). If the actions are taken greedily, Q-learning is almost the same as Sarsa.

Expected Sarsa

Expected Sarsa follows the same pattern as Q-learning, but instead of the maximum next value, **it computes the expectation wrt. to the target policy π of the Q-value for the next state.**

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \mathbb{E}_\pi[Q(S',A')|S] - Q(S,A)] \tag{23}$$
$$= Q(S,A) + \alpha[R + \gamma \sum_a \pi(a|S)Q(S,a) - Q(S,A)] \tag{24}$$

Expected Sarsa is equivalent to Q-learning if the policy is the greedy one, but it has lower variance than Sarsa. Unlike Sarsa, however, **it can be off-policy** (see page 133). Like all methods that depend on expectation, one should have access to the full model or at least be able to perform one lookahead step.

Maximization Bias and Double Q-Learning

For any algorithm which takes actions with a policy (the behaviour policy) which prefers the best value functions (eg. Sarsa with ε-greedy or Q-learning), **using the same value function for both estimation and action leads to maximization bias**. This means that the estimates will be skewed towards the maximal value of returns (see page 135 for example).

To combat this, **we can use 2 value functions for estimation and compute one's estimate to update the other**. We use their average to choose actions and update them alternatively wrt. to a Bernoulli trial:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha(R + \gamma Q_2(S', \arg\max_a Q_1(S',a)) - Q_1(S,A)) \tag{25}$$

Randomly, we then alternate to the equivalent function for Q_2 (see page 136 for algorithm). A similar update rule can be applied to expected Sarsa to get rid of maximization bias:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha(R + \gamma \sum_a \pi(a|S')Q_2(S',a) - Q_1(S,A)) \tag{26}$$

Note: In some cases where many value-action pairs lead to the same state (ie. chess, tic-tac-toe) it is often more useful to design value functions around the steads they lead to - called **afterstates**.

---

(7) n-step Bootstrapping

n-step bootstrapping is a family of algorithms which does temporal difference learning on chains of multiple actions. They **provide a unifying framework for both TD(0) and MC methods** (see page 146 for a diagram). For example, MC methods can be viewed as ∞-step TD. In order to implement them, we define timestep-based versions of the value function and of the bootstrapped returns:

$$G_{t:t+n} \triangleq \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n V_{t+n-1}(S_{t+n}), n \geq 1, 0 \leq t < T - n \tag{27}$$

Given these n-step returns, we have a new update rule for the state-value function:

$$V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t)\right] \tag{28}$$

The algorithm resulting from applying this rule is called **n-step TD** (see page 144). These methods also have the property that **the worst estimation error for the expectation wrt. state is still smaller or equal to the error of the worst state** (see page 144). Because of this guarantee, one can prove the convergence of n-step TD methods.

The return formula can be applied to returns written wrt. the Q-value function. We can then rewrite the update rule for the Q-values as well (see page 147 for algorithm).

$$G_{t:t+n} \triangleq \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), n \geq 1, 0 \leq t < T - n \tag{29}$$

Note: One benefit of using multiple steps for an update is in the case of sparse reward signals, **where only a small handful of states generate non-zero TD-errors** (see figure 7.4 on page 147).

---

n-step Sarsa

Dealing with multistep Q values, we can define an **n-step version** for Sarsa leading to n-step control. n-step Sarsa has the following update rule (see page 147 for algorithm):

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{30}$$

The same method **can be adapted to expected Sarsa** if we replace the last bootstrap estimate with an expectation:

$$G_{t:t+n} \triangleq \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), n \geq 1, 0 \leq t < T - n \tag{31}$$
$$\bar{V}_t(s) \triangleq \sum_a \pi(a|s) Q_t(s, a)$$

Off-policy control with Importance Sampling

Borrowing from MC methods, n-step Sarsa can be further adapted to off-policy control by multiplying the TD error with the **importance sampling ratio** for either state or state-action value function

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t:t+n-1} \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{32}$$
$$\rho_{t:h} \triangleq \prod_{k=t}^{min(h, T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

With this update rule, we can implement off policy **n-step off-policy control** (see page 149).

N-step off-policy Sarsa uses importance sampling for all steps **except for the last one**. Thus, $\rho_{t+1:t+n}$ would be replaced by $\rho_{t+1:t+n-1}$ and, the last state would still use expectation.

Without Importance Sampling - Tree Backup

The Tree Backup algorithm allows us to use n-step TD off-policy without importance sampling, by bootstrapping the **expectation** of the Q-values at each step in the chain (see page 152 for diagram).

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \forall t < T - 1 \tag{33}$$
$$G_{t:t+1} = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a), t = T - 1$$

Basically, the last action's (timestep $T - 1$) return is a normal expectation wrt. π, while the others are computed also taking into account the result of the taken actions (see page 154 for algorithm). This gives rise to the following update rule:

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha (G_{t:t+n} - Q_{t+n-1}(S_t, A_t)) \tag{34}$$

For $n = 1$, tree backup is equivalent to expected Sarsa.

---

n-step $Q(\sigma)$

This algorithm is a **generalization of both n-step Sarsa and Tree Backup** (see diagram on page 155 for comparison diagram). At timestep t of the computation, **the proportion in which we use sampling is given by the discrete random variable $\sigma_t \in \{0, 1\}$. This variable can be any function of the timestep**.

If σ_t is 1 **then sampling is used, with the importance sampling ratio** ρ for that step. **For 0, we use the expectation**. This means that in the case in which $\sigma_t = 1$, the algorithm is equivalent to Sarsa with Importance sampling, and when $\sigma_t = 0$, it's equivalent to the Tree Backup algorithm (see page 156 for full algorithm).

Note: One difference is that, unlike n-step Sarsa, all intermediate steps are updated not just the first one, more similar to how Tree Backup works. The diagram on page 155 is a pretty important one!

Summary

In this chapter we saw how we can extend previous algorithms such as Q-learning and Sarsa to update using n-steps. **The two approaches presented use either sampling or expectation (or both in the case of n-step $Q(\sigma)$)**. By using multiple steps we ensure faster training in the case of sparse rewards.

---

(8) Planning and Learning with Tabular Methods

Similarly to how MC and TD methods can be unified through the n-step framework, in this chapter **we are trying to unify model-free and model-based methods**. Whereas model-free methods rely purely on learning, model-based ones rely (mostly) on **planning**.

Models and planning

Models can be either distribution models or sampling models. The former assumes full or learned knowledge of the MDP's dynamics $p(s', r|s, a)$. A sampling model, however, can only generate one sample from the MDP at a time (a simulation of the environment might fall into this category). On the other hand, DP, for example, assumes a perfect distribution model.

We can consider 2 types of planning: **state-space panning** which searches through the state space for an optimal policy and **plan-space planning which searches through the space of plans**. The latter needs some operators which transform the plans and a partial ordering relationship for them (eg. evolutionary methods) and are beyond the scope of this book.

A simple example of planning is **a planning Q-learning version, called Q-planning**. A simple version of this algorithm involves selecting a state action-pair randomly and giving it to the model. and then using the reward to train using standard Q-learning.

Note: As a rule of thumb, planning can and should be done in small incremental steps intermixed with learning. This is a key requirement for efficient learning.

