Alex Van de Kleut Department of Neuroscience
Brock University
St. Catharines, Canada
Email: av15fj@brocku.ca

Abstract

A. Notation

Capital letters appearing inside expectation expressions denote 'placeholders' for variables, whereas lowercase letters appearing inside expectation expressions denote actual values.

$$\mathbb{E}\big[R_t|S_t = s\big]$$

R_t, S_t are hypothetical and s is real.

CONTENTS

I. Background

The field of reinforcement learning (RL) in computer science is historically rooted in the field of operant conditioning from psychology. Operant conditioning is a technique to modify behaviour through the use of reward and punishment: behaviour should increase in frequency when associated the addition of a positive stimulus, and should decrease in frequency when associated with the removal of a positive stimulus or the addition of a negative stimulus [**?**].

From this, RL researchers have gleaned a guiding principle that informs every development in the field: the **reward hypothesis**. The reward hypothesis states that *every action of a rational agent can be thought of as seeking to maximize some cumulative scalar reward signal* [**?**]. The

reward hypothesis is foundational to reinforcement learning, since it gives us a basic framework for designing agents that behave rationally.

Reinforcement learning relies heavily on its theoretical foundations. Problems in reinforcement learning are framed as **Markov Decision Processes** (MDPs). MDPs are extensions of stochastic models known as **Markov Processes**.

A. Markov Processes

A Markov Process is, formally, a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where $\mathcal{S}$ is a set of states and $\mathcal{P} : \mathcal{S}^2 \to [0, 1]$ is a function describing the probability of transitioning from state s to state s':

$$\mathcal{P}_{s,s'} = \mathbb{P}\big[S_{t+1} = s' | S_t = s\big] \tag{I.1}$$

Markov processes are used to model stochastic sequences of states $s_1, s_2, \ldots, s_T$ satisfying the **Markov Property**:

$$\mathbb{P}\big[S_{t+1} | S_t\big] = \mathbb{P}\big[S_{t+1} | S_1, S_2, \ldots, S_t\big] \tag{I.2}$$

that is, the probability of transition from state S_t to state S_{t+1} is independent of previous transitions.

B. Markov Reward Processes

A **Markov Reward Process** is an extension of a Markov Process that allows us to associate rewards with states. Formally, it is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle$ that allows us to associate with each state some reward

$$\mathcal{R}_s = \mathbb{E}\big[R_t | S_t = s\big] \tag{I.3}$$

the expected reward of being in state s. [1]

Consider a sequence of states

$$s_t, s_{t+1}, s_{t+2}, \ldots, s_T$$

visited in a Markov Reward process, associated with a sequence of rewards

$$r_t, r_{t+1}, r_{t+2}, \ldots, r_T \tag{I.4}$$

[1] Some authors prefer to associate the reward with the next time step $t+1$ rather than the current one. As a matter of preference I opt for associating it with the current time step t.

We may be interested in finding sequences that maximize (I.4). In practice, we modify the above expression so that rewards that occur later in the sequence are weighted less. Even if we could identify a sequence that maximizes (I.4), we cannot guarantee that such a sequence will occur due to stochasticity of the environment, and since our prediction error increases exponentially over time, we might consider discounting rewards exponentially over time as well. This line of reasoning leads us to the **return** G_t, the return starting at time point t:

$$\begin{aligned} G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^{T-t} R_T \\ &= \sum_{k=t}^{T} \gamma^{k-t} R_k \end{aligned} \tag{I.5}$$

where γ is a discount factor between 0 and 1.

C. Value Function

G_t above is a general equation describing the return from a sequence of rewards. In practice, we can use the expected value of G_t to determine the **value** of a certain state s:

$$V(s) = \mathbb{E}\big[G_t | S_t = s\big] \tag{I.6}$$

We can decompose $V(s)$ into two parts: the immediate reward R_t and the discounted value of being in the next state S_{t+1}:

$$\begin{aligned} V(s) &= \mathbb{E}\big[G_t | S_t = s\big] \\ &= \mathbb{E}\big[R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t} R_T | S_t = s\big] \\ &= \mathbb{E}\big[R_t + \gamma (R_{t+1} + \cdots + \gamma^{T-t-1} R_T) | S_t = s\big] \\ &= \mathbb{E}\big[R_t + \gamma G_{t+1} | S_t = s\big] \\ &= \mathbb{E}\big[R_t + \gamma V(S_{t+1}) | S_t = s\big] \end{aligned} \tag{I.7}$$

The last form of $V(s)$ in (I.7) is known as the **Bellman Equation**. We can evaluate the expectation operator $\mathbb{E}$ to obtain a recursive formulation:

$$V(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} V(s') \tag{I.8}$$

where the first term is the expected reward of being in state s and the second term is the discounted weighted sum of the values of the next possible states.

