The City College of New York

CSC 36000:
Modern Distributed Computing *with AI Agents*

By Saptarashmi Bandyopadhyay
Email: sbandyopadhyay@ccny.cuny.edu
Assistant Professor of Computer Science
City College of New York and Graduate Center at City University of New York

November 3, 2025 CSC 36000

Today's Lecture

Deep Q-Learning

- Refresher: Q-Tables
- Limitations of Tabular Q-Learning
- Deep Q-Learning
- Real-World Use Case: Atari
- Example: DQN Training

Alternative Deep RL Paradigms

- Direct Policy Optimization
- REINFORCE
- Actor-Critic

Distributed Deep RL

Recall: Q-Learning

A **Q-value** is how valuable it is to take a particular **action** at a particular **state**, and usually consists of the two paired together like (s, a)

Q-learning is the process of learning Q-values for different state-action pairs, so that we can eventually make the best decisions by taking the actions with the highest Q-values!

Previously, we were storing our Q-values in *tabular form*, like in this picture:

What could be a limitation of this approach?

states	actions			
	↑	↓	←	→
(1,1)	0	0	0	0
(1,2)	0	0	0	0
...	...	...	...	...
(m,n)	0	0	0	0

The Curse of Dimensionality

With a small number of actions and states, a Q-Table is no problem. But as our environment grows more complex, the Q-Table grows *exponentially* larger!

Example:

- Atari Games like Breakout have an 84×84 screen
- Each pixel can have one of 256 colors
- Therefore the number of possible states is $256^{(84 \times 84)}$
- This is larger than the number of atoms in the universe!

This explosion in complexity is very common in broader machine learning and is known as the *curse of dimensionality*.

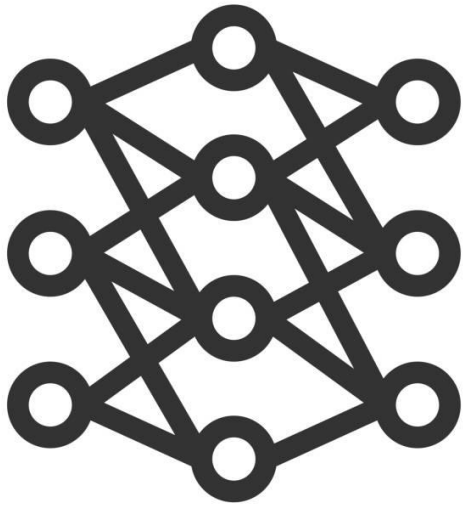
Solution: Deep Q Networks

Instead of storing every single state-action pair, we need a way to *generalize* to new states.

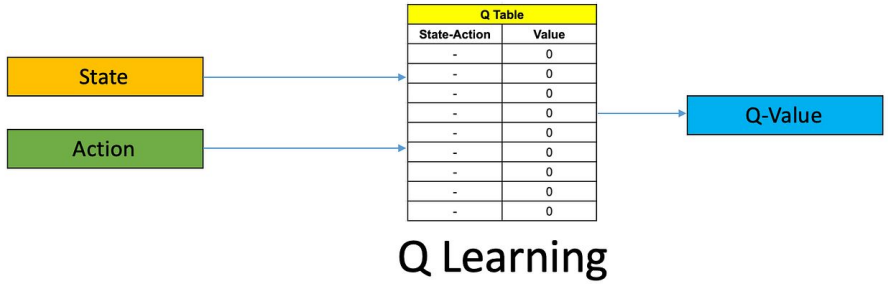
Neural Networks are a natural choice for this due to their status as *universal function approximators*

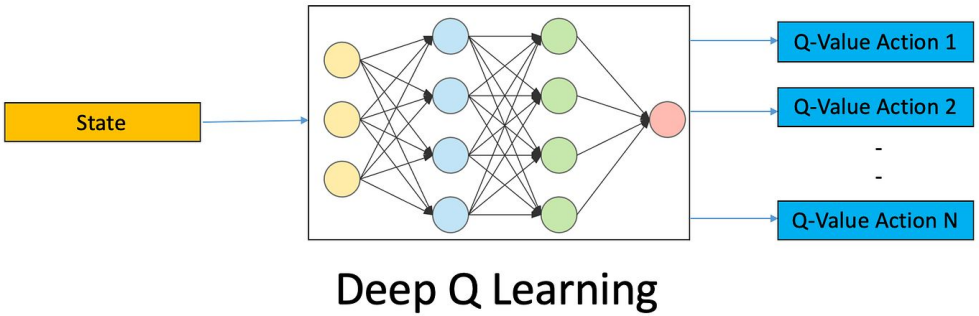
Replace the Q-Table with a Neural Network that takes in our state s and outputs a vector of values; one for each possible action a

This neural network will have parameters θ so our Q-function becomes $Q(s, a; \theta)$