Dyna-Q

With a planning agent, real data is used for both model learning and direct reinforcement learning (improve the value function like the other methods) (see page 162, for a concise diagram of the process). The model **can also improve the function through planning, this is called indirect reinforcement learning**.

Dyna-Q is a model-based algorithm which does direct RL through Q-learning and learns a deterministic tabular model (each (S, A) pair has a single (S', R) transition). This is alternated with an **indirect RL step which updates the Q-values, by randomly sampling the model n times** (if $n = 0$, it means that the agent is non-planning).

Dyna is shown empirically to have faster convergence on deterministic environments (see page 164 for algorithm, and page 165 for the learning curve diagram).

When the model is wrong

The model can be incorrect and, thus, the policy learned can be suboptimal. Reasons for the model being wrong:

- The environment is stochastic and it is hard to fully learn the MDP dynamics.
- The MDP is learned through function approximation and is, thus, imperfect.
- The environment is non-stationary.

In the Dyna-Q case, we can solve the problem of non-stationarity by rewarding exploration. We can do this by adding to the reward of the pair (S, A) a term $k\sqrt{\tau}$ where k is a small constant and, τ is the number of timesteps since the last visit of the state-action pair. We call this algorithm **Dyna-Q+** (see page 167 for comparison).

Prioritized sweeping

Prioritized sweeping is a method based on the observation that **using all state-action pairs from the model usually doesn't have a lot of states they can learn from** (this happens especially with sparse rewards). This suggests that planning **should be working backward from *goal-states*** (basically anything with a stronger reward signal).

To implement prioritized sweeping we **introduce all state-action pairs (S, A) whose update difference P is larger than a threshold θ with P as a priority into a queue**.

$$P = |R + \gamma \max_a Q(S', a) - Q(S, A)| \tag{35}$$

This can happen **both during the direct RL phase or during the indirect one**. During the indirect RL phase, for each action sampled from the queue, **we insert all the state-action pairs leading to it** if they satisfy the above condition (see page 170 for algorithm).

Note: Prioritized sweeping can be extended to stochastic MDPs by trying to learn the stochastic model as the *frequency* of next state-action pairs from each pair. We can also naturally use *expected updates* as a better approximator.

State-space planning can be viewed as a sequence of value updates based (or not) on the model which vary **in the type of update** (expected or sample, large or small) and **in the order in which they are done** (eg. prioritized or sequential).

Expected and Sample Updates

One-step updates can be classified in 3 dimensions. One is **whether they update the state or state-action value function**. The other one is **whether the update is wrt. an arbitrary given policy** π **or the optimal policy**. This gives rise to 4 functions whose values are updated: v_π, v_*, q_π, q_*.

For each one of these functions, **one can use either sample or expected updates**. This differentiation is, however, relevant only in the case of stochastic environments. Each combination of function and update type results in a specific update rule (eg, Sarsa, Q-learning, etc) (**see page 172 for a diagram**).

Each of them varies in utility in certain situations, **for example, sample update rules seem better when the branching factor is moderately large** (>100) as expected updates would require b (branching factor) computations to do an update while sample updates tend to asymptotically reach the same error reduction in fewer steps. Moreover, sample updates, appear to be less affected by estimation errors than expected updates (see diagram on page 174).

Trajectory sampling

Exhaustive sweeps, such as those performed in DP devote an equal amount of time to the entirety of the state-action space. Another approach **would be to sample according to a certain distribution, such as the on-policy distribution**. Depending on the distribution (ie. if the number of visits of each step does not go to infinity in the limit), the convergence guarantee may be broken. However, it may be fine in practice.

Sampling based on the on-policy distribution is called trajectory sampling and can be advantageous in terms of speed in MDPs with many states and small branching factor, but in the long run, it may hurt exploration and provide diminishing improvements (see page 176 for experiment and diagram). The diminishing results are usually because more time is spent on states whose values are already learned well enough.

Real-time Dynamic Programming (RTDP)

RTDP is a variant of **asynchronous DP where the values visited are distributed according to the on-policy distribution**. Actions are selected greedily and expected updates are applied to the current state. Any set of **arbitrary states** can be updated as well at each step (eg. limited horizon look-ahead, etc). By using expected updates, RTDP is a form of **value iteration**.

If the episode ends in the goal state, RTDP will converge to the optimal policy **over all provided relevant states** given a few simple constraints, such as all the initial values being zero.

Note: This is a very brief presentation of the subject. See chapter 8.7 on page 177 for more in-depth info.

Planning at decision time

The type of planning done until now to **improve the estimation of value functions is called *background planning***. Another type of planning, which **takes into account the current state S_t and its goal is to produce a better A_t is called *decision-time planning***.

The two types of planning are not mutually exclusive and decision-time planning methods can store their results by updating the value functions too, but **the primary goal of decision time planning is to improve the quality of actions taken at each control step**. What this type of planning usually provides, is deeper and better look-ahead than using the raw value function.

The two types can be combined, and often are, but decision time planning is often used in tasks where inference time is not a concern as the look-ahead incurs some overhead.

Heuristic search

Heuristic search is a popular decision-time planning method frequently used in classical AI which uses **tree expansion from the current state to look ahead and search for the best action to take**. This can be integrated into RL **through value function improvement**. What this means is the fact that through look-ahead we can improve the estimate of the value function, and thus improve the policy. Unlike in traditional heuristic search, however, we can use this traversal to also do updates on the value function.

Heuristic search methods can be seen as a **generalization of single-step greedy** action selection. If look-ahead has a significant enough impact (ie. either γ^k - total discount - is very small or the episode end is reached) we can learn perfect optimal value functions at the cost of computation. This happens because **if we fully traverse the tree, we know what the *true return* actually is**. One-step or multi-step updates can be used to actively improve the value functions during this search step.

For tree expansion, we can use a random traversal, but we are much more likely to get a better estimate if we use an on-policy distribution (see figure 8.9 on page 189 for how this may be implemented with one-step updates).

Rollout algorithms

Rollout algorithms **are decision-time planning algorithms based on MC-simulated estimations starting from the current state ab using a fixed policy π called *rollout policy***. The average MC estimates are not used towards learning an optimal policy, but for choosing the action for the current state.

The role of this is to improve the estimate $q_\pi(s, a)$ based on these MC averages. This way, according to the *policy improvement theorem*, we know that the actions taken greedily wrt. the better estimates of the value function are **at least as good** as those of the policy.

This can be viewed **as a single step of asynchronous value iteration** (async because we do not sweep the entire state-action space) starting from the current state S_t **for which we do not memorize the results**.

Note: Rollout algorithms *do not perform any learning*, instead, they refine already existing policies with MC estimates to improve control.

Monte Carlo Tree Search (MCTS)

MCTS is a rollout algorithm which uses a truncated lookahead tree to store estimates of future values. To implement an MCTS algorithm we require these 4 steps:

- **A tree policy is used to traverse this memorized expanded tree of lookahead states (ie. the selection step) until a leaf node is reached**. Selecting which node to traverse to can be done using whatever method we want (eg. Upper Confidence Bound (UCB), ε-greedy, etc).
- Depending on the implementation, **when reaching a leaf node, we can expand it and memorize its children in the search tree as well (ie. the expansion step)**. When exactly we opt for expansion depends on the algorithm. Some algorithms expand when a certain threshold of visits is reached, while others decide based on the estimated value of the leaf.
- **We can also perform a simulation step when reaching a leaf instead of an expansion one**, This involves simulating a complete episode to get its return **using a simple and fast rollout policy**. This is very similar to what classical MC rollout algorithms do.
- The **resulting value is then used to update the estimates of all nodes on the root-to-leaf path**. This step is called the **backup step** (see page 186 for figure).