D. Markov Decision Processes

A **Markov Decision Process** (MDP) is an extension of a Markov Reward Process that allows state transitions to be conditional upon some action. Formally, it is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{A}$ is a set of action available to an agent in a state s. We reformulate (I.3) as follows:

$$\mathcal{R}_s^a = \mathbb{E}\big[R_t | S_t = s, A_t = a\big] \tag{I.9}$$

This is the model we will use to describe problems in reinforcement learning. A common schema we will use when discussing MDPs is that of a **transition**, which takes the form

$$\langle s_t, a_t, r_t, s_{t+1} \rangle$$

where an agent in state s_t takes action a_t, causing a transition to state s_{t+1} and recieving a reward r_t as a result.

Note that we must also update $\mathcal{P}$ to be the probability of transitioning to state s_{t+1} given that the current state is s_t and the current action is a_t.

$$\mathcal{P}_{s,a,s'} = \mathbb{P}\big[S' = s' | S = s, A = a\big] \tag{I.10}$$

E. Policies

Our goal is to design an agent capable of behaving rationally, that is, capable of maximizing the return. In the context of MDPs, this means having a strategy for choosing an action a_t given the state s_t. One approach is to describe a **policy** for the agent:

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$$

which is taken to be a probability distribution over the possible actions the agent may take

$$\pi(a|s) = \mathbb{P}\big[A_t = a | S_t = s\big] \tag{I.11}$$

Given a policy π, an agent can now choose actions at each state to shape the sequences of states that it visits. We thus have a new formulation of the value function (I.6):

$$V_\pi(s) = \mathbb{E}\big[G_t | S_t = s\big] \tag{I.12}$$

which can be thought of as the expected value of starting in state s and choosing actions in subsequent states according to the policy π.

F. Action-Value Function

We can extend our redefined function (I.12) to consider the expected return of taking action a_t in state s_t and from there following the policy π at each subsequent state.

$$Q_\pi(s,a) = \mathbb{E}_\pi\big[G_t | S_t = s, A_t = a\big] \tag{I.13}$$

Whereas V_π associates a 'goodness' with a state s according to a policy π, $Q_\pi(s,a)$ describes the **quality** of taking an action a in a state s.

Just as in (I.7), we can decompose $Q_\pi(s,a)$ as follows:

$$Q_\pi(s,a) = \mathbb{E}_\pi\big[R_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) | S_t = s, A_t = a\big] \tag{I.14}$$

G. Incorporating Stochasticity

Consider being in some state s. With each action a available from this state, we can consider the quality $Q_\pi(s,a)$ of performing this action. From this, we can define the value $V_\pi(s)$ of a certain state by simply taking the quality of each possible action and weighting it by the likelihood of choosing that action. Thankfully, we already have a definition for how to choose that action! This is exactly our policy π.

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s,a) \tag{I.15}$$

Conversely, consider choosing some action a in some state s. With each state s' available as a result of this action, consider the value $V_\pi(s')$ of this state. We can define the quality of choosing an action a in a state s as the expected reward of choosing action a in state s combined with the value of the next possible states, weighted by the likelihood of the transition from s to s'.

$$Q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} V_\pi(s') \tag{I.16}$$

We can then incorporate (I.15) into (I.16) to get

$$Q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \sum_{a' \in \mathcal{A}} \pi(a|s') Q_\pi(s',a') \tag{I.17}$$

H. Optimal Policy

Under the paradigm of the Q function, what does it mean for an agent to have an **optimal policy**? We say that an agent learns an optimal policy Q^* when:

1) Q^* is the exact quality of a state-action pair (s,a)

2) Q^* follows a policy π^* that maximizes the quality:

$$Q^*(s,a) = \max_{\pi} Q_\pi(s,a) \tag{I.18}$$

By defining the optimal policy to be that which maximizes $Q(s,a)$ over all state-action pairs, we derive a greedy optimal policy:

$$\pi^*(a|s) = \begin{cases} 1, & a = \arg\max_a Q^*(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{I.19}$$

The optimal form of the Bellman Equation is thus simply a modification of (I.17) according to (I.19)

$$Q^*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \max_{a'} Q^*(s',a') \tag{I.20}$$

where the weighted sum of $Q_\pi(s',a')$ over possible next actions a' collapses to the single optimal value of Q^*.

