An intro to

Reinforcement Learning

Daniel Heesch

Contents

1 Overview **4**
- 1.1 Policy . . . 4
- 1.2 Rewards . . . 4
- 1.3 Value functions . . . 5
  - 1.3.1 State-value function . . . 5
  - 1.3.2 Action-value function . . . 6

2 Dynamic programming **8**
- 2.1 Iterative policy evaluation . . . 8
- 2.2 Policy improvement . . . 8
- 2.3 Generalised Policy Iteration and Value Iteration . . . 9

3 Monte Carlo Methods **11**
- 3.1 Monte Carlo Policy Evaluation . . . 11
  - 3.1.1 The importance of state-action values . . . 11
  - 3.1.2 Tracking averages . . . 12
- 3.2 Monte Carlo Policy Improvement . . . 12
  - 3.2.1 Exploring starts . . . 12
  - 3.2.2 ϵ-greedy policies . . . 13
  - 3.2.3 Constant-α . . . 14
  - 3.2.4 Off-Policy Methods . . . 14

4 Temporal difference learning **16**
- 4.1 Policy Evaluation . . . 16
- 4.2 Sarsa: solving the control problem . . . 17
- 4.3 Expected Sarsa and Q-Learning . . . 18
- 4.4 Double Learning . . . 18

5 Unifying MC and TD methods **20**
- 5.1 n-step TD prediction . . . 20
- 5.2 Forward TD(λ) . . . 20
- 5.3 Backward TD(λ) using eligibility traces . . . 21
- 5.4 Combining Eligibility Traces with Sarsa . . . 22
- 5.5 Combining Eligibility Traces with Q-Learning . . . 23
- 5.6 Other kinds of traces . . . 23

6 Model-free vs model-based **25**
- 6.1 Prioritised sweeping . . . 25
- 6.2 Sampling vs Averaging . . . 26
- 6.3 Trajectory sampling . . . 27

6.4 Monte Carlo Tree Search 27

7 Reinforcement learning in continuous spaces 29
7.1 Overview 29
7.2 Discretising continuous spaces 29
7.2.1 Coarse coding and tile coding 29
7.2.2 Towards function approximation 30
7.3 Neural networks and reinforcement learning 31
7.3.1 Monte Carlo learning with function approximation 31
7.3.2 Temporal diffference learning with function approximation 32
7.3.3 Q learning with function approximation 32
7.4 Deep Q Network (DQN) 33
7.4.1 Experience replay 33
7.4.2 Fixed Q targets 33
7.5 Improvements on DQN 34
7.5.1 Double DQN 34
7.5.2 Prioritised Replay 34
7.5.3 Dueling Networks 34

8 Policy gradients 35
8.1 Policy performance 36
8.2 Maximising J 36
8.2.1 Score function gradient estimation 36
8.3 Introducing baselines 38
8.4 Actor-critic methods 39
8.5 Deterministic Policy Gradients 40

9 Conclusions 40

1 Overview

Reinforcement learning deals with decision problems in which we want to learn *sequences* of individual decisions or actions so as to achieve some long-term goal or maximise overall reward. Unlike in supervised learning, the value or correctness of an action is not explicitly given to the learner but needs to be inferred based on the overall reward or goal accomplishment. The learning problem therefore becomes more difficult but also more realistic as many real-world problems have a structure involving action sequences and delayed rewards.

1.1 Policy

The dynamics of many situations can be modelled reasonably well by a finite Markov decision process or MDP in which the next state depends on the past only through the current state and the chosen action. The goal of reinforcement learning then is to figure out the best action $a \in \mathcal{A}$ when the agent is in some state $s \in \mathcal{S}$. These rules are referred to as a policy. A policy could be stochastic returning probabilities or a density over actions:

$$\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$$

or deterministic returning the best action:

$$\pi : \mathcal{S} \mapsto \mathcal{A}$$

We use both kinds of policy notations writing $\pi_s(a)$ for a stochastic policy and $\pi(s)$ for a deterministic policy.

1.2 Rewards

A crucial part of the problem set-up is the rewards associated with states. It's the rewards that ultimately define the learning task. If the agent is an autonomously flying drone, it will depend on the reward function whether the drone learns to hover at some point, to land or to fly to some goal. The same task can be specified through different reward functions, and some are better than others. Changing reward functions to maximise learning efficiency is called reward reshaping, a fascinating topic all by itself, and beyond the scope of this little intro (see [7] for a much more thorough treatment).

Rewards may depend on just the current state, the current state and its preceding action, or even the preceding state-action pair and the current state. Al-

though in many scenarios, a simple mapping of states onto real-valued rewards is sufficient, we use the $R(s, a)$ notation to emphasise its possible dependencies on things other than the current state.

1.3 Value functions

Given a reward function R and a policy π, we can define the value of a state as the expected *cumulative* reward that would accrue if one were to follow that policy:

$$V_\pi(s) = \mathbb{E}\big[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots | s_0 = s\big]$$

Here $\gamma \leq 1$ is a discount factor that captures the intuition that more immediate rewards are worth more than those in the future. It is conceptually akin to the discounting of future cashflows in finance. For episodic tasks that are meant to end eventually in some goal state, a discount factor is important to encourage the agent to finish the task more quickly.

What is the expectation over? That depends on which aspects of the process are stochastic. Most commonly, the only source of stochasticity is the transitions between states, which are described as a probability distribution over successor states given a state-action pair:

$$P_{s,a}(s') = Pr\{S_{t+1} = s' | S_t = s, A_t = a\}.$$

This stochasticity is not due to the policy. It is part of the environment over which the agent has no control.

1.3.1 State-value function

We call $V_\pi(s)$ the state-value function of the policy. It can be written recursively in terms of the state values of successor states,

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V_\pi(s').$$

This is known as the Bellman equation. Given a policy, there is one unique state-value function satisfying this equation. Importantly, the Bellman equation says nothing about the optimal policy, that is a policy π^* such that

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

for all states s and all other policies π'. The solution to the Bellman equation is the state-value function corresponding to some, not necessarily the optimal, policy. The Bellman equation of course also applies to the optimal state-value function V^* and can then be written slightly differently:

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P_{s,a}(s') V^*(s') \right)$$

In order to maximise the value of the state s (the discounted cumulative reward), we simply choose the action that does so. The corresponding optimal policy is composed of just those actions a. This is the Bellman optimality equation for the state-value function.

1.3.2 Action-value function

V is a function only of the present state. It is computed by choosing *all* actions according to the given policy. Sometimes it is preferable to add the first action in as a parameter (with all subsequent actions being given by the policy), as follows

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P_{s,a}(s') V(s') \\ &= R(s,a) + \gamma \sum_{s' \in S} P_{s,a}(s') Q(s', \pi(s')) \end{aligned}$$

We call this function the *action-value function*. Q tells us more than V. It gives us the expected reward starting with *any* of the possible actions, not just the one produced under π. V is a special case of Q with the first action chosen according to the policy:

$$V_\pi(s) = Q_\pi(s, \pi(s)).$$

The optimal Q function we *define* as

$$Q^*(s,a) = \max_\pi Q_\pi(s,a)$$

Note that this is a point-wise maximum: for any s and a, the optimal Q value is the best achievable under any policy. The policy could be different for different s and a. It turns out that for MDPs, this is not the case and that there is one and the same policy π^* such that

$$Q^*(s,a) = Q_{\pi^*}(s,a)$$

for *all* s and a [7]. With these definitions, we can write the Bellman optimality equation for the action-value function as follows:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{s,a}(s') \max_a Q^*(s',a).$$

In both the action-value and state-value optimality equation, the policy is implicitly given by the optimal value function but more directly so in the action-value function: for any state s, the correct action under the optimal policy is the one that maximises $Q^*(s,a)$.