These 4 steps are repeated until a number of iterations is reached, or some other constraint is satisfied. The best action is then chosen wrt. the values accumulated in the tree (or to the number of visits). MCTS is ran for every new state we visit, being a decision-time planning method. **The learned tree can be discarded or we partially kept for new values**.

MCTS benefits from both sample-based evaluation of policies and policy improvement (RL) for the tree. Moreover, it focuses search only on promising states according to MC simulations.

Summary of planning methods

Planning methods **require a model of the environment which may be either a distribution or a sample model**. The former can be used for expected updates but it is harder to learn than the latter. These models can be priorly known (eg. having some knowledge about the environment like in the game of Go) or learned through experience.

Simulated experience generated by the model can be used to update the value functions used for our policies. **Learning and planning can be seen as the same algorithm applied to 2 different sources of experience.**

Another aspect through which planning algorithms differ from one another is through how they prioritize the updates. Algorithms such as RTDP, for example, update values wrt. an on-policy distribution while Dyna-Q may sample uniformly from the model.

Planning can also be done to augment decision making for just the current step rather than being used for learning - **these are called decision-time planning methods**. Rollout algorithms such as MCTS are examples of this, where through clever MC simulations we improve upon a weaker policy.

Summary of part I - Dimensions

All RL methods presented (and all RL methods in general) have in common the fact that they **estimate value functions, they update the value functions using backups and they follow the framework of Generalized Policy Iteration (GPI)**.

RL algorithms do, however, have some conceptual differences. **These differences can be seen as a spectrum across a few dimensions**, out of which we mention the 2 most important ones:

1. **Width of updates**. At the extremes of this dimension lie sample updates at one end and expected updates at the other one.
2. **Depth of update**. Methods vary from using a single-step update (like TD(0) methods do) to using the entire length of the episode as in MC methods.

The classical methods lie at the extremes of this spectrum:

- MC has infinite depth and uses sample updates
- TD(0) has depth one and uses a sample update
- Dynamic Programming has depth 1, but uses expected updates
- Exhaustive search occupies the remaining corner, using full episodes and expected updates

However, many methods, such as *n-step TD* and *heuristic searches*, lie somewhere in the middle of the spectrum (**see page 190 for one of the most informative diagrams in the book**).

Another dimension which we didn't include is whether the method is on- or off- policy. This is orthogonal to the other 2 and generally (if not always) binary (there are no methods which are *kinda* off-policy).

Other dimensions can be added to this taxonomy. Among these are whether the method operates on continuing or episodic tasks, whether it is asynchronous, what value function it approximates, and so on. **These are not exhaustive and not mutually exclusive**

The remaining, and **one of the most important dimensions is whether the method is tabular or uses function approximation**. All methods presented in part I were tabular; part II will cover function approximation.

Part II - Approximate solution methods

Integrating function approximation methods with RL boils down to using function approximation methods such as those used in **supervised learning** to approximate value functions. This is done so we can apply RL methods to MDPs with an **infinite number of states** where we **don't have any guarantee of convergence** to an optimal policy even in the limit of infinite data and time.

However, function approximation has its own issues arising with things such as dealing with non-stationarity, bootstrapping, and delayed targets.

---

(9) On-policy prediction with approximation

We consider the case of approximating the **state-value** function using a parameterized model $\hat{v}(s, w) \approx v_\pi(s)$ with the parameters $w \in \mathbb{R}^d$. This model can be any **parametric model** from a linear model to an ANN or a decision tree. Typically $d << |S|$ to prevent memorization. This leads to different dynamics than in the case of tabular methods. One example is that approximate methods tend to work better **on partially-observable environments**.

Value-function approximation

The target of updates in RL is to move the value function for some particular states towards some desired values. We'll use the notation $s \mapsto u$ to indicate that the value of the **value function for state s should change in the direction of value u.** This way MC updates, for example, can be written as $S_t \mapsto G_t$ and TD(0) ones as $S_t \mapsto R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t)$.

Each update can be seen as a single training example in a *supervised learning* scenario. This means that we should use learning methods which can deal with the non-stationarity that comes from the fact that the policy π is continually changing and so is its value function as well.

The prediction objective (VE)

Unlike in tabular methods where the updates of a few states' values did not affect others, function approximators do not have the same guarantee. An update for the value of one state will influence the approximation of potentially all the other states.

Therefore, an **unbiased optimization objective is impossible to write**. We instead define a distribution $\mu : s \mapsto [0, 1], \sum_{s \in S} \mu(s) = 1$ by which we weight the MSE of our approximations and obtain what we call the **Mean Squared Value Error**:

$$\overline{VE}(w) \triangleq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2 \tag{36}$$

We can define $\mu(s)$ as the proportion of time spent by the policy in state s. We call this the **on-policy distribution**. If we define $h(s)$ and $\eta(s)$ as, respectively, the probability of starting in state s and the average number of timesteps spent in state s per episode. We then get the following linear system:

$$\eta(s) = h(s) + \sum_{\bar{s} \in S} \eta(\bar{s}) \sum_{a \in A(\bar{s})} \pi(a|\bar{s}) p(s|\bar{s}, a), \forall s \in S \tag{37}$$

We can then derive the value of μ as the average of average time spent per episode in state s amongst all averages.

$$\mu(s) = \frac{\eta(s)}{\sum_{s' \in S} \eta(s')} \tag{38}$$

It is not yet clear that $\overline{VE}$ is the best objective to optimize to get a better policy. The optimization task is also a difficult one for non-linear models and most of the time we are satisfied with getting a **local minimum**.

Stochastic-gradient and semi-gradient methods

To optimize $\overline{VE}$, we apply the classical SGD algorithm on our approximating function f. Therefore we get the following update rule for the weights w_t:

$$\begin{aligned} w_{t+1} \leftarrow & w_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, w_t)\right] & (39) \\ = & w_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, w_t)\right] \nabla \hat{v}(S_t, w_t) & (40) \end{aligned}$$

Here $\nabla \hat{v}(S_t, w_t)$ stands for the gradient of $\hat{v}$ wrt. the components of w. Although ideally, we would want to perform the update in the direction of $S_t \mapsto v_\pi(S_t)$ we do not have the real value and instead, we will use a **noisy estimate** we call U_t. If U_t is unbiased, it will be equal to $v_\pi(S_t)$ in expectation. If we only use observed examples for updating then we guarantee that **we are using the on-policy distribution.**

This is basically the **vanilla SGD** algorithm if this estimate is unbiased and independent of w_t. Therefore, one solution would be to use MC estimates (which are unbiased) as the targets of the update and the convergence is guaranteed (see page 202).

By bootstrapping, however, we do not have the same guarantee. For example, using the TD(0) target $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t)$, our estimate depends on the current values of w_t and a true gradient method would have to recursively compute the gradient. We do not do this, and instead, use the gradient for just the current values. This is called a **half-gradient** method and although it has weaker convergence guarantees, it can be used for **online and gradual** learning.