II. Q-Learning

A. SARSA

Some environments give the agent a **discrete** action space $\mathcal{A}$. These kinds of environments are often simpler to learn than cases where the environment may permit a **continuous** action space. We often run into discrete action space environments in games, where at each turn there is a small number of moves to make, or video games, where at each time step you can only choose combinations of button presses.

'Learning' an environment means being able to learn the Q function. However, the formulations given in (I.14) and (I.17) are recursive. This is impractical for many reasons; in the real world it is not possible to test every action since the environment would change as a result (i.e., a state transition would occur). Furthermore, unless if it is possible to reach the same state s' from state s through more than one sequence of actions, then recursion would result in repeated work. As a result, we need to develop an approach that can handle the environment changing as a result of our actions.

One method is to use the SARSA algorithm, which is an abbreviation of

$$\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$$

A sequence of experiences of an agent that can be used to learn the Q function. Consider the following sequence of events:

1) The agent is in a state s_t, and chooses some action a_t according to a policy π.
2) The agent transitions from state s_t to state s_{t+1}, recieving a reward r_t.
3) The agent is in a state s_{t+1} and chooses some action a_{t+1} according to a policy π.

At this point, the agent has a better estimate of $Q(s_t, a_t)$, namely

$$r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) \tag{II.1}$$

We refer to this estimate as the time-difference target or **TD target**. Then our estimate of $Q_\pi(s_t, a_t)$ can be updated according to some learning rate α as follows:

$$Q_\pi(s_t, a_t) \leftarrow (1-\alpha)Q_\pi(s_t, a_t) + \alpha(r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})) \tag{II.2}$$

It is important to note here that at time t and also $t+1$ we use the policy π to select actions a_t and a_{t+1}. If we make the logical assumption that an agent uses the Q function to guide π at each step, then we are both updating our policy and using it to guide our decisions at the same time. This is called **on-policy** learning.

B. Q-learning

Q-learning is essentially SARSA with the policy π being exactly the kind of greedy policy described in (I.19):

$$\pi(a|s) = \begin{cases} 1, & a = \arg\max_a Q(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{II.3}$$

With this in mind, we can reformulate the sequence of events considered in the SARSA algorithm as follows:

1) The agent is in a state s_t and for each possible action a_t calculates $Q_\pi(s_t, a_t)$. The agent chooses the action a_t that maximizes Q.
2) The agent transitions from state s_t to state s_{t+1} recieving a reward r_t.
3) The agent is in a state s_{t+1} and for each possible action a_{t+1} calculates $Q_\pi(s_{t+1}, a_{t+1})$. The agent chooses the action a_{t+1} that maximizes Q.

Then we simply modify the update rule for SARSA in (II.2):

$$Q_\pi(s_t, a_t) \leftarrow (1-\alpha)Q_\pi(s_t, a_t) + \alpha(r_t + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})) \tag{II.4}$$

C. Deep Q-Learning

Implicit in the above formulation of Q-learning is the 'storage' of $Q(s, a)$ for state-action pairs (s, a). While a tabular approach to Q-learning may be feasible for environments with very small state spaces and action spaces (i.e., a table of size $|\mathcal{S}| \times |\mathcal{A}|$), there are problems for large state or action spaces, or potentially infinite state spaces.[2]

One approch to solving this is to use a function approximator for Q that takes state-action pairs (s, a) and produces a scalar prediction for what $Q(s, a)$ should be. An extremely popular approach is to use a **deep neural network** to approximate Q. Briefly, a deep neural network is a differentiable computational graph made up of layers of processing nodes. See figure 1 for a schematic.

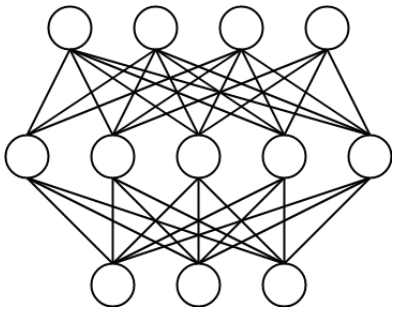


Fig. 1. A schematic representation of a deep neural network.

One big advantage to using a neural network is the ability of the neural network to output multiple values. Instead of the neural network taking state-action pairs and output scalar values, they can instead take states as input and produce vectors of Q values corresponding to the quality of each action in that state. This is a much more efficient approach. We call this network the **Q-network** (DQN - deep Q-network for short). See figure 2 for a schematic.