The solution methods for reinforcement learning problems are all closely related. In fact, dynamic programming, Monte Carlo and temporal difference methods can all be viewed as different instances of the same unified abstraction.
We will often use the terms prediction and control when describing algorithms. In this context *prediction* problem is that of estimating the correct value function under a given policy, meanwhile the *control* problem is that of finding the best policy.

2 Dynamic programming

If the agent knows the transition probabilities P and the reward structure, a dynamic programming approach as developed by Bellman in 1957 can be used to find an optimal policy. Note from the start that for interesting real-world problems, such information is almost never known, so in practice the methods below are rarely used. It's still instructive to briefly review them as some of the concepts will carry over to more complex problem settings.

Just like most other approaches we discuss later, the Dynamic Programming solution iteratively improves both the policy and the value function. This is done by alternating *policy evaluation* (estimating the value function given some policy) and *policy improvement* (updating the policy given some value function), until both have converged to their respective optimum.

2.1 Iterative policy evaluation

Given a deterministic policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, policy evaluation updates the value for each state by assigning it

$$V(s) \longleftarrow R(s, \pi(s)) + \gamma \sum_{s'} P_{s,\pi(s)}(s')V(s')$$

If we do this for all states, and repeat the process, $V(s)$ converges to the correct value function for that policy, *as long as* the value function is well defined (that is finite) for each state (this is guaranteed if either $\gamma < 1$ or the policy is such that the agent reaches the goal with probability 1 from any starting state). Convergence occurs irrespective of starting states, so these are typically set to 0.

To the extent that the policy is not yet optimal, the value function is not yet the best we can achieve. To get a better value function, we first need to take a step towards improving the policy.

2.2 Policy improvement

Given a state-value function (as computed during the preceding policy evaluation step) and its matching policy π, we choose a better policy by setting the action to the one that maximises the expected cumulative reward under the existing policy:

$$\pi(s) \longleftarrow \arg\max_a \left(R(s, a) + \gamma \sum_{s'} P_{s,a}(s')V(s')) \right)$$

The *policy improvement theorem* tells us that this process is sound. In particular, the theorem says that if π and π' are policies with

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

for all s, then the policy π' must be at least as good as π. The policy improvement theorem equally applies to stochastic policies.

2.3 Generalised Policy Iteration and Value Iteration

Combining the two steps of policy evaluation and policy improvement leads to a sequence of ever-improving policies. This method of policy iteration is costly when done exactly because every policy evaluation involves multiple sweeps across the entire state set. The idea of *generalised* policy iteration is to obtain an approximation of the value function under the current policy before updating the policy.

In the extreme case, we take each state in turn and perform policy evaluation (approximation) and improvement in one update. This is called *value iteration*:

$$V(s) \longleftarrow \max_a \left(R(s, a) + \gamma \sum_{s'} P_{s,a}(s') V(s')) \right).$$

Note that in value iteration, one no longer needs to keep track of the current policy. Upon convergence it's simply derived from the value function.

Algorithm 1 Value iteration

$\theta \leftarrow$ some small number
$V(s, a) \leftarrow 0$ for all s, a
repeat
    $\Delta \leftarrow 0$
    for all $s \in \mathcal{S}$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a R(s, a) + \gamma \sum_{s'} P_{s,a}(s') V(s')$
        $\Delta = \max(\Delta, |v - V(s)|)$
until $\Delta < 0$

The biggest issue with the above methods is the requirement that the transition probabilities be known. The value of a state is averaged over all the legitimate successor states that one *could* end up. If a policy is stochastic and draws from a distribution over actions, the average is over successor states *and* actions. This limitation is overcome by simulation-based methods such as Monte Carlo and

temporal difference methods.

3 Monte Carlo Methods

Monte Carlo (MC) methods update value functions by running instances of the Markov process, noting the reward and propagating it to the individual states that were visited along the way. Running the Markov process can be done without having explicit knowledge of the transition probabilities and the rewards. Think of a card game, *any* card game. It is much easier to simulate instances of such a game than to explicitly compute and work with all the possible transitions.

Just like the previous methods, Monte Carlo methods involve a succession of value function and policy updates. Value functions are updated based on episodes under some fixed policy, then the policy is updated based on the new value function and another episode is obtained under the the new, improved policy.

3.1 Monte Carlo Policy Evaluation

Given some policy, how do we use it to estimate value functions based on sampled episodes from the Markov process? Monte Carlo methods simply assign to a state (or state-action pair) the total reward from all episodes in which that state or state-action pair was visited. What if there are several such visits during an episode? In *every-visit MC*, the average is formed based on all visits, in *first-visit MC*, the episode only contributes once. In practice both do much the same thing. In both cases, and unlike in the Dynamic Programming approach, we typically do updates after having sampled an *entire* episode or several episodes and observed the total reward.

3.1.1 The importance of state-action values

If we knew the transition probabilities (*the model*), state values would be enough to derive a policy. Given some state, we choose the action that maximises the expected value at the next step (averaged over all the states that can be reached following that action). Not knowing the distribution of states an action may lead to, we can only derive a policy from state-action values. It is therefore state-action values that we need to learn during policy evaluation. Concretely, we are interested in

$$Q_\pi(s, a),$$

the value of choosing action a in state s, and following policy π thereafter. This value function is updated in much the same way as before the state-value

function. We simply record the average for state-action pairs rather than just states.

3.1.2 Tracking averages

Instead of only updating Q after observing rewards from many episodes, one would typically update it after *each* episode, such that it is always the average return up to that point. This not only gives us an immediate snapshot of the process of Q function evaluation, but also allows us to update the policy after each episode based on the current values of Q. The update can be done incrementally by changing an 'error' term to the current average:

$$Q(s,a) \longleftarrow Q(s,a) + \frac{1}{n}(G_n - Q(s,a))$$

where G_n is the reward from the nth episode.

3.2 Monte Carlo Policy Improvement

Given a new value of Q obtained under some policy (after one or several episodes), how do we use it to improve that policy? Recall that dynamic programming methods simply pick the best action for every state. The policy becomes deterministic with the result that some states may subsequently not be visited if the Markov process were to be run. This is not a problem for dynamic programming methods since it is not based on sampling transitions or entire episodes but loop through the entire state space and for each state exploit knowledge of the one-step dynamics. Monte Carlo relies on samples, and a deterministic policy may compromise its ability to explore all state-action pairs. What's to be done?

3.2.1 Exploring starts

When the policy to be evaluated is deterministic, some state-action pairs, may never be visited with the result that the corresponding Q values cannot be estimated. To ensure sufficient exploration, one can simply assign each state-action pair a non-zero probability of being chosen to start an episode.

Algorithm 2 Monte Carlo with exploring starts

initialise $Q(s, a)$ and $\pi(s)$, randomly
initialise $R(s, a)$ to hold returns
repeat
    Choose S_0 and A_0 such that each pair has a non-zero probability
    Generate an episode starting with S_0, A_0 according to π
    Note final return G
    for all unique s, a of episode **do**
        Append G to $R(s, a)$
        $Q(s, a) \leftarrow \text{mean}(R(s, a))$
    for all s of episode **do**
        $\pi(s) \leftarrow \arg\max_a Q(s, a)$
until we have seen enough episodes

3.2.2 ϵ-greedy policies

The idea of exploring starts is not terribly efficient. It provides good exploration of start states but the ensuing episodes are constrained through the use of a deterministic policy. Thus, it does not explore very well the space of possible episodes. Moreover, some start states may rarely be encountered in practice, yet even these would continue to be visited under this method.