A SGD variant used is **state aggregation** where multiple states S_t are grouped and a single component of the weight vector w_k is updated (its gradient is considered 1, while the others' are 0). This basically simplifies the problem **through discretization**.

Linear methods

One important case of function approximation is that of the **linear model**. In this case, we define a feature function $x : S \mapsto \mathbb{R}^d$, $x(s) \triangleq \left[(x_i(s))_{i=1}^d\right]^T$ where $x_i : S \mapsto \mathbb{R}$ are feature functions. The function approximation is thus a linear transformation of these *features*.

$$\hat{v}(s, w) \triangleq w^T x(s) = \sum_{i=1}^{d} w_i x_i(s) \tag{41}$$

The gradient of the value function wrt. w is $\nabla\hat{v}(s, w) = x(s)$ and the update rule for the weights becomes:

$$w_{t+1} \leftarrow W_t + \alpha\left[U_t - \hat{v}(s_t, w_t)\right] x(S_t) \tag{42}$$

For this model, **the MC algorithm has a convergence guarantee to the global optimum**. The **semi-gradient TD(0)** (where U_t is the bootstrapped approximation), too, has a guarantee of convergence for the linear model, not to the global minimum, but rather to a point near it. It can be shown that the semi-gradient method converges to a point called the **TD fixed point** which is within a bounded expansion of the global minimum:

$$\overline{VE}(w_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(w) \tag{43}$$

This result, as other convergence guarantees, is based on *weighting according to the on-policy distribution*.

Note: Because the discounting factor γ is usually close to 1, it means that the bound is usually large. Also, the semi-gradient update rule can be extended to an n-step update easily.

Feature Construction for Linear Models

Non-linear transformations and dependencies between features are not captured by linear models. We can use different basis functions to capture these.

One example of candidate features are **polynomial** features, where every component of the x vector, given a state $s \in \mathbb{R}^k$, where $c_{i,j} \in \{i\}_0^n$, has the form:

$$x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}} \tag{44}$$

Here n stands for the highest degree polynomial for the features. The result is that each feature is a polynomial of the the components of the state vector.

One can also use a **Fourier** basis for the features, which in practice seems to have better performance than the polynomials, but is not very good at dealing with discontinuities.

If we consider our state's components belonging to the interval $[0, 1]$ and given the vectors $c^i = (c_1^i, \dots, c_k^i)$ which define the frequency over each component, we get the features:

$$x_i(s) = cos(\pi s^T c^i) \tag{45}$$

Another possibility is the use of **coarse coding** where each state is mapped to a binary vector. Each component of this vector corresponds to whether the states is within a **receptive field**. A receptive field can be any contiguous set of points in the state space, but, usually, they are **repeating and overlapping tiles or circles** (see page 215).

Tiling is the simpler and overall more feasible alternative. State aggregation is a particular case of coarse coding where a single such tiling is used. Also, asymmetrical offsets for the tiles are often preferred to uniform ones (see Fig 9.11 on page 219).

Hashing can be used to reduce a high number of tiles to a manageable number. Hashing is, however, non-injective and assigns multiple tiles to the same bin, effectively creating larger, and even non-contiguous tiles.

Another important type of coding are **Radial basis functions (RBFs)**, which calculate a Gaussian-weighted distance from a set of centers c_i with variances σ_i.

$$x_i(s) \triangleq exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right) \tag{46}$$

These are differentiable and, thus, learnable. The centers and variances can be learned or not, but regardless, both are part of a class of models named **RBF networks**.

Nonlinear function approximation with ANNs

Artificial Neural Networks (ANNs) can be used together with reinforcement learning methods in several ways. For example, they can be trained to approximate a value function **given the TD error**, maximize expected reward like a **gradient bandit**, or through **policy-gradient** methods.

All of the methods of improving training and generalization in ANNs can be applied to ANNs in RL as well.

Least-Squares TD

When using linear functions, we used an iterative gradient-based method to get to the **TD fixed point** which is close to the real minimum $w_{TD} = A^{-1}b$. However, since A and b are the expectations of $x_t(x_t - \gamma x_{t+1})^T$ and $R_{t+1}x_t$, we can compute then as an average for all x and R values. We can use these estimates $\widehat{A_t}$ and $\widehat{b_t}$ (see page 228) to compute the closed-form value of w_{TD}.

$$w_t \triangleq \widehat{A_t}^{-1}\widehat{b_t} \tag{47}$$

This is called **Least-Squares TD (LSTD)** because of its use of the least-squares method. Although it uses the closed-form and therefore a fixed number of computations, this number can be large mainly due to the inverse.

Another small advantage is the lack of a step-size hyperparameter. However, this means that it cannot be applied online and it is not practical for non-stationary environments.

Memory-based function approximation

These are a form of **non-parametric** models which do **lazy learning.** They store seen values and only when a new estimate is required, they generate it based on the memorized values.

Kernel-based function approximation

Kernel-based methods are methods based on a **generalization strength** metric, $k : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}$ between some reference state and a query state. One simple way of approximation is using a set $\mathcal{D}$ of memorized states and a function $g : \mathcal{S} \mapsto \mathbb{R}$ mapping to their memorized values. Then we can use the average weighted by this metric as an approximation for the value of state s. This is called **kernel regression**. One of the most commonly used functions for kernels is the RBF.

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in d} k(s, s') g(s') \tag{48}$$

Any linear parametric regression can be used as a kernel regression if k is defined as the inner product of the feature vectors. However, not all kernel functions can be written like this. This is the basis of the so-called **kernel trick**.

$$k(s, s') = x(s)^T x(s') \tag{49}$$

Looking Deeper at On-policy Learning: Interest and Emphasis

Using the on-policy distribution has strong theoretical results for semi-gradients methods. We can, however weigh the distribution according to a random variable. In this case we will use the variable I_t which we call **interest**. The value of this variable is non-negative and will typically be less or equal to 1.

We also define a discounted version of interest called **emphasis**.

$$M_t \triangleq I_t \gamma^n M_{t-n} \tag{50}$$

The formula above is for the n-step emphasis. Correspondingly, we also have an emphasis-weighted n-step update rule:

$$w_{t+n} \leftarrow w_{t+n-1} + \alpha M_T \left[G_{t:t+n} - \hat{v}(S_t, w_{t+n-1}) \right] \nabla \hat{v}(S_t, w_{t+n-1}) \tag{51}$$

For the MC case, we can let $G_{t:t+n} = G_t$ and only update at the end of the episode.

The benefit of using interest-based methods is that we can arbitrarily control the impact of every step on the learning.

Summary

This chapter explored on-policy function approximation reinforcement learning. For the parametric model case of approximation, we defined the mean squared error value for those parameters $\overline{VE}(w)$ as the difference between the approximated value function v_{π_w} and the real values **weighted by their on-policy distribution** μ.

We studied the **linear** parametric model case mostly as the $\overline{VE}$ has strong convergence guarantees to the optimum when using MC approximations. The linear models also converge to a point close to the optimum called the **TD fixed point** when using **semi-gradient TD**.

Different functions can be used as basis functions for the features of the linear model. They range from polynomials to Fourier basis function and coarse coding. Coarse coding, specifically **tile coding**, discretizes the feature space to reduce computational complexity. RBFs can be also used as a form of continuous coding.