Neural networks need to be differentiable so that we can use the optimization technique of **gradient descent** to train it. Typically a neural network is specified abstractly as a set of parameters θ that determine the output of the network given the input. We define a loss L for the network that we want to minimize. Gradient descent works by taking the gradient of L with respect to θ and taking a small step in the direction opposite the gradient ('down' the gradient, i.e., gradient descent). This is the basic formulation; several modern extensions of

[2]Note that the action space must still be discrete in Q-learning.

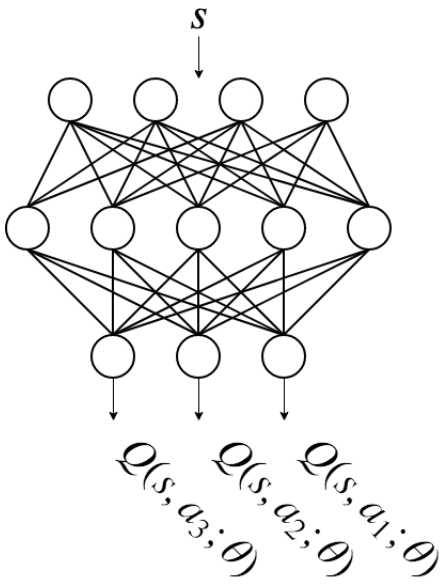


Fig. 2. A Q-network. The state is a 4-dimensional vector and there are 3 discrete actions available. The network takes a state s as a parameter and for each action a predicts the quality $Q(s, a; \theta)$ of that action.

gradient descent exist that improve training of neural networks, the details of which we will not cover in this thesis.

Consider the Q-learning update rule (II.4). When $Q_\pi(s_t, a_t)$ is exactly equal to the TD target, there is no update. Noticing this, we might consider the loss of our neural network to be 0 when the value for $Q(s_t, a_t)$ is equal to $r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$. This is exactly the framework of a regression problem. We can then define the loss to be the squared error between the two:

$$L(\theta) = \mathbb{E}_{a\sim\pi}[(y_i - Q(s, a; \theta))^2] \tag{II.5}$$

where the TD target is y_i:

$$y_i = \mathbb{E}_{a'\sim\pi}\big[R_t + \gamma \max_{a'} Q(s', a'; \theta) | S_t = s, A_t = a\big] \tag{II.6}$$

Note that here instead of writing $Q_\pi(s_t, a_t)$ we now write $Q(s_t, a_t; \theta)$. This is because our Q value is now no longer driven by the policy of the agent; the policy is defined to be the greedy policy and the output of the Q function is determined by the network parameters θ (as well as s_t and a_t).

Training the neural network would consist of the following sequence of events:

1) The agent in state s_t calculates $Q(s_t, a_t; \theta)$ using network parameters θ for each possible a_t. Using the greedy policy, it selects the action a_t that maximizes $Q(s_t, a_t; \theta)$.
2) As a result of choosing this action, the state transitions to state s_{t+1}. The agent recieves a reward r_t as a result.
3) The agent is now in state s_{t+1} calculates $Q(s_{t+1}, a_{t+1}; \theta)$ using network parameters θ for each possible a_{t+1}. The maximal value of $Q(s_{t+1}, a_{t+1}; \theta)$ is chosen.
4) The network is trained to minimize the loss, with $Q(s_t, a_t; \theta)$ being the prediction of the network, and with y_i being the reward for transitioning and the discounted maximal Q value for state s_{t+1} determined in step 4 (i.e., $r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta)$).

See figure 3 for a diagrammatic representation.

To perform any variation of gradient descent requires defining the gradient of the loss function. In this case, we can just apply the chain rule to (II.6). If we treat the TD target y_i as a constant (which is appropriate), we get a rather simple expression for the gradient:

$$\nabla_\theta L(\theta) = \mathbb{E}_{a\sim\pi}\big[2(y_i - Q(s, a; \theta))\nabla_\theta Q(s, a; \theta)\big] \tag{II.7}$$

and to update the the network parameters, we simply use **stochastic gradient descent**:

$$\theta \leftarrow \theta + \frac{1}{2}\alpha\nabla_\theta L(\theta) \tag{II.8}$$