A less wasteful and altogether more effective way to ensure exploration involves the use of stochastic ϵ-greedy policies. These have a strong preference for the optimal action but assign a non-zero probability to all other actions. In practice one often chooses the following distribution:

$$\pi_s(a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{for } a = \arg\max_x(Q(s, x)) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

Because all action-state pairs will eventually be visited no matter where the episode starts, one can limit the start states to those actually expected in the wild.

As we grow more confident about the value function and the resulting policy, the need for further exploration diminishes. The degree of exploration can of course be controlled via ϵ. By steadily decreasing ϵ, the Q function converges towards the true, optimal solution to the problem. The method is "greedy in the limit with infinite exploration", a property with its own acronym: GLIE.

3.2.3 Constant-α

Recall how we can update the value of Q after each episode such that it reflects the average total reward for a state-action pair up to that episode. Because of the $1/n$ term, more recent rewards contribute less and less to the Q function. This makes sense if we keep the policy the same, as the process then accurately estimates the Q function under that policy. If we *do* change the policy after each or several episodes, it make sense to give more recent rewards relatively more weight compared to earlier rewards. They are after all a better measure of a changing policy. The desired effect can easily be achieved by replacing the discount term $1/n$ by a constant:

$$Q(s,a) \longleftarrow Q(s,a) + \alpha[G_n - Q(s,a)]$$

By ensuring that a reward G_n now has the same effect on Q no matter how much has been seen before, we are effectively discounting the past.

Algorithm 3 contant-α GLIE Monte Carlo

initialise $Q(s,a)$ arbitrarily
$\alpha \leftarrow$ some small value $\in (0,1)$
repeat
    $\epsilon \leftarrow \frac{1}{i}$
    $i \leftarrow i+1$
    $\pi \leftarrow \epsilon$-greedy(Q,ϵ)
    Choose s_0 randomly, draw a_0 according to π
    Generate an episode $s_0, a_0, R_1, \ldots s_n$ according to π
    for all $t \in \{0, \ldots n-1\}$ **do**
        $G(s_t,a_t) \leftarrow \sum_{i=t+1}^{n} R_i$
        $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(G(s_t,a_t) - Q(s_t,a_t))$
until we have seen enough episodes
$\pi \leftarrow \epsilon$-greedy$(Q,0)$

3.2.4 Off-Policy Methods

One powerful idea that's used throughout reinforcement learning is that of using two policies at the same time: a target policy which we want to improve, and a behaviour policy which controls the agent's behaviour. Because the behaviour is not governed by the policy we want to learn, these methods are said to be *off-policy*.

Consider the problem of figuring out value functions with $\pi \neq \mu$ being two policies such that any action selected by π has a non-zero chance of being selected under μ. The total reward observed by following a trajectory under μ needs to

be assigned back to the sequence of state-action pairs, but in a way that reflects the fact that the trajectory might have been much less likely under π than under μ. The ratio of these probabilities turns out to be

$$w_t^T = \prod_{i=t}^{T-1} \frac{\pi(A_i|S_i)}{\mu(A_i|S_i)}$$

as the transition probabilities are the same for both trajectories and therefore cancel. In *ordinary importance sampling*, the value assigned to $V(s)$ (or $Q(s,a)$) is the w-weighted average over all the total rewards following a visit to s (or s,a). If G_i, $i = 1,\ldots,N$ are the samples (here the returns following a visit) and w_i the associated ratio, ordinary importance sampling produces the estimator as

$$E = \frac{\sum_{i=1}^N w_i G_i}{N}.$$

An estimator with less variance (but some bias) is produced by *weighted importance sampling* which produces a convex combination of individual returns thus,

$$E = \frac{\sum_{i=1}^N w_i G_i}{\sum_{i=1}^N w_i}$$

so the largest value it can achieve is $\max G_i$.

Instead of waiting until we have seen enough episodes and then computing the average return using one form of importance sampling, we can update the value function after each episode. For weighted importance sampling, instead of keeping track of all the individual returns, we only need to track the sum of the weights $W_t = \sum_{i=1}^t w_i$ and the last estimate E_t, and obtain the new estimate upon observing w_t and G_t, as follows:

$$\begin{aligned}
E_{t+1} &= \frac{\sum_1^t w_i G_i}{W_t} = \frac{\sum_1^{t-1} w_i G_i + w_t G_t}{W_t} \\
&= \frac{W_{t-1}}{W_t} \times \frac{\sum_1^{t-1} w_i G_i}{W_{t-1}} + \frac{w_t G_t}{W_t} \\
&= \frac{W_{t-1}}{W_t} E_t + \frac{w_t G_t}{W_t} \\
&= E_t - \frac{w_t}{W_t} E + \frac{w_t G_t}{W_t} \\
&= E_t + \frac{w_t}{W_t}(G_t - E_t), \quad t \geq 1 \qquad (1)
\end{aligned}$$

4 Temporal difference learning

Before we understand temporal difference methods, let's briefly reflect on the merits and limitations of Monte Carlo methods as presented in the previous section. They do not need a model of the environment: no knowledge of the reward structure and transition probabilities is required. That makes them more broadly applicable than dynamic programming methods. Unlike dynamic programming methods, however, Monte Carlo methods update the value functions not after every transition but at the end of an episode. How about problems that are not episodic, have episodes that may take a long time to finish, or provide knowledge about the reward structure during an episode that could be exploited? It is not immediately clear how Monte Carlo methods could be adjusted to cope with such more realistic settings. Could it be that temporal difference method can come to the rescue? You bet.

4.1 Policy Evaluation

Temporal difference (TD) methods differ from Dynamic Programming and Monte Carlo methods in the way value functions are evaluated. Monte Carlo methods wait until the total reward for the episode is known and either move the value closer to that value (after each episode) or sets it to the average (over many episodes). TD methods use the estimated value of the successor state and the known reward at that state to update the current value. Consider again the constant-α MC update rule from the previous section, applied to the state value function V:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big].$$

where G_t is the actual total return observed at the end of the episode. What if we use an *estimate* of that return, based on the actually observed reward at $t+1$ and the value function at that new state $V(s_{t+1})$ (as per Bellman's equation)? We update our estimate of $V(S_t)$ using an estimate of $V(S_{t+1})$ and some actually observed information (R_{t+1}) with the result that the estimates get better over time.

The simplest formulation of this idea is known as the TD(0) method with the following update rule:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big].$$

Instead of delaying the update until G becomes known, the TD method uses the current estimate from the next state which itself is still being updated. This

is in the spirit of DP methods with the difference that the latter average over all possible successor states, and therefore need to know the transition dynamics.

Just as we can update V, we can update Q:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big]$$

There is only one important difference: whereas s_{t+1} is immediately given by the action a_t, a_{t+1} is not. We can therefore only update Q once the next action has been chosen.

4.2 Sarsa: solving the control problem

Now that we know how to estimate Q given a policy, how do we improve the policy? We could sample transitions until Q has sufficiently converged, then update the policy and start sampling all over. This would work but is not very efficient. Instead we update the policy right away once our Q function has changed. In fact, we don't even maintain an explicit policy: our actions flow directly out of our current estimate of Q, and they immediately effect our new view of Q.

Because the transitions are sampled, just as in Monte Carlo methods, we do not have a guarantee that all state-action pairs are visited if we were to choose actions greedily under our current Q. A common fix is to use an ϵ-greedy policy with ϵ gradually decreasing over time. The corresponding algorithm is simple enough:

Algorithm 4 Sarsa: on-policy optimisation of Q

initialise $Q(s, a)$ somehow, with $Q(s, a) = 0$ if s is a terminal state
initialise $\epsilon \in [0, 1]$
repeat for each episode
    Choose S and A
    Choose A from policy derived from Q (e.g. ϵ-greedy)
    repeat for each step of episode
        Take A and observe R and S'
        Choose action A' from policy derived from Q (e.g. ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$
        $A \leftarrow A'$
    until episode ends
    reduce ϵ
until we have seen enough episodes

Because the policy being optimised is also the one governing the transitions, Sarsa is an example of an *on-policy* method.