Comparison: Tabular vs. Deep Q-Learning





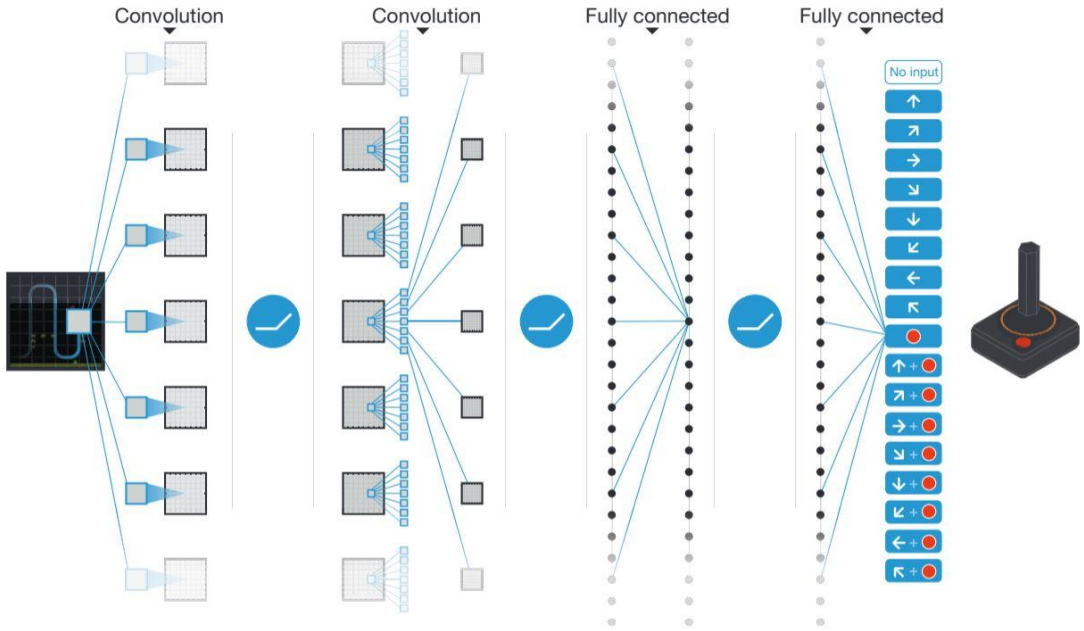
Real-World Use-Case: Deep Q-Learning for Atari Games

Input: The state (last four game frames)

Network:

- Convolutional layers (useful for image data)
- Fully connected layers (normal neural network)

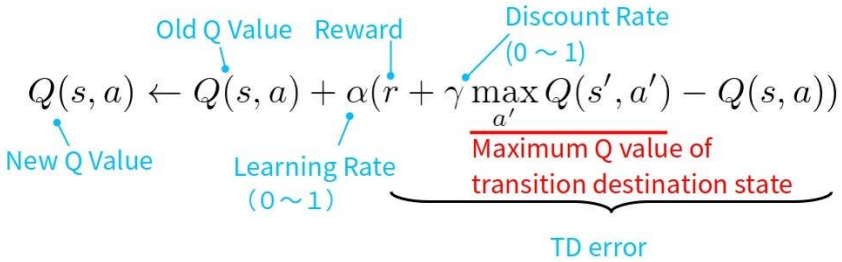
Output: A vector of q-values, one for each action (controller input)



Real-World Use-Case: Deep Q-Learning for Atari Games

Besides Deep Q-Networks, two other key innovations made Deep Q-Learning a Success:

1. **Experience Replay:** Instead of always using the most recent experience, store the agent's experiences in a *replay buffer* and sample them in *mini-batches*
2. **Fixed Q-Targets:** Recall the Bellman equation (below). We use the same Q-function for the current and target Q-value, meaning updates will be a moving target! Instead, we can occasionally save a copy of the main Q-network and use that instead.



Example: DQN Training (Buildup to Distributed DQN)

An agent is in a given state s and executes an action. We have the following information:

- State: s
- Action Taken: $a = \text{'move right'}$
- Reward Received: $r = 10$
- Next State: s'
- Discount Factor: $\gamma = 0.9$

The main network (with weights θ) processes state and predicts the following Q-values:

$$Q(s, \text{'left'}; \theta) = 15$$

$$Q(s, \text{'right'}; \theta) = 20$$

The predicted Q-value for the action that was actually taken is therefore $Q(s, \text{'right'}; \theta) = 20$

Example: DQN Training

To calculate the target for our loss function, we must determine the value of the next state s'

This is done by feeding s' into the separate, fixed target network (with weights θ^-)

The target network with different parameters processes state and predicts the following Q-values:

- $Q(s', \text{'left'}; \theta^-) = 25$
- $Q(s', \text{'right'}; \theta^-) = 30$

s

Example: DQN Training

The target Q-value, often denoted as y, represents the "ground truth" that our main network's prediction should move towards. It is calculated using the Bellman equation with the values from the target network.

The formula for the target is:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

First, find the maximum Q-value for the next state s' from the target network's output:

$$\max_{a'} Q(s', a'; \theta^-) = \max(25, 30) = 30$$

Now, substitute this value in the target Q value formula, along with the reward and discount factor, into the formula:

$$y = 10 + (0.9 \times 30)$$

$$y = 10 + 27 = 37$$

The target Q-value is **37**. This is the value our main network should have predicted for taking the 'move right' action in state .