Using gradient methods with non-linear models such as ANNs has become popular in recent years under the name of **deep RL**.

---

(10) On-policy control with approximation

For on-policy control, we study **function approximation Sarsa**. The extension to this case is simple for episodic tasks but has to be reevaluated in the case of continuing ones. In the process we will also start using **state-action values** instead of state values.

Episodic semi-gradient control

Here we discuss a semi-gradient version of Sarsa. To do this, we first define a semi-gradient TD updater rule for state-action values.

$$w_{t+1} \leftarrow w_t + \alpha \left[U_t - \hat{q}(S_t, A_t, w_t)\right] \nabla \hat{q}(S_t, A_t, w_t) \tag{52}$$

If we replace U_t with the discounted return of one-step Sarsa we get the following update rule:

$$w_{t+1} \leftarrow w_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_t, A_t, w_t) - \hat{q}(S_t, A_t, w_t)\right] \nabla \hat{q}(S_t, A_t, w_t) \tag{53}$$

This being Sarsa, to guarantee convergence we have to only use **soft policies** such as ε**-greedy**. **Optimistic initial estimates** can also be used for exploration (for full algorithm see page 244).

Semi-gradient n-step Sarsa

Just by writing the n-step return in terms of the new $\hat{q}$ function, we obtain the semi-gradient version of n-step Sarsa. For the full algorithm, see page 247.

Similarly we can also obtain a function-approximation version **of expected Sarsa**.

Average reward: A new problem setting for continuing tasks

Instead of discounting the rewards to get the return, we can instead use the average reward as the optimization objective of our methods. We do this because the discounting setting is difficult to use with function approximation **in the continuing setting** and average reward is better suited.

For a given policy, we can compute the average expected reward in the limit to infinity steps:

$$r(\pi) \triangleq \lim_{h \to \infty} \frac{1}{h} \mathbb{E}\left[R_t | S_0, A_{0:t-1} \sim \pi\right] = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \tag{54}$$

Here $\mu_\pi(s)$ stands for the **steady-state distribution** and is equal to $\mu_\pi(s) = Pr\{S_t = s | A_{0:t-1} \sim \pi\}$. This distribution has the property that sampling states from it and sampling actions according to π, we remain under the same distribution.

$$\sum_s \mu_\pi(s) \sum_{s',r} \pi(a|s) p(s', r|s, a) = \mu_\pi(s') \tag{55}$$

For this distribution to exist, we must assume an **ergodic** MDP. Ergodic means that the probability of getting to a state, in the long run, is not conditioned by the starting state. We can then define a **differential** version of our returns so that the values are bounded:

$$G_t \triangleq \sum_{i=1}^{\infty} R_{t+i} - r(\pi) \tag{56}$$

Based on these *differential returns* we can define the differential versions of the Bellman equations and subsequently of the TD error as well. We do this by keeping an estimate of the average reward $r(t)$ with the variable $\bar{R}_t$.

$$\begin{aligned} \delta_t &\triangleq R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t) \\ \delta_t &\triangleq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \end{aligned} \tag{57}$$

Because these errors are given in terms of differential returns, they are bounded and we can use them with the **semi-gradient update rule**. In the long run, the TD error will thus converge to 0.

$$w_{t+1} \leftarrow w_t + \alpha \delta_t \nabla \hat{q}(s_t, A_t, w_t) \tag{58}$$

The main downside is that during training, we also need to **keep an average of the rewards**. A full algorithm is presented on page 251.

Deprecating the discounted setting

In the continuing formulation, under the assumption of **ergodicity of the MDP**, for the function approximation methods, the discount is redundant in the computation of returns.

Ergodicity, like in the case of differential returns, ensures that an average reward can be computed in the limit to infinity. Under this assumption, we can show that the average reward when using discounts **is proportional** to one without discounting as its value is $r(\pi)/(1-\gamma)$. This renders the discounting factor γ useless (a full proof is available on page 254).

The authors claim this stems from **the lack of a policy improvement theorem** for the function approximation case, where improvement in one state may not necessarily mean improvement of the policy.

Differential semi-gradient n-step Sarsa

Here we present the average reward formulation of the n-step Sarsa algorithm. The main difference is that at timestep $t+n$, **the return is defined in terms of differential** rewards instead of discounted ones.

$$G_{t:t+n} \triangleq (\sum_{i=1}^{n} R_{t+i} - \bar{R}_{t+n-1}) + \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) \tag{59}$$

Here $\bar{R}_{t+n-1}$ is the estimate of the average reward at timestep $t+n-1$. Thus, our TD error becomes:

$$\delta_t \triangleq G_{t:t+n} - \hat{q}(S_t, A_t, w) \tag{60}$$

Using this TD error, the algorithm is the same as normal Sarsa, with the additional step of estimating the average reward (see page 255).

Summary

In this chapter, we saw adaptations of the Sarsa algorithm to function approximation cases. Due to the use of semi-gradient methods, we had to use **average rewards $r(\pi)$ for the continuing case** to ensure they converged. This variant of Sarsa was called **differential Sarsa** due to its use of differential rewards.

---

(11) Off-policy methods with approximation

This chapter deals with the extension of off-policy methods to function approximation. The challenges are partially the same as those for tabular off-policy methods, but here, the on-policy distribution has a much greater role in **guaranteeing convergence of semi-gradient methods**.

Semi-gradient methods

Unlike tabular methods, function approximation methods **are not guaranteed stable**. Tabular methods correspond to a special case of function approximation.

One way to adapt off-policy methods to function approximation is to use **importance sampling** and one step TD(0).

$$w_{t+1} \leftarrow w_t + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, w_t) \tag{61}$$

In this equation, ρ_t is the importance sampling ratio and δ_t is the TD error, **either discounted or differential**. This way we can also generalize *expected Sarsa*. For example, the one step case of expected Sarsa looks like this:

$$\begin{aligned} &w_{t+1} \leftarrow w_t + \alpha \delta_t \nabla \hat{q}(s_t, A_t, w_t) \\ &\delta_t \triangleq R_{t+1} + \gamma \sum_a \pi(a|S_{T+1}) \hat{q}(s_{t+1}, a, w_t) - \hat{(}S_t, A_t, w_t) \text{ (episodic)} \\ &\delta_t \triangleq R_{t+1} - \bar{R}_t + \gamma \sum_a \pi(a|S_{T+1}) \hat{q}(s_{t+1}, a, w_t) - \hat{(}S_t, A_t, w_t) \text{ (continuing)} \end{aligned} \tag{62}$$

The problem of weighting the action-states is still **open to research** as the expectation can not be computed as clearly as in the tabular case. *N-step expected Sarsa* can be obtained through the same substitutions.

We can also adapt the *backup tree algorithm* to the function approximation case by putting it in the semi gradient framework. The TD error here is the same as the one mentioned before for episodic or continuing cases.

$$\begin{aligned} W_{t+n} &\leftarrow w_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, w_{t+n-1}) \right] \nabla \hat{q}(S_t, A_t, w_{t+n-1}) \\ G_{t:t+n} &\triangleq \hat{q}(S_t, A_t, w_{t-1}) + \sum_{k=t}^{t+n-1} \delta_k \prod_{i=t+1}^{k} \gamma \pi(A_t|S_i) \end{aligned} \tag{63}$$

Examples of off-policy divergence

Due to the interdependence of states in the approximation of value functions (ie. modifying the estimation for one state modifies it for all of them) **off-policy methods can diverge**.