4.3 Expected Sarsa and Q-Learning

Consider a situation in which one of the actions carries a substantial penalty. Under Sarsa, the action will still be taken with some non-zero probability. Whenever it does, it has a huge effect on the Q value of the preceding state-action pair. In order to reduce the impact of such actions, that is to say the variance in the errors, *Expected Sarsa* [14] computes the expected Q value over all possible actions, weighted according to the current policy, that is

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \sum_{a'} \pi(a'|S') Q(S',a') - Q(S,A) \right]$$

It effectively moves Q not by the sampled error but the expected error. The policy over which the expectation is computed can be the same ϵ-greedy policy used to draw the next action. But it could equally be a different one, for example the greedy policy that always selects the value-maximising action. In the latter case, the learning is said to be *off-policy*, and the expectation can be written as

$$\sum_{a'} \pi(a'|S') Q(S',a') = \max_{a'} Q(S',a')$$

Thus we obtain, as a special case of Expected Sarsa, MaxSarsa or Q-Learning:

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

4.4 Double Learning

The maximisation operation in TD methods introduces a bias towards larger values. The values eventually converge to the correct ones but during learning they tend to be overestimates. This bias was first noted and investigated in [12]. The proposed solution was dubbed *double learning*. Instead of using one and the same Q function to determine the maximum action *and* supply the associated value to update the Q value, we maintain (and learn) two Q functions. One supplies the maximising action, the other its current Q value, i.e.

$$Q_1(s,a) = Q_2(s, \arg\max_a Q_1(s,a))$$

and similarly for Q_2. At every step, we randomly decide which of the two to update. This simple modification of the original algorithm can make a big difference in terms of learning speed. The idea can be applied to other tabular

update methods as well as to function approximations [13].

5 Unifying MC and TD methods

Monte-Carlo methods and one-step TD methods can be viewed as sitting at opposite ends along a continuum, as special cases of a more general method. Whereas MC methods performs value updates towards the total return achieved at the end of an episode, TD methods update the value towards that of the next state or state-action pair. Clearly, it's all about the number of time steps that one looks ahead.

5.1 n-step TD prediction

We can easily modify our one-step TD prediction method and extend it to several time steps. The TD error for n-step TD prediction is

$$\Delta V_t(S_t) = \alpha\Big[G_t^{(n)} - V_t(S_t)\Big]$$

where

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + V(S_{t+n})$$

is the n-step return at time t. For one-step TD, the error reduces to $R_{t+1} + \gamma V(S_{t+1}) - V_t(S_t)$. The n-step formalism helps us conceptually but the idea is much easier to implement using what's known as eligibility traces.

5.2 Forward TD(λ)

What may look initially like an unneccesary complication turns out to simplify the handling of multiple time-steps. The idea is to move the current value towards a weighted average over different n-step returns (with different n) such that the weights decay exponentially with n. Concretely,

$$G_t^{\lambda}(s) = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

and the increment between current and new value is

$$\Delta V_t(S_t) = \alpha\Big[G_t^{\lambda}(S_t) - V_t(S_t)\Big].$$

As we vary λ continuously between 0 and 1, we move from 1-step TD methods to Monte Carlo methods. Consider first $\lambda = 0$. The total return becomes $G_t^{(1)}$, that is the 1-step return as in TD(0). For $\lambda = 1$, let's assume the episode reaches a terminal at time T, that is we only have $T - t - 1$ distinct returns in the sum, and everything thereafter is G_t, the total return. We can therefore split the sum

into

$$\begin{aligned} G_t^\lambda(s) &= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_t^{(n)} + (1-\lambda)G_t\sum_{n=T-t}^{\infty}\lambda^{n-1} \\ &= (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_t^{(n)} + \lambda^{T-t-1}G_t \end{aligned}$$

For $\lambda = 1$, the first term vanishes, and we are left with the total return G_t, which is precisely a Monte Carlo update.

5.3 Backward TD(λ) using eligibility traces

Instead of looking forward figuring out which states might be visited in the future, the implementation of TD(λ) takes a backward view. For each state we keep a variable $Z_t(s)$ that is incremented by 1 each time it is visited, and reduced by a constant factor every time it is not. We call this the *trace*. Concretely, the trace is updated as follows:

$$Z_t(s) = \begin{cases} \gamma\lambda Z_{t-1}(s) + 1 & \text{if } s = S_t \\ \gamma\lambda Z_{t-1}(s) & \text{otherwise} \end{cases}$$

When an error is computed at time t

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$$

this error can now be propagated backwards by updating previously visited states in accordance with their trace.

$$V_{t+1}(s) = V_t(s) + \alpha\delta_t Z_t(s)$$

for all states. The algorithm implementing TD(λ) is simple enough:

Algorithm 5 TD(λ)

initialise $V(s)$, somehow
repeat
    initialise $Z(s) =$ for all s
    initialise S to start a new episode
    repeat
        $A \leftarrow \pi(S)$
        take action A to get reward R and enter state S'
        $\delta \leftarrow R + \gamma V(S') - V(S)$
        $Z(S) \leftarrow \gamma\lambda Z(S) + 1$
        for all s **do**
            $V(s) \leftarrow V(s) + \alpha\delta Z(s)$
            $Z(s) \leftarrow \gamma\lambda Z(s)$
        $S \leftarrow S'$
    until episode ends
until we have seen enough episodes

$TD(1)$, that is with $\lambda = 1$, the algorithm implements a Monte Carlo update, but it is more general than the ones described earlier. $TD(1)$ is not limited to episodic tasks. Because it makes updates continuously during sampling, it is just as applicable to non-episodic tasks.

5.4 Combining Eligibility Traces with Sarsa

The previous section described the use of eligibility traces for evaluating a policy. How can it be used to optimise a policy? As before, we want to derive new policies based on our current state function $V(s)$ or state-action function $Q(s, a)$. To derive a policy from V, we would need the transition probabilities. If we don't have these, we need to evaluate and apply Q instead. This is done just as we did before with V:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta_t Z_t(s, a)$$

with

$$\delta_t = R_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})) - Q_t(s, a)$$

and

$$Z_t(s, a) = \begin{cases} \gamma\lambda Z_{t-1}(s, a) + 1 & \text{if } s = S_t, a = A_t \\ \gamma\lambda Z_{t-1}(s, a) & \text{otherwise} \end{cases}$$

As before, we would use some soft policy derived from Q to choose our actions and obtain our next error. Easy.

5.5 Combining Eligibility Traces with Q-Learning

Q learning is distinguished from other forms of finding an optimal policy by its separation of exploration and learning. The policy that governs the random walk across states can be highly exploratory. The state-action values are updated not by the values of subsequent state-action values but the best one (even though this may not subsequently be followed). This has one important consequence regarding the use of eligibility traces. When an exploratory move happens, any subsequent errors should not propagate back. Consider an extreme case in which the exploratory move leads the agent into a state with much lower state-action values than if it had taken the best move (according to the current policy). This would lead to a significant error causing the previous state-action value to be substantially decreased although much better options existed at that point. We can prevent such updates by setting all traces to zero whenever taking an exploratory action. In situations in which learning and thus exploration matters most, updates will be limited to only a few state-action pairs. As with plain Q-learning, the error that propagates is always computed based on the best action:

$$\delta_t = R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)$$

An alternative to zeroing out the traces is Peng's Q(λ) which does not distinguish between exploratory and greedy actions. It is more difficult to implement but has been found to perform significantly better. Given a non-greedy policy, the state-action values converge to a point between the optimal state-action values and the state-action values of that policy. If the non-greedy policy is made progressively more greedy, Q seems to be converging on the optimal values.

