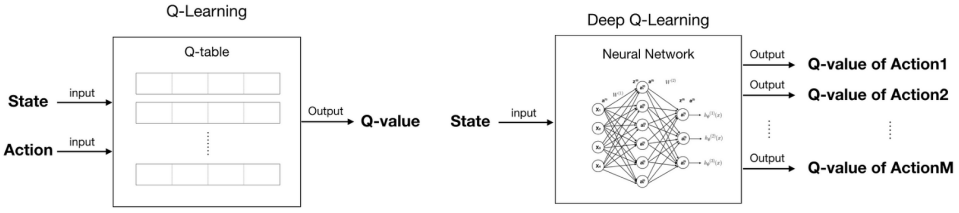
ECE433/COS435 Spring 2024 Introduction to RL

Lecture 10: Deep Q Network

1 Deep Q-Learning

For most problems, it is impractical to represent the Q-function as a table containing values for each combination of s and a. Further, tabular Q learning is slow because it updates Q values one-by-one, where each state transition (s, a, r, s') only provides information about a single Q value $Q(s, a)$.

Instead, we train a function approximator, such as a neural network with parameters θ, to estimate the Q-values, i.e. $Q_\theta(s, a) \approx Q^*(s, a)$. By using the neural network, we have transformed the problem of estimating individual state-action values to approximating the full Q function over a parameter space. This improves generalizability of RL, allowing us to learn information about θ from all transitions.

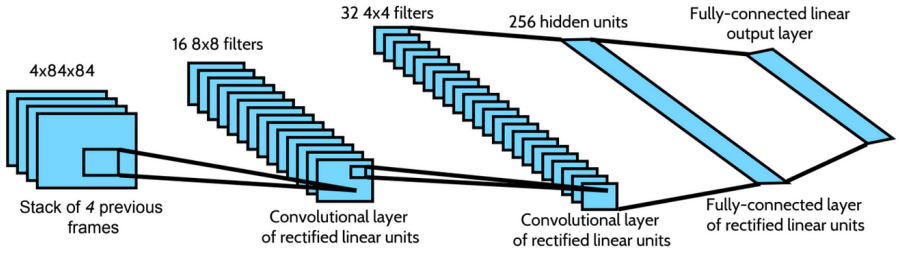


1.1 DQN Step by Step

Q network architecture

The DQN (Deep Q-Network) algorithm was developed by DeepMind in 2015 [1,2]. It was able to solve a wide range of Atari games (some to superhuman level) by combining tabular Q learning and deep neural networks.

In the DQN paper, the network takes as input the stack of 4 previous frames (viewed as state s), process them three two convolutional layers and one full connected layer. The output is a vector of dimension A, corresponding to $Q(s,a)$ for each a in the action space.



Q network architecture

Training the Q network can be done by minimizing the following loss when given a sample state transition (s,a,s',r) :

$$\ell(\theta) = (y - Q_\theta(s,a))^2 \text{ where } y = r + \gamma \max_{a'} Q_{target}(s',a')$$

Here, y_i is called the TD (temporal difference) target, and $y_i - Q_\theta(s,a)$ is called the TD error. Q_{target} is a copy of the Q network with frozen weight, and we may update the target Q network using the current weights every a number of batch updates:

$$Q_{target} \leftarrow Q_\theta, \qquad \text{every fixed number of batch updates}$$

Using Q_{target} to compute the target helps to stabilize the learning processes.

Note that Q-Learning is an **off-policy algorithm**. Q learning update can work with any transition samples (s,a,s',r), regardless of the policy being used. Q-learning converges to the optimal Q function and can find the optimal policy. It is not used for evaluating the value of a fixed policy.

Exploitation-exploration

The overall goal of Q-learning is to find the optimal policy such that $a = \max_a Q^*(s,a)$. However, greedily choosing actions according to Q_θ may result in lack of exploration and getting stock at suboptimal actions (similar to the MAB examples in Lectures 1,2).

The off-policy nature of Q learning allows us to use a different behaviour policy for acting in the environment/collecting data. This behaviour policy is usually an ϵ-greedy policy that selects the greedy action $a = \max_a Q_\theta(s, a)$ with probability $1 - \epsilon$ and a random action with probability ϵ to ensure good coverage of the state-action space.

Experience Replay

The Atari DQN work [1,2] introduced a technique called Experience Replay to make the network updates more stable. At each time step of data collection, the transitions are added to a circular buffer called the replay buffer. Then during training, instead of using just the latest transition to compute the loss and its gradient, we compute them using a mini-batch of transitions sampled from the replay buffer (according to a uniform distribution in the most basic version of the method). This has two advantages: better data efficiency by reusing each transition in many updates, and better stability using uncorrelated transitions in a batch.