As the example on page 260 shows, this happens in the off-policy case and not the on-policy one because, now, over-/under-estimations of the value functions may never be corrected. Because, when using off-policy methods, the actions are not taken or updated according to the on-policy distribution, the importance sampling ratio ρ_t **is often less than 1 and sometimes even 0 in**

the off-policy case. This leads to initial errors in the estimation of values for some states to **never be corrected**.

The paper presents on pages 261 and 263 two examples in which even using expected and/or synchronous updates leads to divergence. **Baird's example** even diverges for Q-learning. Q-learning does seem to always converge when given a policy which is close **to the target greedy one**, although there is no theoretical proof of this.

Some specific function approximation methods can be used which do not extrapolate to unobserved targets. This guarantee **does not apply** to popular ones such as ANNs and linear models with tile coding.

The deadly triad

The cause of instability and divergence in RL is not a single factor, but a combination of **function approximation**, **bootstrapping**, and **off-policy training**. These, as shown in the previous section, lead to divergence. We look at the pros of each of the causes individually.

- **Function approximation**, for example, is the only way to deal with arbitrarily large MDPs and states. This was not possible with tabular methods, and, thus, we can not give it up.
- **Bootstrapping** is not necessarily required, as we could use non-TD learning such as MC methods. However, MC has large memory requirements, and **more often than not TD learns much faster than MC** on tasks where states repeat (most tasks). To make TD more like MC, the stochasticity of updates can be diminished by using n-step TD, but sometimes it leads to decreased performance. Overall bootstrapping is greatly beneficial.
- **Off-policy learning** may seem the least indispensable one as good on-policy methods such as Sarsa do exist. However, off-policy learning **encourages exploration** more and **has strong behavioral motivations** as it allows parallel learning of multiple target policies using only one behavior one.

Linear value-function geometry

In this section, similar to the book, we describe a toy example of function approximation on the state space $\mathcal{S} = \{s_1, s_2, s_3\}$ and an approximate value function $v : \mathcal{S} \mapsto \mathbb{R}$, considering v_w as being the linear model with the weights w where $w = (w_1, w_2)^T$. Having only 2 components means that it can not map all states in $\mathcal{S}$.

Now, given a policy π we can define a value function v_π. This value function *may be* a 2-component linear one, but more often than not, it is not. If we consider v_w and v_π to also be their image $Im(\mathcal{S})$, we can **define a norm**:

$$||v||^2_\mu \triangleq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2 \tag{64}$$

This is basically the euclidean nor, weighted component-wise with the on-policy distribution $\mu : \mathcal{S} \mapsto [0, 1]$. Thus, we can **use it to write** the *Mean Squared Value Error*, for a linear model, for example: $\overline{VE}(w) = ||v_w - v_p i||^2_\mu$. This means we can also define **a projection operator** from an arbitrary v_π to the subspace of 2d linear models.

$$\Pi v \triangleq v_w \text{ where } w = \arg\min_{w \in \mathbb{R}^d} ||v - v_w||^2_\mu = \arg\min_{w \in \mathbb{R}^d} \overline{VE}(w) \tag{65}$$

Given the Bellman equation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{66}$$

There is only one value function for the policy π which solves it exactly. Therefore, when using another policy (ie. the one based on w) we will have a difference for each state called the **Bellman error at state s**:

$$\begin{aligned} \bar{\delta}_w(s) &\triangleq \left(\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]\right) - v_w(s) \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t) | S_t = s, A_t \sim \pi\right] \end{aligned} \tag{67}$$

We can see that this error is actually the **expectation of the TD error**. $\bar{\delta}_w$ is also the image of this error on the state space $\mathcal{S}$. Using this notation, we can then define its on-policy weighted mean:

$$\overline{BE}(w) = ||\bar{\delta}_w||^2_\mu \tag{68}$$

$\overline{BE}$ usually can not be minimized to zero, but for the linear approximation case there is value for which it is minimal and, in general, **it is different** from the minimal $\overline{VE}$ point Πv_π. To update iteratively the value vector using the Bellman equation, we have the **Bellman operator** defined by

$$(B_\pi v)(s) \triangleq \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{69}$$

Repeatedly applying this is **as if we did DP updates** the process converges to the true value function $v_\pi = B_\pi v_\pi$. With approximation, we can only do steps of Bellman updates followed by projection, similarly to DP.

We can also project the Bellman error giving rise to the **Mean Square Projected Bellman Error** $\overline{PBE}(w) = ||\Pi\bar{\delta}_w||^2_\mu$. Unlike for Bellman and MSE error, this can be minimized to 0 in the approximators' subspace and this point is the **TD fixed point** w_{TD},

A figure of all these points and operators can be found on page 267.

Gradient descent in the Bellman error

Until now, for TD, unlike for MC, we only had semi-gradient methods to work with the function approximation case. Because they are not **true gradient** methods, SGD does not offer its normal convergence guarantees.

One way would be to use the TD error directly

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \tag{70}$$

We can then define the *Mean Squared TD Error* which is the expectation of the squared TD error

$$\overline{TDE}(w) = \sum_{s \in \mathcal{S}} \mu(s) \mathbb{E}\left[\delta_t^2 | s_t = s, A_t \sim \pi\right] = \mathbb{E}_b\left[\rho_t \delta_t^2\right] \tag{71}$$

By minimizing this with SGD, we are applying what we call the **naive residual-gradient algorithm**. This converges robustly to its minimum, but as the example on page 271 shows, it is sometimes far from the true values.

As an alternative, we can try and **minimize the Bellman error** $\overline{BE}$. The point of zero Bellman error is, of course, the point of the real value function, so in the approximating case we should, at least, get close to it (see eq. on page 272).

One disadvantage of this minimization is that it is the product of 2 expectations so **we need to sample the environment twice from the same state** (usually not possible) and in the approximating case values **still diverge** (see page 273).

Note: For the deterministic environments, the TD error and Bellman error minimizations lead to the same results.

The Bellman error is not learnable

For the RL context, the authors consider that something is ***learnable*** if it can be learned *at all* from an infinite sample of experience data. It turns out that 2 different MDPs can generate identical data distributions, thus **unless we have access to their dynamics**, we cannot learn them.

This means that neither the $\overline{VE}$, nor the $\overline{BE}$ objective, is directly learnable (see pages 274-276). We can introduce an MC-based objective called the *mean square return error* which is learnable:

$$\overline{RE}(w) = \mathbb{E}\left[G_t - \hat{v}(S_t, w)\right] = \overline{VE}(W) + \mathbb{E}\left[(G_t - v_\pi(S_t))^2\right] \tag{72}$$

Because $\overline{RE}(w)$ and $\overline{VE}(w)$ differ only through a variance term, it means that their optimal parameter w^* is the same and, thus, **definitely learnable** for the former too.

However, for $\overline{BE}$ we can find counterexamples in which MDPs generating the same data distribution **actually have different optima** (page 276). This makes $\overline{BE}$ ill-suited as an optimization objective.

We could only use the residual-gradient algorithm with $\overline{BE}$ in situations where we **could double sample** for a state. Therefore, this **restricts it to model-based** methods where we have knowledge of the MDPs transitions.