5.6 Other kinds of traces

Instead of accumulating 1's for every visit, an alternative assignment simply sets the trace value to 1. In this case we refer to the variables as replacing (as opposed to accumulating) traces and the corresponding learning and control methods as replace-trace methods. What are the benefits of replacing traces? Consider an agent taking the wrong turn several times from the same state until finally taking the right one and eventually hitting a terminal reward. With accumulating traces, the wrong state-action pair might end up with a higher trace value than the right state-action pair, despite being less recent (and wrong at that!). With replacing traces, the correct state-action pair is guaranteed to have a higher value. In such situations, replacing traces can speed up learning.

Even better than a replacement trace is what's been named Dutch trace. Accu-

mulating and replacement traces can be viewed as special instances of a Dutch trace, which is formally defined as

$$E_t(S_t) = (1 - \alpha)\gamma\lambda E_{t-1}(S_t) + 1$$

For $\alpha = 0$ we get an accumulating trace, for $\alpha = 1$ we get a replacement trace.

6 Model-free vs model-based

Methods for finding optimum solutions to the Bellman equation vary with respect to the amount of information that they require about the environment. Dynamic programming methods need to know the distribution of transition probabilities. Temporal difference methods only need the ability to sample episodes. When the environment is not simulated but real, we don't need a model at all. Otherwise, the common process of finding a policy is as follows:

$$\text{model} \longrightarrow \text{simulated experience} \longrightarrow \text{values} \longrightarrow \text{policy}$$

Even for those that start off not having a model, a model can be built whilst interacting with the environment. This model in turn can be used to generate simulated experiences to enhance the learning. This is the basic premise of Sutton's Dyna-Q algorithm [10] shown below:

Algorithm 6 Dyna-Q algorithm

initialise $Q(s, a)$ and Model(s, a)
repeat
  $S \leftarrow$ current state
  $A \leftarrow \epsilon$-greedy(S, Q)
  observe reward R and next state S'
  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
  Model$(S, A) \leftarrow R, S'$
  for $i = 1 : n$ **do**
    $S \leftarrow$ random previously visited state
    $A \leftarrow$ random previously taken action from S
    $R, S' \leftarrow$ Model(S, A)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
until forever, really.

A real interaction is followed by a number of simulated experiences using whatever knowledge one has gained up to that point. It is a form of consolidation, which may sometimes stand in the way of exploration, especially when the environment changes after the agent has found an optimal solution to the original problem.To alleviate this tendency, Dyna-Q^+ gives bonus rewards to state-action pairs which the agent has not experienced for a while, thus leading to biased sampling during simulations.

6.1 Prioritised sweeping

Instead of selecting simulated state-actions at random, it makes rather more sense to favour state-action pairs upstream of those that we know have changed.

The prioritised sweeping algorithm achieves this by maintaining a queue of state-action pairs that would change significantly when updated. The magnitude of the change is simply

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

A pair is added to the queue if P exceeds some threshold. Note that Q is not actually updated at this point of the algorithm but only when the pair is picked up from the queue (Step 12). Additional data structures would be needed to efficiently determine the state-action pairs that would lead to S (line 13).

Algorithm 7 Prioritised Sweeping

1: Initialize $Q(s, a)$, Model(s, a), for all s, a, and P-Queue to empty
2: **repeat**
3: $S \leftarrow$ current (nonterminal) state
4: $A \leftarrow$ policy (S, Q)
5: Execute action A; observe resultant reward, R, and state, S'
6: Model $(S, A) \leftarrow R, S'$
7: $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$
8: if $P > \theta$, then insert S, A into P-Queue with priority P
9: **for** $i = 1 : n$ **do**
10: $S, A \leftarrow$ first(P-Queue)
11: $R, S' \leftarrow$ Model(S, A)
12: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
13: **for** $\bar{S}, \bar{A}$ predicted to lead to S **do**
14: $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
15: $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A}|$
16: if $P > \theta$ insert $\bar{S}, \bar{A}$ into P-Queue with priority P
17: **until** the end of time

The prioritised sweeping algorithm is not easily extensible to continuous state-action spaces with an infinitude of possible pairs qualifying for inclusion in the queue. This is an active area of research.

6.2 Sampling vs Averaging

Dynamic programming methods average over all the actions possible from a given state, and the states that any one action may lead to. The new value estimate of the value function is as best as we can do. Its correctness depends only on the correctness of values of the successor states. Methods using samples such as Q-learning come up with estimates that are subject to sampling error but are computationally cheaper. If we have a model, we have the choice between these two types of updates. It turns out that in many cases, sampling may actually be preferable over averaging if there are lots of states and a high

branching factor.

6.3 Trajectory sampling

Instead of sampling all states uniformly during one sweep in model-based dynamic programming methods, one can sample according to a distribution that's informed by the current estimates of the value function.

6.4 Monte Carlo Tree Search

Monte Carlo Tree Search methods ([2], [3] and [4]) are behind the best performing implementations of GO (e.g. Deepmind's AlphaGo) and other games with large branching factor. Monte Carlo Tree Search grows a search tree in the direction of good moves. In each iteration, a simulation is run starting with actions within the existing search tree until a node is reached than can be expanded. The choice of actions is governed by the *tree policy*. The simulation then proceeds by repeatedly choosing actions according to some other *default policy* (most commonly choosing moves at random) until the game terminates.

The final reward is propagated back along all the nodes of the current search tree. Specifically, each node has one count for the number of times it's been visited, and another for the number of wins. In subsequent iterations, these counts are used to determine a potentially new trajectory through the now expanded search tree.

In summary, each iteration of updating action scores consist of the following four steps:

1. selection of nodes from root R to leaf L (of the current search tree)
2. expansion: choose some new node C, one step beyond L
3. play randomly from C till termination
4. update scores for nodes R to C

Once the loop completes, the best action can be read off at the root node of the search tree. The new state reached after following this action forms the new root node from which a new tree is grown. Note that in a simple implementation information from the previously built tree is simply discarded. More info about the basic algorithm and links to its many variations can be found at .

In order to ensure exploration, child nodes are typically selected based on a combination of the current number of wins w_i and the number of times they were

visited n_i. The most popular tree policy is the UCB method (Upper Confidence Bound) known from armed bandid problems and introduced to MCST in 2006 by Kocsis and Szepesvari. The child with the greatest

$$\frac{w_i}{n_i}\sqrt{\frac{\log t}{n_i}}$$

is chosen as the next node. Here t is the total number of simulations across the current node (the sum over all n_i of the child nodes). MCST with UCB is often referred to as the UCT algorithm. The UCT algorithm has the nice property that given enough iterations, the chosen action is the same as the optimal minimax solution (for zero-sum two-player deterministic games).

7 Reinforcement learning in continuous spaces

7.1 Overview

The algorithms of the previous sections assume that the number of states and actions is discrete, finite, and ideally small. The algorithms typically involve a loop over all states or actions (e.g. value iteration) and if they do not, there may be a max operation over action values that is linear in the discrete case but potentially very difficult in the continuous case (e.g. Q-learning). Without modifications to these algorithms, they can either not be applied at all, or become computationally very expensive.

There are two kinds of solutions to the problem: either we adapt the data, or the algorithm. The first usually involves discretising the action / state spaces, the second involves function approximations. We look at both approaches below with an emphasis on the latter.