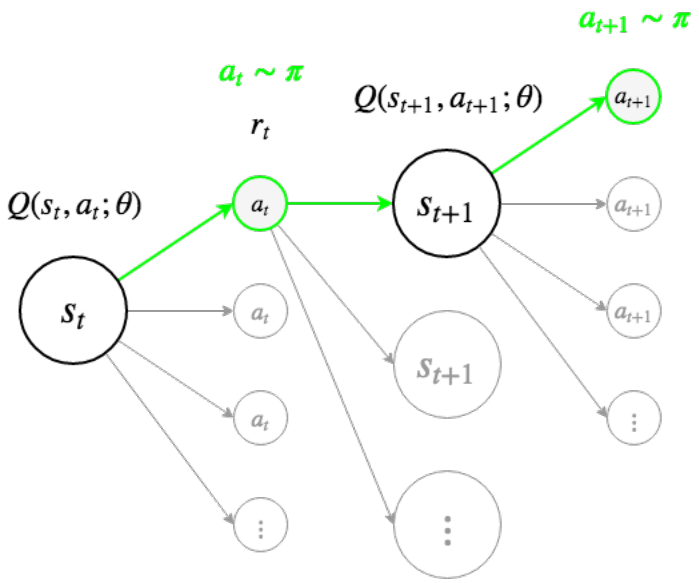
where α is the learning rate.

D. Target Networks

One may have realized that this network θ is trying to predict it own output. This kind of learning is unstable, meaning performance can quickly deteriorate (to even worse than random [**?**]). One reason for this is that the neural network is differentiable. When we modify the parameters θ while training the network, we actually change the predictions for similar states to that which we just trained on. Research has shown that training can be stabilized by using two networks: θ_i and θ_{i-1}. Instead of one set of parameters, we have two (i.e., we have two neural networks). We call the second network (θ_{i-1}) the **target network**, and it uses the parameters from the Q-network that synchronize with the current Q-network every n_θ timesteps.

We define two policies:

1) μ: The **behaviour policy** that uses the Q-network ($Q(s, a; \theta_i)$) and is updated every time step.



$$L_i(\theta) = (\, r_t + \gamma Q(s_{t+1}, a_{t+1}; \theta - Q(s_t, a_t; \theta)\,)^2$$

Fig. 3. A diagram showing how Q-learning gathers data for training.

2) π: The **target policy** that uses the target network ($Q(s', a'; \theta_{i-1})$) and is updated to match μ (i.e., θ_{i-1} is updated to match θ_i) every n_θ timesteps.

The fact that the target network has fixed parameters for n_θ timesteps and predicts similarly to the Q-network stabilizes the learning.

As a result, we get a new formulation of (II.5):

$$L_i(\theta_i) = \mathbb{E}_{a\sim\mu}\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{II.9}$$

where the TD target is y_i:

$$y_i = \mathbb{E}_{a'\sim\pi} R_t + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | S_t = s, A_t = a] \tag{II.10}$$

See figure 4 for a diagrammatic representation.

E. Exploration-Exploitation

One major problem in RL is determining the optimal balance between **exploration** and **exploitation**. Exploitation refers to an agents tendency to choose actions that it thinks are best for it at any given time. This is the case with our greedy behaviour policy μ. This comes at the

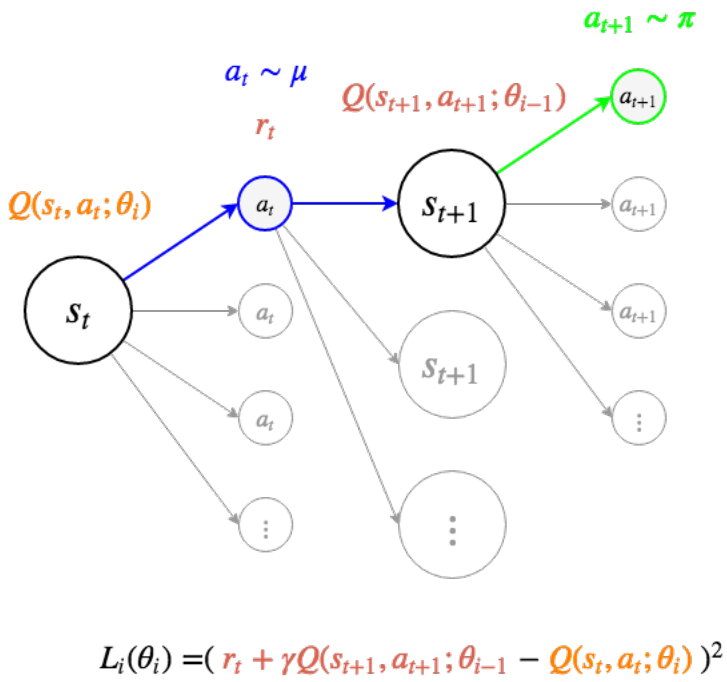


Fig. 4. A diagram showing how Q-learning gathers data for training using the target network and behaviour network.

cost of potentially not exploring new states that the agent has yet to visit. For the neural network to well approximate the Q function, it needs to have exposure to as much of $\mathcal{S}$ as possible.