An improved method, known as **Prioritized Experience Replay** [3], associates each sample transition (s, a, r, s') with a TD difference $\delta = r + \max_{s'} Q(s', a') - Q(s, a)$. It uses importance sampling to increase the probability of getting a transition sample with higher TD difference.

1.2 Full algorithm of DQN (from the original paper [1])

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

1.3 Results of Impacts of DNQ

Prior to DQN, traditional RL struggled with high-dimensional state spaces and was mostly limited to domains where handcrafted features or representations were available. The DQN algorithm achieved human-level performance on several Atari 2600 games, a milestone in demonstrating the potential of integrating deep learning with RL. The introduction of DQN by DeepMind marked the convergence of deep learning and reinforcement learning.

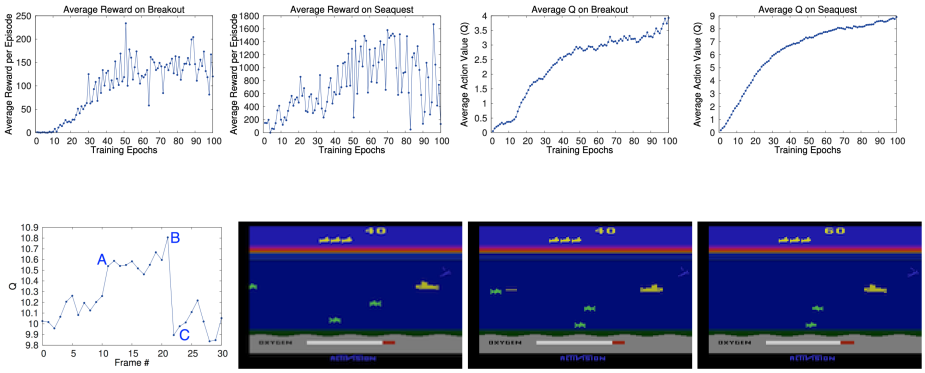


Figure above shows a visualization of the learned value function on the game Seaquest. The figure shows that the predicted value jumps after an enemy appears on the left of the screen (point A). The agent then fires a torpedo at the enemy and the predicted value peaks as the torpedo is about to hit the enemy (point B). Finally, the value falls to roughly its original value after the enemy disappears (point C). Figure 3 demonstrates that our method is able to learn how the value function evolves for a reasonably complex sequence of events.

2 Double DQN

Q learning (both tabular Q learning and DQN) suffer from the **maximization bias**. Specifically, the Q update involves the following term

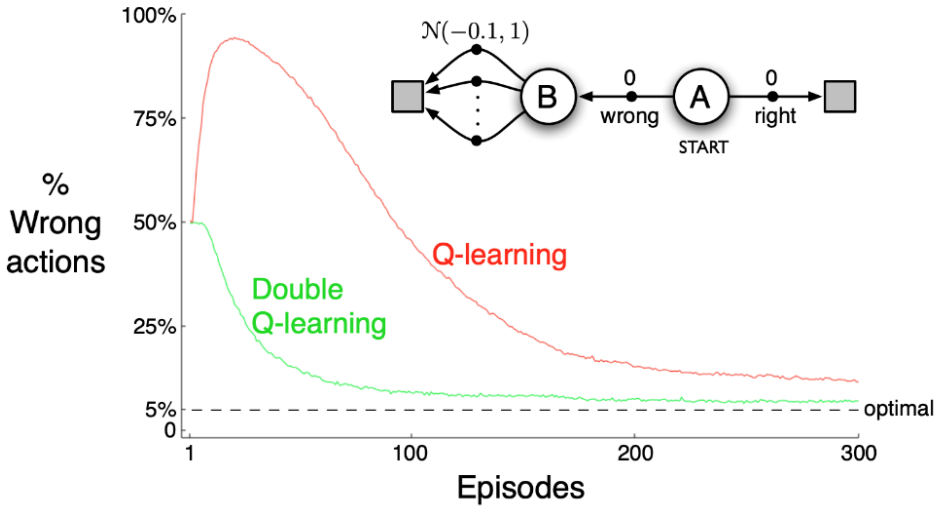
$$\max_{a'} Q^* \left(s', a'\right)$$

Taking the maximal overestimated values introduces a systematic overestimation bias. And since Q-learning involves bootstrapping - learning estimates from estimates - such overestimation can be problematic.

Example of maximization bias

Consider a simple two-state example (see figure below). The agent has to choose an action from the start state A. The optimal action is to go right with a reward 0. However, there are many suboptimal actions from state B that would take the agent to go left, and each of these wrong actions generates a random reward from the Gaussian distribution $N(-0.1, 1)$. Due to the high variance of this Gaussian distributions, many of those wrong actions may end up generating positive rewards in the first few trials, leading to $Q(B, a) > 0$ for many a's, in the initial period of learning. Thus, it is very likely that $\max_a Q(B, a) > 0$. Thus the greedy choice of action by $\text{argmax}_a Q(B, a)$ is very likely to be a wrong action.

