1 Introduction to Reinforcement Learning

Edward Young - ey245@cam.ac.uk

Environments and policies

- RL involves an interaction between an **agent** and an **environment**.
- The agent chooses an **action** a_t to take based upon the current **state** s_t of the environment.
- As a result of that action, the environment returns a **reward** r_{t+1}, and transitions to a new state s_{t+1}.
- The interaction between the agent and the environment continues until we reach a **terminal state** s_T at time T. If termination occurs, the **done signal** $d_T = 1$. At all other time steps, the done signal $d_t = 0$.
- The collection of states for the environment is the **state space**, which we denote by $s \in \mathcal{S}$
- The collection of actions for the environment is the **action space**, which we denote by $a \in \mathcal{A}$.
- The environment is characterised by a **dynamics** function and a **reward** function.
    1. The dynamics function gives the probability of transitioning into state s', given that you take action a in state s. This is denoted $p(s'|s, a)$.
    2. The reward function indicates the reward that the agent receives for being in state s and taking action a, $r(s, a)$.
- The agent is characterised by a **policy**, which takes in a state s and returns a probability distribution over actions a, $\pi(a|s)$

Reward, Return, and Value

The **return** at time t is the sum of rewards obtained after time t until the time T at which the terminal state is reached, discounted according to how far in the future they are:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + \gamma^{T-t-1} r_T = \sum_{k=0}^{T-t-1} \gamma^k r_{t+1+k} \tag{1}$$

We call γ the **discount rate**. The discount rate quantifies how much we weight we place on nearby rewards vs. distant rewards.

Values are *expected* returns.

1. The **state-value function** of a policy π is the expected return, conditional on being in a particular state s and forever after selecting actions according to π:
$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{2}$$
2. The **action-value function** of a policy π is the expected return, conditional on being in a particular state s and taking a particular action a and forever after selecting actions according to π:
$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{3}$$
3. The **optimal action-value function** is the greatest possible action-value, over all possible policies:
$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \tag{4}$$

An **optimal policy** π^* is a policy which selects actions by maximising the optimal action-value function in each state.

Bellman equations

Bellman equations are *consistency* equations for value functions. They tell us the value at one time point in terms of the value of the next.

1. The state-value function of π:

$$V_\pi(s) = \mathbb{E}\left[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s, a_t \sim \pi(a|s)\right] \tag{5}$$

2. The action-value function of π:

$$Q_\pi(s,a) = \mathbb{E}\left[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a\right] \tag{6}$$

3. The optimal action-value function:

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a\right] \tag{7}$$

We can also form Bellman equations that relate action-value functions and state-value functions:

$$Q_\pi(s,a) = \mathbb{E}[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s, a_t = a] \tag{8}$$

and

$$V_\pi(s) = \mathbb{E}[Q_\pi(s_t, a_t) | s_t = s, a_t \sim \pi(a|s)] \tag{9}$$

2 DQN

DQN is an **action-value** method. This means that we learn an action-value function, and then use this to select actions. To learn the action-value function we will employ several tricks to turn the RL problem into a **supervised** learning problem.

Function approximation

Neural networks are **universal function approximators**. This means that, for any continuous function $f(x)$, given a sufficiently wide neural network $f(x;\phi)$, there are some parameters ϕ such that the network approximates $f(x)$ arbitrarily well:

$$f(x;\phi) \approx f(x) \text{ for some parameters } \phi \tag{10}$$

In DQN, we apply this principle to the **optimal action value function**, Eq. (4). That is, we will attempt to find parameters ϕ such that our neural network, $Q(s,a;\phi)$ approximates $Q^*(s,a)$:

$$Q(s,a;\phi) \approx Q^*(s,a) \tag{11}$$

The problem is finding the parameters ϕ which make Eq. (11) true. We will do this by **training** ϕ using **gradient descent** on a **supervised regression** loss.