One strategy is to select a constant $0 \leq \epsilon \leq 1$. We use ϵ to choose an action either randomly or according to μ by comparing a random number $p \sim \mathcal{U}(0, 1)$ to ϵ; if $p < \epsilon$ we choose a random action. Otherwise, we choose an action according to μ.

There are various strategies one can use for ϵ-greedy policies. One example is to choose some initial value ϵ_i and some final value ϵ_f, and a number of iterations n_ϵ over which ϵ_i decays into ϵ_f. Another strategy is to choose ϵ empirically:

$$\epsilon = \frac{1}{\sqrt{T+1}}$$

where T is the number of iterations (state transitions) in the episode.

F. Experience Replay

When we update out Q-network, we do it after every SARSA transition. Recall that we are using gradient descent to train the network parameters θ_i. One problem with this is that we get a biased estimate for the gradient. A better estimate would involve taking the mean gradient

estimate over many pairs of $Q(s, a)$ and TD targets. Furthermore, by training the network only on the most recent SARSA transition, we are making the network better at predicting the most recent TD target. This can actually undo some of the progress made training the network on earlier transitions. This problem is known as **catastrophic forgetting**.

We can solve the above problems by including an **experience replay buffer** $\mathcal{D}$ that stores transitions

$$\langle s_t, a_t, r_t, s_{t+1} \rangle$$

Note that we do not need to store a_{t+1} since our network will choose a_{t+1} based on π anyways.

We then train the network by taking batches of transitions from $\mathcal{D}$. For each transition, we use the Q-network to predict $Q(s_t, a_t; \theta_i)$ and to consequently choose an action $a_t \sim \mu$, and we use the target network to predict $Q(s_{t+1}, a_{t+1}; \theta_{i-1})$ and to consequently choose an action $a_{t+1} \sim \pi$. Over all transitions in the batch, we calculate the gradient of the loss $L_i(\theta_i)$ of the Q-network. We take the average of these gradients and update the network parameters.

G. Double Q-Learning

A common theme you may have noticed is that many developments in **deep reinforcement learning** (using neural networks in RL) add new features to existing techniques. Above, we showed the addition of a target network, using an ϵ-greedy action selection strategy, and using a memory replay buffer. Double Q-learning is another addition that builds on these.

In the target network formulation of deep Q-learning, we use the target network to compute $Q(s_{t+1}, a_{t+1}; \theta_{i-1})$. We use π to greedily choose an action a_{t+1} as a result of this calculation. Thus, computing the quality of an action and choosing an action is tightly coupled. In double Q-learning, we decouple these.

We choose an action a_{t+1} based on the current behaviour policy μ (that is, we calculate $Q(s_{t+1}, a_{t+1}; \theta_i)$ and select greedily from the results). We then use this a_{t+1} to calculate y_i (that is, we use a_{t+1} to calculate $Q(s_{t+1}, a_{t+1}; \theta_{i-1})$). As a result, we can rewrite our TD target y_i:

$$y_i = \mathbb{E}_{a' \sim \mu}\big[r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta_{i-1})\big] \tag{II.11}$$

See figure 5 for a diagrammatic representation.