7.2 Discretising continuous spaces

A simple trick belonging to the former category is to discretise the space. For example, if spatial coordinates were part of the state, then we could simply divide up the space into cells (in 2D) or cubes (in 3D). This division need not be uniform. It is often the case that certain areas of the space being modelled require a greater resolution than others, for example the areas around objects in the occupancy map of a robot. In any case, a finite number of cells allows us to treat the continuous state or action space as a discrete space of, say meta-states or meta-actions, and we can use tabular methods as before.

7.2.1 Coarse coding and tile coding

A general problem afflicting discretisation schemes that partition the space is the step change at the boundaries. Points close to each other may have vastly different values if they happen to fall onto opposite sides of a cell boundary. One simple way to alleviate this problem is to allow cells to overlap, such that the value of a point is a function of the values for each of the cells it belongs to, e.g. the average.

Tile coding is a special case of coarse coding. The space is overlaid with multiple tilings, each tiling being a partitioning of the space. Just as in coarse coding, each point can then be encoded as a binary vector $\in \{0,1\}^T$ with ones for the cells or tiles (out of T) that it *activates*. Each tile is associated with a value (or weight) and the value of a point is simply the sum of the values of the

activated tiles. Points across the boundary in one tiling are likely to be part of the same tile in another tiling, so the effect is that of smoothing the value function (although it is still discrete). One may ask how one is to choose the tilings. It is reasonable to expect that every problem has its own optimal tilings. Adaptive tile coding [16] takes the burden off you. The tiling is refined during learning by splitting existing tiles based on some heuristics (such as the value function not changing much over time).

7.2.2 Towards function approximation

We can view tile coding as a kind of function approximation. Because of the binary nature of the representation, the approximation is not smooth: the function is piece-wise constant as it has the same value for all inputs that map onto the same binary representation. We can make it 'more' smooth by increasing the number of cells that overlap at a point, but that means storing an ever-increasing number of cell values. More importantly, the approach suffers from a lack of generalisation: we cannot infer a point's value from that of another point, if the two points don't share at least one cell.

We can keep the notion of cells (or more generally a finite number of somehow privileged points in our state / action space) and still make the approximation function 'smoother than piece-wise constant': by allowing for *degrees of cell membership*. Instead of using a cell's value v_i only if the point falls inside, we can account for all the cells' values by weighting their contributions by some function $f_i(s)$ that falls off with their distance from our point of interest (such as a Gaussian centered on each cell centre). The result is a continuous function that, depending on the function f, may also be smooth, that is infinitely differentiable:

$$V(s) = \sum_i w_i x_i(s) = w^T x$$

We can view function x as a transformation of our input vector s into a higher-dimensional space in which we can then take the dot product with some weight vector w that is to be learned. In the feature space defined by x, the function approximation is linear, but it need not be in the original space of s.

To increase the representational capacity of the model further, we can pass the dot product through yet another function

$$V(s) = f(w^T x)$$

where x is some function of s, e.g. the state in our reinforcement learning

problem. This is the general functional form of the computation performed by artificial neural networks *at every node* with the activation function f typically being a non-linear function, such as a sigmoid or a piece-wise linear function. In multi-layer networks, the function's output becomes the input to nodes in subsequent layers.

The weights are learned by gradient descent: we first measure some sort of error or *loss* between the actual and the true value of V for different training examples x. The derivative of the loss function (at the current loss) with respect to each of the weights then gives us the direction in which to move weights so as to maximally decrease the loss.

7.3 Neural networks and reinforcement learning

In reinforcement learning problems, unlike in supervised learning situations, we do not know the correct output value to compute our loss. Recall that in all the algorithms discussed we iteratively refine our knowledge of the value functions starting with a rough approximation. We are effectively dealing with a moving target: as we learn the weights of our function approximation through iterative gradient descent, we also update our target function as we interact with the environment.

7.3.1 Monte Carlo learning with function approximation

Consider the Monte Carlo update step. At the end of an episode the realised return G_t is used to update the value function at each state S_t visited during the episode:

$$V(S_t) \longleftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Because of the stochastic nature of the Monte Carlo method, G_t varies between episodes. Still, G_t seems to be our best guess at the true value of V for a given state S_t. Let $\hat{V}(S, w)$ be our function approximation modelled by the neural network, then a suitable loss would be

$$L(w) = \frac{1}{2}(G_t - \hat{V}(S_t, w))^2$$

with gradients

$$\nabla L(w) = -(G_t - \hat{V}(S_t, w))\nabla\hat{V}(S_t, w)$$

and weight update

$$w \longleftarrow w + \alpha\Big(G_t - \hat{V}(S_t, w)\Big)\nabla\hat{V}(S_t, w).$$

We can use the same formalism to compute an approximate action value functions $\hat{Q}(S_t, A_t, w)$.

The above update takes care of policy evaluation. If we want to find the optimal policy, we improve the policy after every weight update by choosing an ϵ-greedy policy based on our updated value function, and repeat until convergence.

7.3.2 Temporal diffference learning with function approximation

Temporal difference methods update value functions based on an estimate of the final return, composed of the reward actually experienced at the next n states, plus the appropriately discounted value of the $(n+1)$th state. In the simplest variant with $n = 1$, TD(0),

$$V(S_t) \longleftarrow V(S_t) + \alpha\Big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big)$$

It seems reasonable to choose $R_{t+1} + \gamma V(S_{t+1})$ as our target to approximate by $\hat{V}(S_t, w)$ and thus the loss to minimise becomes

$$L(w) = \frac{1}{2}\Big(R_{t+1} + \gamma\hat{V}(S_{t+1}, w) - \hat{V}(S_t, w)\Big)^2$$

Carefully note that when computing the gradients, we treat w in $V(S_{t+1}, w)$ as a fixed parameter, that is we reduce the loss through the effect of w on $\hat{V}(S_t, w)$ only. Following the same steps as above for Monte Carlo, we arrive at the following weight update

$$w \longleftarrow w + \alpha\Big(R_{t+1} + \gamma\hat{V}(S_{t+1}, w) - \hat{V}(S_t, w)\Big)\nabla\hat{V}(S_t, w)$$

Since we need to sample actions based on their values, we want to approximate not the state value function but the action value function Q, but the principle is the same and the update equation is

$$w \longleftarrow w + \alpha\Big(R_{t+1} + \gamma\hat{Q}(S_{t+1}, A_{t+1}, w) - \hat{Q}(S_t, A_t, w)\Big)\nabla\hat{Q}(S_t, A_t, w)$$

7.3.3 Q learning with function approximation

Recall that Q learning is almost identical to TD(0) with the twist that we learn one policy but choose actions from another. The typical choice is to derive both

from the same underlying Q function with actions selected according to an ϵ-greedy policy, and the Q values being updated according to the greedy policy. Concretely, the target we want the network to approximate is

$$R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, w)$$

so the square error loss becomes

$$L(w) = \frac{1}{2}\Big(R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, w) - \hat{Q}(S_t, A_t, w)\Big)^2$$

With the gradients given by

$$\nabla L(w) = -\Big(R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, w) - \hat{Q}(S_t, A_t, w)\Big)\nabla \hat{Q}(S_t, A_t, w)$$

the weight update holds no surprise:

$$w \longleftarrow w + \alpha\Big(R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, w) - \hat{Q}(S_t, A_t, w)\Big)\nabla \hat{Q}(S_t, A_t, w)$$

7.4 Deep Q Network (DQN)

Take the old Q-learning algorithm, use a deep neural network to approximate your action-value function and throw in a few nifty ideas to make learning more stable, and you get Deepmind's DQN [6]. We'll take a look at two of those nifty ideas, namely *experience replay* and *fixed Q targets*.