Value regression

Any ML problem in which we are given input variables-target variable pairs (x,y), and we attempt to learn a mapping $f(x) = y$ is called a **supervised learning** problem. When the targets y are continuous (as opposed to discrete) this is called a **regression** problem. To train a neural network $f(x;\phi)$ on a regression problem, we typically perform **mini-batch gradient descent** on the **mean squared error loss**,

$$\mathcal{L}(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(f(x_i;\phi) - y_i\right)^2 \tag{12}$$

where the sum is over a **mini-batch** of N of our input-target pairs (x,y). We now apply this principle to our neural network $Q(s,a;\phi)$. The input variables are state-action pairs (s,a), and the target variable is $y = Q^*(s,a)$. This gives us the loss:

$$\mathcal{L}(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(Q(s_i,a_i;\phi) - y_i\right)^2 \tag{13}$$

Bootstrapping

There's only one problem - we can't use $y = Q^*(s,a)$ as the regression targets, since we don't know $Q^*(s,a)$! We must use a clever trick to form regression targets.

$$\begin{aligned}
y_t &= Q^*(s_t,a_t) && \text{What we want}\\
&= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1},a') | s_t, a_t\right] && \text{Eq. (7)}\\
&\approx r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1},a') && \text{Expectation } \rightarrow \text{ Sample}\\
&\approx r_{t+1} + \gamma(1-d_{t+1}) \max_{a'} Q(s_{t+1},a';\phi) && \text{Replace } Q^*(s,a) \text{ with } (1-d)\ Q(s,a;\phi)
\end{aligned}$$

In the last line, we replaced the (unknown) optimal action-value with our approximation to it, $Q(s,a;\phi)$. This allows us to learn *better* approximate values from our current approximate values. This is called **bootstrapping**.

Important: We do not back-propagate through the regression targets. They are treated as constants in the MSE loss.

The Memory Replay buffer

To form the MSE loss Eq. (12) we average over a **mini-batch** of N input-output pairs (x, y). In standard supervised learning, we sample mini-batches uniformly from a static dataset of input-output pairings. In RL our dataset is not static, because:

1. As we interact with the environment we see new state-action pairs (*i.e.*, non-static inputs)
2. As we learn our regression targets update because of bootstrapping (*i.e.*, non-static outputs)

To mitigate the first source of non-stationarity we use a **memory replay buffer**. When we interact with the environment see state-action-reward-next state, done **transitions**, (s, a, r, s', d). These are loaded in a store of the last M such experienced transitions called the **replay buffer**. When M is large, the replay buffer acts as a semi-static dataset.

Target network

We will employ another trick to mitigate the second source of non-stationarity. Because our regression targets y are formed using our network $Q(s, a; \phi)$, as we update the parameters of the network the regression targets also change. Because our network is trying to hit a moving target, learning can be unstable. We will replace the value network in the regression targets with a **target network**, which has the same architecture as the value network but uses **lagged parameters** ϕ^-. Given a **transition** (s, a, r, s', d), the regression target for (s, a) becomes

$$y = r + \gamma(1 - d) \max_{a'} Q(s', a'; \phi^-) \tag{14}$$

The parameters of the target network are periodically **synchronised** with those of the value network:

$$\phi^- \leftarrow \phi \tag{15}$$

ϵ-greedy policy

DQN is an **action-value** method. This means that it learns an action-value function, and uses this directly to select actions. When we evaluate our network we should pursue whichever action our value network says is best, $a \in \arg\max Q(s, a; \phi)$. This is called a **greedy** strategy. During training, we want to ensure that **explore** the environment to generate a *diversity* of experiences (and try out all our options). We therefore select actions according to an **ϵ-greedy policy**. With probability $1 - \epsilon$ we act greedily, *i.e.*, select the action we believe has highest value. With probability ϵ we randomly select an action.

Putting it together

DQN uses two networks, the **value network** (with parameters ϕ) and the **target network** (with parameters ϕ^-). There are three processes occurring on gradually longer timescales:

1. Gather **transitions** (s, a, r, s', d) using an **ϵ-greedy policy** and load them into the **memory replay buffer**.
2. Every 10 interaction steps, update the value network:
   (a) Randomly sample N transitions from the replay buffer
   (b) Form **regression targets** using Eq. (14)
   (c) Form the **mean squared error loss**, Eq. (16).
   (d) Perform a single gradient *descent* step on this loss.
3. Every 1000 gradient steps, **synchronise** the target network parameters, Eq. (15).

3 Advantage Actor-Critic (A2C)

A2C is an actor-critic method. This means that, in addition to learning a value function (a **critic**), A2C learns a policy (an **actor**), $\pi(a|s;\theta)$. In A2C, the value function is used to provide feedback on the action choices of the policy.