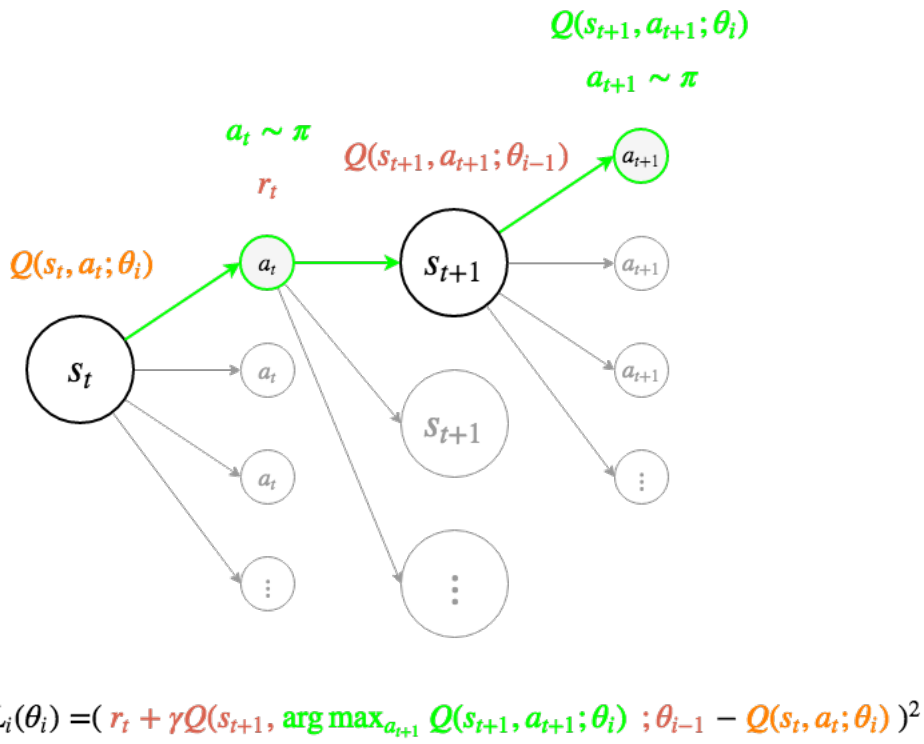


Fig. 5. A diagram showing how Q-learning gathers data for training using the double Q-learning.

III. Policy Gradient

In Q-learning, we have a policy π that is simply a greedy (or ϵ-greedy) strategy. While this can be effective in problems with small action spaces, we may face problems with large (or continuous) action spaces. We can circumvent this by instead directly trying to model π. We can parametrize the policy using some parameters θ to produce a distribution over actions:

$$\pi_\theta(a, s) = \mathbb{P}\big[A_t = a | S_t = s; \theta\big] \tag{III.1}$$

Modelling the policy directly has an additional advantage over the greedy policy from Q-learning, which is that it can learn a **stochastic policy**. We have the option of picking an action stochastically over the distribution π.

Consider some objective function J that we want to optimize using our policy (and consequently J is a function of θ). An obvious choice would be the expected rewards over all possible **trajectories** τ (that is, sequences of states, actions, and rewards):

$$J(\theta) = \mathbb{E}_{\pi_\theta}\big[r(\tau)\big] \tag{III.2}$$

where $r(\tau)$ represents the total reward over a trajectory τ. Finding an optimal policy π_θ amounts to finding a set of parameters θ that maximizes J. One way to do this is to use a differentiable function approximator for π such that we can perform **stochastic gradient ascent**

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{III.3}$$

A. The Policy Gradient Theorem

This simple formulation is rather decieving. We now have the rather difficult task of evaluating $\nabla_\theta J(\theta)$.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}\big[r(\tau)\big] \\ &= \int \pi_\theta(\tau) r(\tau) d\tau \end{aligned} \tag{III.4}$$

So we express the expectation of the reward over all trajectories τ as expected to be followed using policy π. Here, $\pi_\theta(\tau)$ represents the probability of generating the trajectort τ using policy π_θ. The gradient of J is

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \tag{III.5}$$

We can use the following identity:

$$\nabla_\theta \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} \tag{III.6}$$

$$= \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) \tag{III.7}$$

and substitute it into (III.5) to get

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \\ &= \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\big] \end{aligned} \tag{III.8}$$

We now need to consider the probability $\pi_\theta(\tau)$ of generating a trajectory τ. Let us define d to be the distribution of states, and let s_0 denote an initial state. Let the trajectory τ be over time steps $1, 2, \ldots, T$.

$$\pi_\theta(\tau) = d(s_0) \prod_{t=1}^{T} \pi_\theta(a_t | s_t; \theta) \mathcal{P}_{s_t, a_t, s_{t+1}} \tag{III.9}$$

We can read this as being the probability of being in an initial state AND the probability of choosing an action a_t given the state s_t and policy parameters θ AND the probability of

transitioning to state s_{t+1} given the current state and action. In (III.8) we are taking the $\log$ of $\pi_\theta(\tau)$. Doing so to (III.9) yields:

$$\log \pi_\theta(\tau) = \log d(s_0) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta) + \log \mathcal{P}_{s_t,a_t,s_{t+1}} \tag{III.10}$$

Finally, we take the gradient with respect to θ:

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta) \tag{III.11}$$

It turns out that since neither the distribution of states d nor the dynamics of the MDP $\mathcal{P}$ have an impact on the gradient! Thus, we finally get an expression for the gradient $\nabla_\theta J(\theta)$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta) r(\tau)\right] \tag{III.12}$$

We call this **the policy gradient theorem**. In fact, it is a slight variant of the policy gradient theorem where we use specific trajectories of actions rather than taking the expectation over all actions, but for implementation purposes it is sufficient [**?**].