Gradient-TD methods

This section considers the minimization of the $\overline{PBE}$ objective instead of the simple $\overline{BE}$ objective. The minimum of this error is, in fact, the w_{TD} point. It also seeks to find an algorithm which can do this in $O(d)$ time.

The full derivation can be found in the book from page 278, but the result is that we can decompose the gradient of the $\overline{PBE}$ into 3 terms.

$$\nabla\overline{PBE}(w) = 2\nabla\left(X^T D\bar{\delta}_w\right)^T\left(X^T DX\right)^{-1}\left(X^T D\bar{\delta}_w\right) \tag{73}$$

These 3 terms can be individually **written as expectations** under the μ distribution:

$$\nabla\overline{PBE}(w) = 2\mathbb{E}\left[\rho_t(\gamma x_{t+1} - x_t){x_t}^T\right]\mathbb{E}\left[x_t {x_t}^T\right]^{-1}\mathbb{E}\left[\rho_t\delta_t x_t\right] \tag{74}$$

Both the first and the last term of this product depend on the next sample x_{t+1}. We cannot sample both and multiply, so we should compute the expectations independently. We can estimate the product of the last two, and then **sample the first one**. For the former term, we then have the following formula and update rule:

$$\begin{aligned} v &\approx \mathbb{E}\left[x_t {x_t}^T\right]^{-1}\mathbb{E}\left[\rho_t\delta_t x_t\right] \\ v_{t+1} &\leftarrow v_t + \beta\rho_t(\delta_t - {v_t}^T x_t)x_t \end{aligned} \tag{75}$$

Here β serves as a secondary step size, used for learning v. Using this estimate, we can, then, update the weights. Depending on how we compute the weights, we get 2 algorithms, called **GTD2** and **TD(0) with gradient correction**, respectively.

$$\begin{gathered} w_{t+1} \leftarrow w_t + \alpha\rho_t(x_t - \gamma x_{t+1}){x_t}^T v_t \\ \text{or} \\ w_{t+1} \leftarrow w_t + \alpha\rho_t(\delta_t x_t - \gamma x_{t+1}{x_t}^T v_t) \end{gathered} \tag{76}$$

Both these algorithms involve 2 learning processes, one for v and one for w and, under certain decreasing step-size conditions, **can stably learn** the value of w. Many different variations of these 2 algorithms have been proposed.

Emphatic-TD methods

Emphatic-TD considers the discounting coefficient γ as the probability of continuing of an episode. Therefore $1 - \gamma$ becomes the probability that the episode will terminate and **restart from the state we transitioned to**. This is called **pseudo termination**.

An example algorithm can be seen on page 282. It is a variation of the interest and emphasis algorithm from chapter 9. Although stable in theory, **it has very high variance** and, therefore, rarely converges on Baird's counterexample.

Reducing variance

Due to the space of policies being very large, many behavior policies may be vastly different from the target one. This can lead to a **very high variance in the importance ratios** despite having bounded expectations.

Because of this, SGD tends to diverge unless small step-sizes are used. This can be mitigated through things such as: backup tree methods, having similar target and behavior policies, weighted importance sampling, etc.

Summary

This chapter summarized how off-policy learning cannot be easily translated to the framework of function approximation. It showed that **bootstrapping methods coupled with function approximation** are not stable in an off-policy setup, an effect called by the authors **the deadly triad**.

Using alternative objectives such as the **Bellman error** are appealing because they can allow stable SGD optimizations. This error, however, was shown to not be learnable, but an alternative **projected Bellman error** can be used at the cost of slightly higher computational complexity.

Off-policy learning in the domain of function approximation is still unsolved, but promises of better exploration and parallel learning of multiple policies make it appealing.

Note: In literature, the *Bellman operator* is more commonly denoted T^π and is called the *dynamic programming operator*.

---

(12) Eligibility traces

Eligibility traces are a method of **unifying TD methods and MC**. For example, TD(λ) refers to the use of an eligibility trace where we have TD(0) at one end ($\lambda = 0$) and MC at the other ($\lambda = 1$). N-step and MC methods employ what are called *forward views*. This mechanism can be, even entirely, achieved through eligibility traces.

The mechanism introduces a short term memory vector $z_t \in \mathbb{R}^d$ whose components mirror those of the weights. Each one is bumped when the component participates in prediction and slowly decays. Its value dictates how fast that component learns.

This mechanism **allows continual and uniform learning**, as opposed to the delayed one provided by n-step learning and MC.

The λ-return

We begin by reiterating the definition of the n-step return from timestep t with bootstrapping:

$$G_{t:t+n} \triangleq \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n \hat{v}(S_{t+n}, w_{t+n-1}), 0 \leq t \leq T - n \tag{77}$$

This is a valid update target for SGD as much as it is for DP. As a matter of fact, **any weighted average of returns** starting from t is a valid updated target which **still holds guarantees of convergence**. The TD(λ) algorithm is a particular form of averaging for a given $\lambda \in [0, 1]$. The formulations for the general case is the following:

$$\begin{aligned} G_t^\lambda &\triangleq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \\ G_t^\lambda &\triangleq (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \end{aligned} \tag{78}$$

This way we can see the equivalence with MC and TD(0) in the cases in which $\lambda = 1$ and $\lambda = 0$ respectively. Thus, we have a mechanism of **smoothly varying between** MC and TD(0) updates with the better results usually being achieved by intermediate values. This approach is called in theory a **forward view** one.

TD(λ)

By suing an eligibility trace vector $z_t \in \mathbb{R}^d$ we can effectively approximate the values of the λ-return in **an online manner**, at each timestep. The vector is initialized to 0, accumulates the gradient and fades away exponentially by $\gamma\lambda$ each timestep.

$$\begin{aligned} z_{-1} &\triangleq 0 \\ z_t &\triangleq \gamma\lambda z_{t-1} + \nabla \hat{v}(s_t, w_t) \end{aligned} \tag{79}$$

The TD error remains the same as before in the function approximation case and so does the update rule:

$$\begin{aligned} \delta_t &\triangleq R_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(S_t, W_t) \\ w_{t+1} &\leftarrow w_t + \alpha \delta_t z_t \end{aligned} \tag{80}$$

Combining these, if we have $\lambda = 0$ **we get exactly TD(0)** and in the $\lambda = 1, \gamma = 1$ the traces do not decay at all over time, amounting to an episodic, undiscounted MC task. However, unlike classical MC, **the updates are done incrementally and online**.

One caveat is that TD(λ) is much more sensitive to larger step-sizes than the offline λ-return algorithm.

Implementation Issues

Eligibility traces pose a complexity increase, **especially when using the tabular setting**, where the weight vector would be the size of the MDP. However, in the case of function approximation, the implementation usually incurs **just double the memory**. N-step versions of the λ-return methods also add a memory requirement and computational overhead.

Conclusions

Eligibility traces provide a way **to continuously vary** between MC and TD methods. They can be used in both on-policy and off-policy tasks and are usually **faster to learn** than TD(0).

Through **backward view online methods,** one can emulate the forward view λ-return methods efficiently. Their similarity to MC means that they also behave better in environments **where the Markovian property is violated** or very **sparse rewards**.