Algorithm outline

The Advantage Actor-Critic algorithm loops through the following steps:

1. Gather a batch of M **transitions** (s, a, r, s', d) of experience by interacting with our current policy $\pi(a|s;\theta)$.
2. Perform the value network update for this batch. Loop over a number of value network training steps:
   (a) Sample a mini-batch of N transitions from the batch.
   (b) Form regression targets using Eq. (17).
   (c) Form the **MSE loss** $\mathcal{L}(\phi)$, Eq. (16).
   (d) Perform a single gradient *descent* step on this loss.
3. Synchronise the target value network parameters, Eq. (15)
4. Update the policy network:
   (a) Loop through the batch, forming an **advantage estimate** for each data point using Eq. (19)
   (b) Form the **policy objective**, (20)
   (c) Perform a single gradient *ascent* step on this objective.

Core difference between A2C and DQN

1. DQN is an action value method, and so learns only a value function. A2C is an actor-critic method, and so learns both a value function and a policy.
2. A2C learns a **state-value function** $V(s;\phi)$, while DQN learns an **(optimal) action value function** $Q(s, a;\phi)$.
3. In DQN, interacting and learning are interleaved - we alternate between interacting for a few steps and performing one parameter update. In A2C interacting and learning are segregated - we first interact for many steps to form a batch of experience, and then we perform many (value) parameter updates.
4. In DQN, we sample mini-batches from our replay buffer to train the value network. In A2C, we do not use a replay buffer, and instead we sample mini-batches from a batch of experiences.

Training the value network

The value network can be trained using the same basic method that we used to train the value networks in DQN - by performing gradient *descent* on a **mean squared error (MSE) loss**. In A2C, we learn a state-value function $V(s;\phi) \approx V_\pi(s)$. The parameters of this network are update by performing gradient descent steps on the MSE loss,

$$\mathcal{L}(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(V(s_i;\phi) - y_i\right)^2 \tag{16}$$

The sum is over a **mini-batch** of N data points taken from the full batch. The regression targets y_i are formed using the same basic method we used in DQN:

$$\begin{aligned} y_i &= V_\pi(s_i) && \text{What we want} \\ &= \mathbb{E}_\pi\left[r_i + \gamma V_\pi(s_i')|s_i\right] && \text{Eq. (5)} \\ &\approx r_i + \gamma V_\pi(s_i') && \text{Expectation } \rightarrow \text{ Sample} \\ &\approx r_i + \gamma(1-d_i)V(s_i';\phi^-) && \text{Replace } V_\pi(s_i') \text{ with } (1-d_i)V(s_i';\phi^-) \end{aligned} \tag{17}$$

Once again, ϕ^- is a *lagged* set of parameters used by a **target network**. After training on this batch of transitions, we **synchronise** the target network parameters, $\phi^- \leftarrow \phi$.

Advantage estimation

The value network is used to form **advantage** estimates. The advantage of a state-action pair is defined by

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{18}$$

The advantage function measures the value of an action a taken in state s compared to the value of the state itself. A *positive* advantage indicates that an action is *better* than the average action in that state, and a *negative* advantage indicates that it is *worse*.

We can play the same game to approximate the advantage of a transition (s, a, r, s', d) using our learned value function:

$$\begin{aligned} A_\pi(s,a) &= Q_\pi(s,a) - V_\pi(s) && \text{What we want} \\ &= \mathbb{E}\left[r + \gamma V_\pi(s')|s,a\right] - V_\pi(s) && \text{Eq. (9)} \\ &\approx r + \gamma V_\pi(s') - V_\pi(s) && \text{Expectation } \rightarrow \text{ Sample} \\ &\approx r + \gamma(1-d)V(s';\phi) - V(s;\phi) && \text{Replace } V_\pi(s) \text{ with } (1-d)V(s;\phi) \end{aligned} \tag{19}$$

Training the policy network

The objective for A2C is derived from **the (stochastic) policy gradient theorem**. The basic idea is to *increases* the probability of actions with *positive* advantage and *decreases* the probability of actions with *negative* advantage. The A2C policy objective is

$$J(\theta) = \frac{1}{M}\sum_{i=1}^{M} A_i \ln\left(\pi(a_i|s_i;\theta)\right) \tag{20}$$

Where A_i is the **advantage** estimate for the transition $(s_i, a_i, r_i, s_i', d_i)$, formed using Eq. (19). Note that the sum is over the entire batch, as opposed to a mini-batch. We perform only a single gradient step (per batch of experience) on this objective to update the policy network.

Important: We do not back-propagate through the advantage estimates. They are treated as constants in the policy objective.