B. REINFORCE

You may note that $r(\tau)$ above remains essentially untouched. In fact, since, past rewards don't influence future rewards, we may replace it with G_t (i.e., the return from time point t onwards). If we wish to take an expectation of this, we must sample trajectories. This requires knowing in advance the return G_t, which consequently requires the agent to finish an episode before updating parameters. This method of sampling complete trajectories and then making updates is called **episodic Monte Carlo policy gradient control**. It is episodic because the agent finishes an episode before updating. It is Monte Carlo because it samples complete trajectories according to the policy before making updates. This gives a modified version of (III.12):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta) G_t\right] \tag{III.13}$$

By considering this gradient, we get the REINFORCE algorithm:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_{\theta_t} \log \pi_{\theta_t}(a_t|s_t;\theta_t) G_t \tag{III.14}$$

At this point, it may be useful to undo the identity (III.6) and intuitively deconstruct why the REINFORCE update rules makes sense:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{\nabla_{\theta_t} \pi_{\theta_t}(a_t|s_t;\theta_t)}{\pi_{\theta_t}(a_t|s_t;\theta_t)} G_t$$

The gradient vector on the top of the fraction points in the direction (in parameter space) that most increases the probability of taking action a_t in state s_t. Using this alone could lead to a feedback loop where the agent continually makes itself more likely to choose this action over and over, so as a result, we divide it by the probability of taking that action. Finally, we scale it by the return to encourage behaviours that maximize the return.

C. Actor-Critic Models (AC)

The issue with the REINFORCE method is that it is a Monte Carlo method. First, it can be prohibitively slow to train, performing updates to the policy only after episode completion. Second, the Monte Carlo sampling approach can be extremely high variance (for example, maybe in one episode G_t is 100 and in another it is 0). Third, it requires training to be episodic (instead of ongoing). This motivates the development of the **actor-critic** model.

The actor-critic model combines the techniques of policy gradient and also Q-learning. Rather than completing entire episodes G_t and then updating (as in REINFORCE), we could simply try to estimate the quality of a state-action pair (s, a) and update the policy after each time step. This is exactly the goal of Q-learning. Let us use a function approximator (again, often a neural network) parametrized by some parameters ϕ to predict $Q_{\pi_\theta}(s, a)$ This gives us an approximation to the gradient of our objective function:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s, a; \theta) Q_{\pi_\theta}(s, a; \phi)\big] \tag{III.15}$$

We now have two sets of parameters to optimize. We can optimize θ (the actor) using the policy gradient in (III.15) and we can use deep Q-learning to optimize ϕ.[3]

D. Advantage Actor Critic (A2C)

Consider an environment where the rewards are not highly informative of the 'goodness' of choosing certain actions over others (e.g., the rewards are $100, 103, 102, \dots$). Conceptually, this means that the quality of choosing an action a in a state s is mostly just the value of the state $V(s)$. The difference in expected return between choosing an action a (as in the quality) versus

[3]The compatible function approximation theorem gives conditions under which the equality is exact. See **??** for details.

just following the policy from state s onwards (as in the value) is called the **advantage** $A(s,a)$ of choosing action a in state s:

$$Q_{\pi_\theta}(s,a) = V_{\pi_\theta}(s) + A_{\pi_\theta}(s,a) \tag{III.16}$$
$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s)$$

Clearly the advantage is a better estimate of the 'goodness' of certain actions, and furthermore has lower variance. We just need to confirm that substituting the advantage in for the quality of normal actor-critic models does not change our expectation in (III.15).

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta)\big(Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s)\big)\right]$$
$$\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t;\theta) V_{\pi_\theta}(s)\right] = \int \tag{III.17}$$

E. PPO

IV. Intrinsic Motivation

A. Sparse Rewards

B. ICM

C. Go-Explore

D. Episodic Curiosity through Reachability

V. Sparse Distributed Memory