Good values of λ usually lie farther away from the MC-like end of $\lambda = 1$, as there, a sharp performance decrease can be noticed. There is **no clear theory on the selection of this hyperparameter**.

The main advantage is data efficiency at a higher computational cost. This may be useful in the case of online and especially real-world environments but might **break even when simulation and data gathering are inexpensive**.

---

(13) Policy gradient methods

Policy gradient methods directly take actions using a **differentiable policy** whose parameters are denoted $\theta \in \mathbb{R}^{d'}$. If we also learn a value function based on which we derive a policy, we have an **actor-critic** algorithm. The value function's parameters are usually denoted $w \in \mathbb{R}^d$.

Policy approximation and its advantages

The only condition to be able to do policy approximation is to have a policy function which is **differentiable wrt. its parameters**. In order to ensure exploration, policies are often **prevented from becoming deterministic**. One way to do this is to parametrize a preference function (ie. logits) and take the **softmax of that**:

$$\pi(a|s,\theta) \triangleq \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \tag{81}$$

Through these methods, we also have a simple way of dealing with **continuous action spaces**. In situations of partial observation (or imperfect function approximation), approximate policy methods can also **learn stochastic policies** (see page 323 for example).

Sometimes directly optimizing the policy **may be easier** than learning the value functions and facilitates the design of policy parametrization.

The policy gradient theorem

Another advantage of approximate policy methods is that action probabilities **change much smoother** than in the ε- greedy case. Because of this continuity, we have **stronger convergence guarantees**. We define the performance measure as the state-value of the initial, deterministic state s_0:

$$J(\theta) \triangleq v_{\pi_\theta}(s_0) \tag{82}$$

Here v_{π_θ} is the true value function of π_θ and we assume no discounting (ie. $\gamma = 1$). Initially, it seems like the performance depends on both action selection and the distribution of states, however, the **policy gradient theorem** shows that $\nabla J(\theta)$ does not need this distribution.

$$\nabla J(\theta) \propto \sum_S \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{83}$$

In the continuing case, the values are actually **equal**, and in the episodic one, the proportionality coefficient is equal to the **average length of an episode**. See page 325 for proof.

REINFORCE: Monte Carlo policy gradient

Using the policy gradient theorem, we can re-write the state value function **as an expectation** (or rather the value it's proportional to):

$$\nabla J(\theta) \propto \sum_S \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \tag{84}$$

We can then further modify the value so that we compute it wrt. to the return of an episode G_t:

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] = \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] \tag{85}$$

It is easy to see how this can be used **as an MC algorithm** with the following update rule (logarithm is used for compactness):

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} = \theta_t + \alpha G_t \nabla \ln \pi(A_t|S_t,\theta) \tag{86}$$

Because it is an MC method, it is well defined only for episodic tasks. An algorithm that takes discounting into consideration can be seen on page 328. Although it has strong convergence guarantees, being MC, **it has high variance** and is thus sensible to step size

---

REINFORCE with baseline

Reinforce can be generalized to include a baseline state value function:

$$\nabla J(\theta) \propto \sum_S \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla \pi(a|s,\theta) \tag{87}$$

This work because the subtracted value $\sum_a b(s) \nabla \pi(a|s,\theta)$ is actually 0. Therefore, for each update, we can subtract the baseline from the return ($G_t - b(S_t)$). This baseline can be, for example, an **approximator of the state value function** which we train using the same MC trials.

This has the same convergence guarantees, but with well-selected step sizes (α^θ and α^w) **we can greatly reduce the variance** and speed up learning of the policy. See page 330 for a full algorithm.

Actor-critic methods

Actor-critic methods involve a pretty straightforward modification. They simply replace the return **with a bootstrapped approximation** of it, the update rule for the **actor weights** θ becoming:

$$\theta_{t+1} \leftarrow \theta_t + \alpha\delta_t \nabla \ln \pi(A_t|S_t, \theta_t) \tag{88}$$
$$\delta_t \triangleq R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$$

Here, for simplicity, we use δ_t as the one step TD error. This can be replaced with an n step version. The **critic has its own set of weights** w which are updated independently. 1 step and eligibility trace versions of the algorithm are available on page 332.

Policy gradient for continuing problems

For continuing problems, we have previously defined performance in terms of average reward rate $r(\pi)$. Under the assumption of ergodicity, **the policy gradient theorem remains the same**.

$$\nabla J(\theta) = \nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s, a) \tag{89}$$

The algorithm arising can be seen on page 333 and only requires us to keep a running average reward. A proof of the theorem can be seen on page 334.

Policy parametrization for continuous actions

Parameterizing policies to account for continuous actions amount to simply changing the distribution they output. Instead of using a softmax distribution with parameterized preferences, we can instead **parameterize the mean and variance of a normal distribution**.

$$\pi(a|s, \theta) \triangleq \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \theta))^2}{2\sigma(s, \theta)^2}\right) \tag{90}$$

For both σ and μ we can use **whatever function** we want, under the constraint that the variance should always be positive.

Summary

This chapter presented an **alternative to action-value methods** by directly optimizing the policy wrt. the value of the initial reward. These methods are called **policy gradient methods**.

Unlike action-value methods, policy gradient ones, are guaranteed to converge to the actual value when trained on-policy. This is called the **policy gradient theorem**.

To get this gradient we can use MC methods and we get the REINFORCE algorithm. The variance of this method can be further reduced by adding a **baseline**, ie. a function that learns the values for states. If we use bootstrapping to compute this kind of value function, we can train using TD methods and get what are called **actor-critic methods**.

All these methods have stronger convergence guarantees than value-based methods and actively researched.

Case studies

Human-level Video Game Play

In Mnih et al. 2015, the authors use DQN to play various Atari games with very good performance with 50 million frames of information per game (38 days).

To do this, the authors tested several models and methods against each other and reached a set of improvements to DQN which led to the greatest performance increases:

First, to reduce memory, **4 frames** are fed to the model at once and actions are taken every 4 timesteps. This can also be seen as a way of *reducing partial observability* and making the process *more Markovian.*

Then as implemented by other papers before, the authors do not sequentially feed the experiences (S_t, A_t, R_t, S_{t+1}), but rather, store them in a buffer and select them randomly for training. This is called **experience replay** and was introduced by Lin 1994.

As expected from Q-learning, the value network is trained with semi-gradient methods, but bootstrapping is cached for C steps at a time to stabilize learning. This means that **a network snapshot is used for bootstrapping** and it is updated every C steps.

Architecture-wise, they tested MLPs, CNNs, and linear models. Out of the three, CNNs, unsurprisingly came out as winners with significant improvements over linear models, especially when paired with the other tweaks we mentioned.

Mastering the game of Go

The successes around the game of Go, namely the AlphaGo family of algorithms all revolve around **MCTS used as a policy improvement operator**. MCTS is a more refined version of a rollout algorithm.

Using MC seems to be necessary as there seems to be no search-based methods to efficiently evaluate states and state-actions for Go (all previous best-performing methods use MCTS). For example, with MCTS we can keep some subtree without fully discarding previous information like in generic MC.

AlphaGo's version of MCTS called Asynchronous Policy and Value MCTS (APV-MCTS), unlike traditional MCTS uses both the rollout policy and the value function to evaluate nodes in the tree. APV-MCTS also **keeps track of edge visits** and selects those with the most.

Acknowledgements

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014–2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

Copyright