This becomes worse in the case of Q learning: The $\text{argmax}_{a'} Q(B, a')$ tends to pick an overly estimated Q value, then this overly estimated $\max_{a'} Q(B, a')$ will be *reinforced* when it is used to update $Q(A, wrong)$, leading to $Q(A, wrong) > 0 = Q(A, right)$. The agent has to rule out every such wrong action for state B, before it can learn the optimal action at A, resulting in slow convergence.



Double DQN

To fix the problem, a clever modification is to use two separate Q-value estimators, instead of one, and have them update each other. Using these independent estimators, we can unbiased Q-value estimates of the actions

selected using the opposite estimator [4]. We can thus avoid maximization bias by disentangling our updates from biased estimates.

Choose (e.g. random) either UPDATE(A) or UPDATE(B)

- if UPDATE(A) then
  – Define $a^* = \arg\max_a Q^A(s', a)$
  — $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s', a^*) - Q^A(s,a)\right)$
- else if UPDATE(B) then
  – Define $b^* = \arg\max_a Q^B(s', a)$
  – $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)\left(r + \gamma Q^A(s', b^*) - Q^B(s,a)\right)$

In the above algorithm, although the action obtained by $\text{argmax}_{a'} Q_A(s', a')$ would be biased towards maximal value of Q_A, this bias would not be reinforced because it will be used to updates Q_A, using the term $Q^B\left(s', \arg\max_a Q^A(s', a)\right)$. Thus the argmax and the max value used for Q updates are decorrelated, which substantially reduces the effect of maximization bias on Q learning. See figures from [4] below for illustration.

Figure: Double DQN reduces the maximization bias in value estimates, compared to DQN

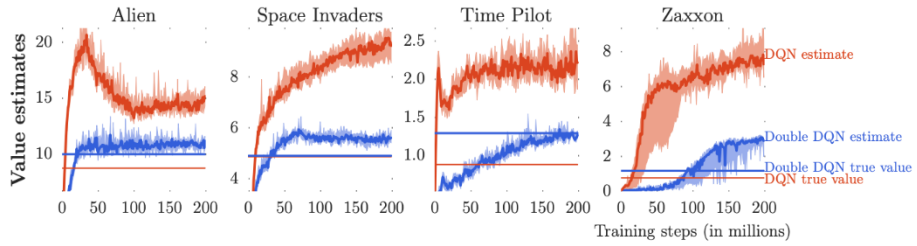
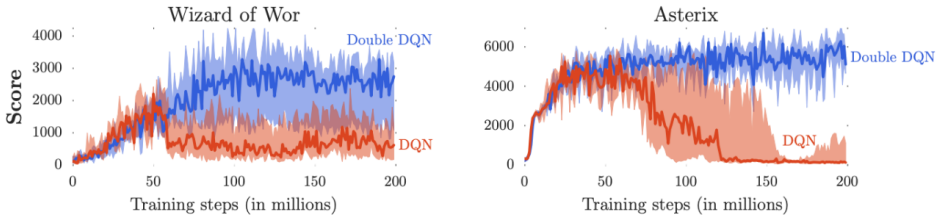


Figure: Double DQN solves RL faster than DQN



[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
DeepMind's 2015 Nature Paper on Human Control RL:
[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
Prioritized Replay Paper:
[3] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay. arXiv preprint arXiv:1511.05952.
[4] Van Hasselt, H., Guez, A., & Silver, D. (2015). Deep reinforcement learning with double Q-learning. In Thirtieth AAAI conference on artificial intelligence.

3 Policy-based vs. Value-based Methods

Q learning method belongs to **value-based RL**. They make direct updates on value functions or Q functions, and only read out policy from the Q function, i.e., $\pi(s) = \text{argmax}Q(s, a)$.

Policy gradient method belongs to **policy-based RL**. They make direct updates on policies, without keeping a value function.

Both methods are model-free: they don't need to model the environment.

Pros/cons of policy gradient

- Pro: unbiased estimate of gradient of expected return
- Pro: can handle a large space of actions (since you only need to sample one and you can use policy parametrization)
- Con: on-policy only, high variance updates (implies poor sample efficiency)

- Con: doesn't do credit assignment

Pros/cons of Q-learning

- Pro: lower variance updates, more sample efficient
- Pro: use off-policy data
- Con: no parametrization, hard to handle many states and actions (since you need to take the max)

Policy gradient and Q learning give rise to more complex, sample efficient algorithms in value-based RL and policy-based RL, respectively. Later in the course we will also learn about their hybrid: actor-critic, and more advanced model-based methods.