7.4.1 Experience replay

Instead of using an observation (s_t, a_t, r_t, s_{t+1}) only once to update the action value function, it is added to a buffer and updates are performed on a randomly drawn *minibatch* of such observations. Correlations in the observation sequence, which are known to cause instability problems in the context of non-linear function approximators, are thereby largely removed.

7.4.2 Fixed Q targets

When approximating Q by a continuous function, the learned function parameters affect both the target and the predicted value. The two become correlated: updating parameters based on the loss changes both the predicted value (the desired effect) but also the target. To reduce the correlation, [6] propose to fix the parameters of the target for several minibatches. One therefore needs to maintain two sets of parameters, one that's being optimised for each minibatch, and the other that is used to compute the targets and which is only updated once in while.

7.5 Improvements on DQN

Since the original paper on DQN came out, a number of significant enhancements have been proposed. We'll briefly sketch three of them below. DQN and its variants all inherit a fundamental limitation from Q-learning, that is, it works best if the action space is discrete since the optimal policy requires a maximisation over the action-value function. We later look at (deep) deterministic policy gradient methods that are designed for continuous action / state spaces.

7.5.1 Double DQN

We have already mentioned double learning in section 4.4 as a technique to counter the tendency of Q learning to overestimate values, especially at the beginning of the learning process. Double DQN [13] is double learning applied to deep Q networks. The overestimation is kept in check by evaluating the value function with a separate set of weights - which usually leads to a lower Q value. Used in conjunction with fixed Q targets obviates the need to train two networks to get that second set of weights, as there is always one set of weights being held constant for computing target values.

7.5.2 Prioritised Replay

Instead of choosing a minibatch of stored experiences at random, [8] explore the idea of prioritising some over others, based, for example, on the magnitude of the TD error, or the norm of the weight change induced by the replay.

7.5.3 Dueling Networks

It is commonly found in reinforcement learning problems that the advantage of one action over others for some given state is relatively small compared to the average value for that state and that it would be sufficient to learn the state value rather than the value for each state-action pair. [15] therefore proposes to decouple modelling state from actions values. Instead of outputting a vector of Q values, one for each action, the network output is split across two streams: one outputs a scalar state value, the other action-specific offsets or *advantages* (the general idea of thus decomposing the Q value goes back to [1]). The network is shown to substantially speed up convergence over the single-stream DQN and to achieve better overall performance.

8 Policy gradients

Value functions were introduced as a means to compute optimal policies. The policy is implicitly defined by the value function and it's the policy we are ultimately interested in, not the value function. An obvious approach would be to optimise policies directly, without using value functions at all, or at least not in their role as policy representations. This is what policy gradient methods are about. Conceptually, this class of methods is pretty straightforward. A policy is modelled as a parameterised function $\pi_\theta(a, s)$ that maps state-action pairs onto action probabilities (for discrete action spaces) or densities (for continuous action spaces). Given some reasonable performance measure in terms of that policy $J(\theta)$, we then compute the performance gradient $\nabla J(\theta)$ based on samples under π_θ with respect to θ and move θ in that direction.

Before we look at the details, let's reflect on possible advantages that policy gradient methods may have over value-based methods.

1. Because we model the policy explicitly, we can easily learn stochastic policies. With value-based approaches, one would need to somehow map action values to probabilities. Of course, a softmax would do the trick but it would be just that, a trick that may not ultimately yield optimal performance. In real-world situations, stochastic policies are often optimal either because of the nature of the problem (e.g. the classic paper-rock-scissors game), or because the environment is only partially observable and some states, though different, are perceived as the same. These *aliased* states are best handled with a stochastic policy.

2. If the action space is continuous or high-dimensional (or even both!) deriving the optimal action from a value function may be a difficult optimisation problem. If we model the policy as a deterministic policy, we obtain the best action at no cost.

3. For certain tasks, the policy may be less complex than the value function, in which case it would be more efficient to optimise the policy than the value function. This advantage of course only comes into play as long as there is good reason to believe - beforehand - that the policy is indeed relatively simple.

8.1 Policy performance

A policy's performance can be measured in a number of ways, for example as the final return for episodic tasks, as the average reward obtained per time step

$$J_{\bar{R}}(\theta) = \mathbb{E}[R],$$

or as an estimate of the return in the form of state or action values:

$$J_{\bar{q}}(\theta) = \mathbb{E}[Q(s, a)].$$

Reintroducing q here seems to run against our original goal of finding policies without having to deal with value functions. Note though that the value function has a different role: it is no longer used to implicitly *define* a policy but to measure its performance. Policy gradient methods that rely on value functions are known as actor-critic methods. They are amongst the earliest methods studied in the context of reinforcement learning but have since come a long way. We'll turn to them later.

8.2 Maximising J

Once J is defined, we can use stochastic hill climbing with all the available tricks like simulated annealing or adaptive noise scaling to find a good local maximum. Or better still, we use the policy gradients with respect to the parameters θ to guide our search for a local maximum. If J is not differentiable we can approximate the gradients using finite differences

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

for each dimension k where u_k is the unit vector along the kth dimension. This, however, can be expensive as we need to collect samples every time we want to estimate J for a different θ. One can use a more efficient gradient estimation method as long as the distribution function is differentiable with respect to the parameters.

8.2.1 Score function gradient estimation

The expected value of J depends on the distribution of states encountered and the actions taken in those states. The policy only gives us the distribution over actions for a particular state, but not the resulting distribution of states. How does J, and in particular its gradient ∇J depend on θ? Somewhat surprisingly, one can show that for a range of different objectives, the gradient of the ex-

pectation $\nabla_\theta J(\theta)$ can be written as the expectation over a term involving the gradient of the policy. Concretely, the policy gradient theorem establishes that for any differentiable policy $\pi_\theta(s, a)$, the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \ln \pi_\theta(s, a) Q^{\pi_\theta}(s, a)\Big]$$

where J can be the value of the starting state (for episodic tasks), the average reward, or the average value per time step (for non-episodic tasks). $\nabla \ln \pi_\theta$ is called the score function.

Note that the expectation is over all actions and states (as encountered by following π_θ). Do we know that distribution? No. But we can sample it by interacting with the environment and thus we can estimate $\nabla_\theta J(\theta)$ stochastically as

$$\nabla \hat{J}(\theta) = \frac{1}{N} \sum \nabla \ln \pi_\theta(s, a) Q^{\pi_\theta}(s, a).$$

Our best bet at maximising J is to take a step in the direction of the estimated gradient

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta).$$

This leads to the simplest Monte Carlo policy gradient algorithm known as REINFORCE due to Williams [17].

Algorithm 8 REINFORCE

initialise θ arbitrarily
for all episodes E $= \{s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T\}$ **do**
    for all $t = 0$ to $T - 1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} r_k$
        $\theta \leftarrow \theta + \alpha \gamma^t G \nabla_\theta \ln \pi_\theta(s_t, a_t)$

8.3 Introducing baselines

The Monte Carlo samples of the gradient converge to the true expectation, but the sample variance tends to be high so in practice policies are slow to converge. A simple trick allows us to reduce the variance whilst keeping the average true to the real expectation. The idea is to subtract another function $b(s)$ from the sampled performances such that the performance measures exhibit less variability. That function $b(s)$ is arbitrary as long as it does not depend on the action. Concretely, we modify the expectation to the following:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \ln \pi_\theta(s,a)\Big(Q^{\pi_\theta}(s,a) - b(s)\Big)\Big].$$

Denoting by $\mu(s)$ the probability distribution over states (defined implicitly by the policy and the environment), we observe that

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \ln \pi_\theta(s,a) b(s)\Big] &= \sum_s \mu(s) \sum_a \pi_\theta b(s) \nabla_\theta \ln \pi_\theta(s,a) \\
&= \sum_s \mu(s) b(s) \sum_a \pi_\theta \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta} \\
&= \sum_s \mu(s) b(s) \nabla_\theta \sum_a \pi_\theta \\
&= \sum_s \mu(s) b(s) \nabla_\theta 1 \\
&= 0.
\end{aligned}$$

The expectation therefore remains unchanged, but the variance can be reduced considerably with a suitable choice of b. The policy update with baseline becomes

$$\theta_{t+1} = \theta_t + \alpha \gamma^t (G_t - b(s_t)) \nabla_\theta \ln \pi_\theta(s_t, a_t)$$

where G_t is the total return counting from t till the end of the episode.

