DQN

Edward Young - ey245@cam.ac.uk

Quick facts

1. DQN is an *action-value method*.
2. DQN is *off-policy*.
3. DQN is *model-free*.
4. DQN is applicable to any kind of state space but only *discrete* action spaces.

Recap: Optimal action values

- The **optimal action value function** $Q^*(S, A)$ tells us the value of being in state S and performing action A, provided we then follow the optimal policy μ^*.
- In each state, the **optimal policy** μ^* chooses the action which maximises the optimal action value function.
- Mathematically, this can be expressed as

$$\mu^*(S) = \arg\max_A Q^*(S, A) \tag{1}$$

- The optimal action value function satisfies the *Bellman optimality equation*:

$$Q^*(S, A) = R(S, A) + \gamma \sum_{S' \in \mathcal{S}} p(S'|S, A) \max_{A'} Q^*(S', A') \tag{2}$$

Trick: Q-learning

Equation (1) suggests the following strategy for getting good behaviour:

1. Use a neural network called the **value network**, $Q(S, A; \phi)$, to learn an approximation to Q^*, *i.e.*, $Q(S, A; \phi) \approx Q^*(S, A)$ from data. The value network takes in states and outputs an estimate of the value of each action in that state.
2. Motivated by (1), select actions to maximise our approximation, *i.e.*, pursue the policy μ given by

$$\mu(S) = \arg\max_A Q(S, A; \phi) \tag{3}$$

The idea of learning an approximation to the optimal action value function is called **Q-learning**

Trick: Bellman consistency

How can we train the value network $Q(S, A; \phi)$ to approximate the optimal value function? Equation (2) is both *necessary* and *sufficient* for Q^*. We therefore train $Q(S, A; \phi)$ to satisfy this equation, by minimising a **loss function** called the **mean squared Bellman error**:

$$\mathcal{L}(\phi) = \frac{1}{N} \sum_{i=1}^{N} \left(Q(S_i, A_i; \phi) - y_i\right)^2 \tag{4}$$

where

$$y_i = R_i + \gamma \sum_{S' \in \mathcal{S}} p(S'|S_i, A_i) \max_{A'} Q(S', A'; \phi) \tag{5}$$

You may recognise this as the *Mean Squared Error* for a regression problem.

Trick: Sampling

We will make two modifications to this objective. Firstly, forming the regression targets y_i requires taking an expectation over next states S'. This requires a distributional model of the environmental dynamics. Instead, we will replace expectation over S' by a **sample** of the next state, which can be gathered by interactions with the environment:

$$y_i = R_i + \gamma \max_{A'} Q(S'_i, A'; \phi) \text{ where } S'_i \sim p(S'|S_i, A_i) \tag{6}$$

Trick: Target network

Secondly, the regression targets y_i at present depends on the value network. This means that the regression targets themselves move around as we update the parameters. This can lead to instability in learning. Instead, we 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^-.

$$y_i = R_i + \gamma \max_{A'} Q(S'_i, A'; \phi^-) \tag{7}$$

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

$$\phi^- \leftarrow \phi \tag{8}$$

Trick: Experience Replay buffer

We now have a loss function that we can use to train our network. This loss function makes use of a **batch** of N state-action-reward-state transitions S_i, A_i, R_i, S'_i. These are taken from a **replay buffer**. This is a collection of $M \gg N$ "memories" of such transitions taken from interacting with the environment.

Trick: ϵ-greedy policy

When the agent interacts with the environment, it must do so according to a policy. We have already reasoned that at **test time** we should use equation (3) to select actions. However, when gathering experience during **training time**, we want to generate an *diversity* of experiences. We therefore select actions according to an **ϵ-greedy policy**. Given a state S, with probability $1 - \epsilon$ we select the action A that maximises $Q(S, A; \phi)$. With probability ϵ, we randomly select an action. The ϵ-greedy policy provides a trade-off between **exploration** of the environment for potentially new promising strategies and **exploitation** of known good strategies.

Conclusion: 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 experience using an **ϵ-greedy policy** and load that experience into the **replay buffer**.
2. Every 10 interaction steps, randomly sample N transitions from the replay buffer. Form **regression targets** using equation (7) and minimise the **loss** given equation (4).
3. Every 1000 gradient steps, **synchronise** the target network parameters by equation (8).

Typically we take the **batch size** $N = 32$ and the **memory capacity** of the replay buffer to be $M = 10000$ transitions. All of the numbers in this section can be played with to increase performance!