Example: DQN Training

The final step is to compute the error, or loss, between the target value and the main network's prediction.

DQN typically uses the Mean Squared Error (MSE) for this calculation. The loss formula is:

$$L = (y - Q(s, a; \theta))^2$$

Plugging in our calculated target and the original prediction:

$$L = (37 - 20)^2 = 17^2 = 289$$

This loss value of 289 quantifies the error in the main network's prediction for this specific experience. This value is then backpropagated through the main network to adjust its weights θ .

The update will nudge the network's output for $Q(s, \text{'right'})$ to be closer to 37, thus improving its accuracy for future predictions.

Alternative Distributed Deep RL Paradigms

Limitations of Deep Q-Learning

Deep Q-Learning (DQL) is powerful and flexible, but has two main weaknesses:

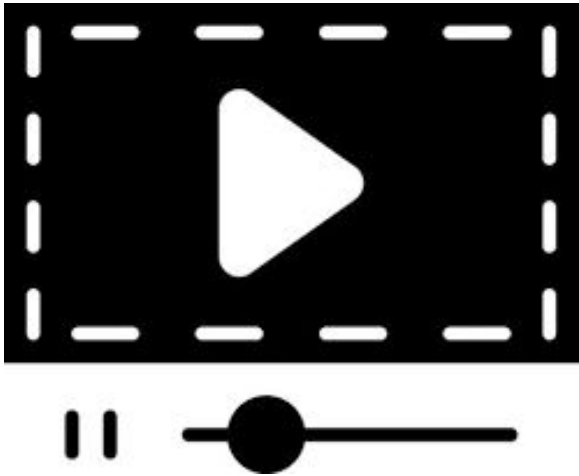
1. **Continuous Action Spaces:** DQL handles small discrete sets of actions (e.g. ‘left’, ‘right’) very well, but many problems, such as self-driving cars, require continuous values as output (e.g. ‘set steering angle to 104.2°’)
2. **Stochastic Policies:** DQL typically learns *deterministic* policies, meaning it *always* chooses the action with the highest predicted Q–value. For many environments, such as rock-paper-scissors, a deterministic policy will fail.

Policy Gradient Methods

Instead of basing our policy around a value function (as in Q-learning), what if we learned our policy directly?

This is the inspiration of *Policy Gradient Methods*, which learn parameterized networks that take states as input and output a *probability distribution* of actions instead of *values* for those actions

Think of a video playback, instead of looking at a frame of every second and then choosing the one closest to the moment you want, you get smoothly scroll and find the right moment!



REINFORCE

The most basic and foundational Policy-Gradient method is REINFORCE.

REINFORCE goes through three main steps:

1. **Act and Observe:** Interact with the environment through the current policy
2. **Evaluate the Outcome:** Calculate the *return* of the policy rollout
3. **Update the Policy:** Adjust the parameters of the policy to make the actions *more likely* if the return was good and *less likely* if if the return was bad

Mathematically: $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi(a_t|s_t;\theta)$

Weaknesses of REINFORCE

REINFORCE does better than Deep Q-Learning for continuous tasks like robotic control and self driving cars, but it suffers from high *variance*.

Our learning signal G_t can be very noisy and unlucky or lucky trajectories can massively influence the entire policy!

Ironically, this was not a problem in Deep Q-Learning.

This motivates the next paradigm, the *Actor-Critic Framework*.

Actor-Critic

Instead of *only* learning a policy or *only* learning a value function, can we combine the best of both worlds? Yes we can!

In Actor-Critic Methods we train two networks:

1. The **Actor** network chooses the action. This is similar to the policy network we used in REINFORCE
2. The **Critic** network evaluates the action chosen by the actor with a value, akin to the Q-Networks we learned in Deep Q Learning

Advantage Actor-Critic

By itself, Actor-Critic isn't much better than REINFORCE. Actions are still judged using the full episode return, which still has lots of variance!

Instead of using the full episodic return, we base our signal on intermediate rewards, giving us a much more nuanced perspective for credit assignment.

Distributed Deep RL

Asynchronous Advantage Actor-Critic (A3C)

How do we extend methods like Advantage Actor Critic into the distributed setting?

We can borrow from the idea of the *Parameter Server* from before, including:

- A **Global Network:** The central neural network that contains for both the Actor and the Critic. It does not interact with the environment itself, but contains the authoritative copy of the parameters.
- **Multiple Worker Agents**: The system spawns multiple processes, each with its own CPU thread, local copy of the Actor-Critic network, and independent copy of the environment

Asynchronous Advantage Actor-Critic (A3C)

Multiple parallel actor-learner threads explore different copies of the environment simultaneously.

Each thread calculates its Q-learning target y using a shared, slow-updating target network (θ-).

Threads compute and accumulate gradients locally based on the difference between the target (y) and the current Q-value Q(s,a;θ)

After a set number of steps, each thread asynchronously updates the shared global network (θ) with its local gradients.

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Questions?