A typical choice for b is the state value function $v(s)$. The function $G_t - v(s_t)$, known as the *advantage function*, captures the *relative* advantage of one action over the average value, has zero mean and much reduced variance than G_t. The state value function can be updated at the same time as the policy parameters, at the end of each episode, as shown in the pseudocode below

Algorithm 9 REINFORCE with baseline

initialise θ and w arbitrarily
choose two step sizes α, β for updating parameter vectors θ and w
for all episodes E $= \{s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T\}$ **do**
    for all $t = 0$ to $T - 1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} r_k$
        $\delta \leftarrow G - \hat{v}(s_t, w)$
        $\theta \leftarrow \theta + \alpha\gamma^t \delta \nabla_\theta \ln \pi_\theta(s_t, a_t)$
        $w \leftarrow w + \beta\gamma^t \delta \nabla_w \hat{v}(s_t, w)$

8.4 Actor-critic methods

Actor-critic methods are policy gradient methods that separately model both the policy (the *actor*) and the value function (the *critic*). The above REINFORCE with baseline algorithm already does that, yet one could argue that it is not strictly an actor-critic method because it is merely used to reduce sample variance in the gradient estimate [11]. The real signal that drives a change in the policy is the total return G. Actor-critic methods use an approximation of the state or action value function to compute an estimate of the policy gradient: the 'actor' interacts with the environment according to some policy, which is separately 'critiqued' using a value function approximation.

Because we are not concerned with the total return, actor-critic methods can be applied to non-episodic tasks. The following algorithm describes a simple actor-critic method that uses a linear function approximation of the action value function $q_w(s, a) = \phi(s, a)^T w$ to evaluate the policy:

Algorithm 10 Action-Critic with Q

initialise θ and w arbitrarily
choose two step sizes α, β for updating parameter vectors θ and w
sample $a \sim \pi_\theta$
for all steps **do**
    sample reward r and next state s
    sample next action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma q_w(s', a') - q_w(s, a)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \ln \pi_\theta(s, a) q_w(s, a)$
    $w \leftarrow w + \beta\delta\phi(s, a)$
    $a \leftarrow a'$, $s \leftarrow s'$

Instead of using the action value $q_w(s, a)$ to compute the policy gradient, one can again reduce the variance by using the advantage function $q^{\pi_\theta}(s, a) - v^{\pi_\theta}(s)$. Both components, the action and state value function, can be approximated using two separate function approximators and parameter vectors, with both

being updated as in the aforementioned algorithm. But we can do better by noting that the advantage function can be written in terms of the TD error:

$$\begin{aligned} A^{\pi_\theta}(s,a) &= q^{\pi_\theta}(s,a) - v^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}\Big[r + \gamma v^{\pi_\theta}(s') \Big| s, a\Big] - v^{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}[\delta^{\pi_\theta}] \end{aligned}$$

Because the TD error is therefore an unbiased estimator of the advantage function, we can use the sampled TD error

$$\delta^{\pi_\theta} = r + \gamma v^{\pi_\theta}(s') - v^{\pi_\theta}(s)$$

or, rather, our best estimate using our current approximation of v

$$\delta = r + \gamma v(s') - v(s).$$

With this method, we only need to update v, rather than both v and q.

8.5 Deterministic Policy Gradients

The policy gradient theorem encountered previously applies to stochastic policies that map state-action pairs onto probabilities or densities. Silver *et al.* [9] showed that a similar result holds for deterministic policies $\pi(s)$. Concretely,

$$\nabla_\theta J(\theta) = \mathbb{E}_s\left[\nabla_\theta \pi_\theta(s,a) \nabla_a Q(s,a)\right]$$

where the expectation is over the distribution of states that would result from following policy π_θ. We can therefore approximate the policy gradient by taking samples of s and a and, for each sample, computing the product of the policy gradient (wrt θ) and the action-value gradient (wrt the action).

In [5], the deterministic policy gradient is applied within an actor-critic framework with both actor and critic represented by a deep neural network. If the agent were to follow the deterministic policy *during learning*, it would not sufficiently explore the space of states and actions. It is therefore critical that the agent follows a noisy version of the policy being learned.

9 Conclusions

The field of reinforcement learning is advancing fast, as is machine learning in general. Keeping up with progress is as simple as being a frequent visitor

(and reader) of openai.com, Andrew Ng's non-profit company that is committed to advancing the state of the art of reinforcement learning all the while ensuring that it remains a public good that's accessible by everyone. It provides implementations of many algorithms at github.com/openai/baselines and a gym environment at gym.openai.com that lets you learn and evaluate your own reinforcement learning algorithms.

References

[1] L Baird. Advantage updating. In *Technical Report WL-TR-93-1146, Wright-Patterson Air Force Base*, 1993.

[2] G Chaslot, M Winands, J Uiterwijk, H van den Herik, and B Bouzy. Progressive strategies for monte-carlo tree search. In *Proc Joint Conf Information Sciences (JCIS)*, 2007.

[3] R Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proc Int'l Conf Computers and Games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] L Kocsis and C Szepesvári. Bandit based monte-carlo planning. In *Proc European Conf Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] T Lillicrap, J Hunt, A Pritzel, N Heess, T Erez, Y Tassa, D Silver, and D Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[6] V Mnih, K Kavukcuoglu, D Silver, A Rusu, J Veness, M Bellemare, A Graves, M Riedmiller, A Fidjeland, G Ostrovski, S Petersen, C Beattie, A Sadik, I Antonoglou, H King, D Kumaran, D Wierstra, S Legg, and D Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[7] A Ng. *Shaping and policy search in reinforcement learning.* PhD thesis, University of California, Berkeley, 2003.

[8] T Schaul, J Quan, I Antonoglou, and D Silver. Prioritized experience replay. In *ICLR*, 2016.

[9] D Silver, G Lever, N Heess, T Degris, D Wierstra, and M Riedmiller. Deterministic policy gradient algorithms. In *Proc ICML*, 2014.

[10] R Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc Int'l Conf Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[11] R Sutton and A Barto. *Reinforcement learning: an introduction (in progress).* 2018.

[12] H v. Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems 23*, pages 2613–2621, 2010.

[13] H van Hasselt, A Guez, and D Silver. Deep reinforcement learning with double Q-learning. *CoRR*, abs/1509.06461, 2015.

[14] H van Seijen, H van Hasselt, S Whiteson, and M Wiering. A theoretical and empirical analysis of expected sarsa. In *Proc IEEE Symp Adaptive Dynamic Programming and Reinforcement Learning*, pages 177–18, 2009.

[15] Z Wang, T Schaul, M Hessel, H van Hasselt, M Lanctot, and N de Freitas. Dueling network architectures for deep reinforcement learning. *https://arxiv.org/abs/1511.06581*, 2016.

[16] S Whiteson, M Taylor, and P Stone. Adaptive tile coding for value function approximation. In *AI Technical Report*, 2007.

[17] R Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.