The City College of New York

CSC 59866-E: Senior Project I
AI Agents for Decision Making in the Real World

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

Deep Learning, Reinforcement Learning (RL) & Multi-Agent Deep Reinforcement Learning (MARL): REINFORCE, Independent Q Learning and Convergence

Today's Agenda

Deep Q-Learning Walkthrough

- Predicting Q and Calculating Loss

Walkthrough: Experience Replay

- How much buffer do we need?

DQL vs. Policy Gradients

- Weaknesses of DQL, Policy Gradient Methods, REINFORCE

Sample Complexity, Generalizability, IQL, Actor-Critic, and MADDPG

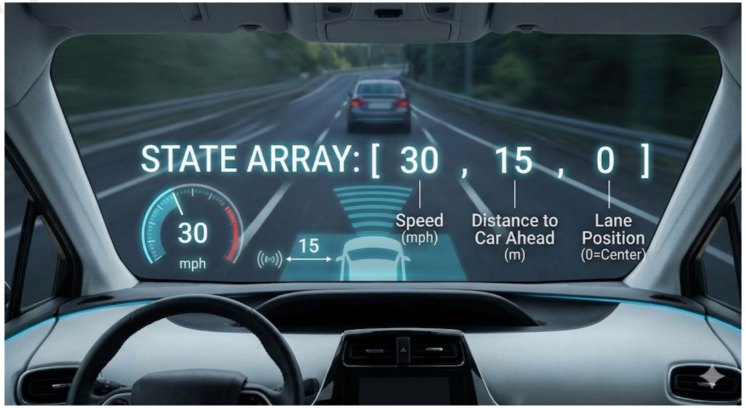
Imitation Learning: Behavioral Cloning, DAgger

Deep Q-Learning Walkthrough

Walkthrough: Predicting Q and Calculating Loss

Step 1: The Input (State)

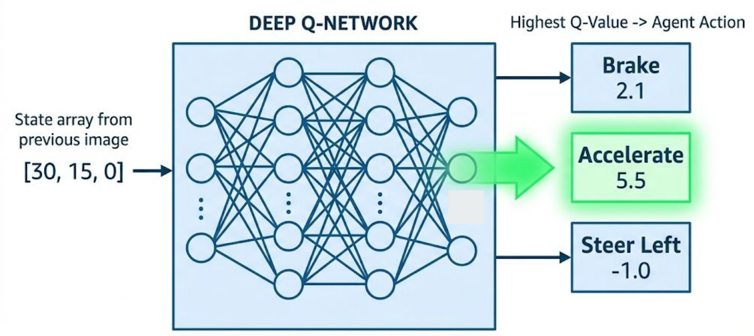
- Instead of looking up a row in a table, our agent observes the environment state as an array of numbers.
- *Example:* A self-driving car observes its state: `[Speed: 30, Distance_to_car: 15, Lane_pos: 0]`



Walkthrough: Predicting Q and Calculating Loss

Step 2: The Forward Pass (Prediction)

- We pass this array through the Neural Network's hidden layers.
- The output layer has exactly one neuron for every possible action.
- *Prediction:* The network outputs estimated Q-values: `[Brake: 2.1, Accelerate: 5.5, Steer_Left: -1.0].`
- *Action:* The agent acts greedily and chooses **Accelerate** (Highest Q-value = 5.5).



Walkthrough: Predicting Q and Calculating Loss

Step 3: The Environment Reaction

- The car accelerates. It gets too close to the car in front!
- It receives a **Reward (*r*) = -10** and observes the **Next State (*s*')**: `[Speed: 35, Distance: 5, Lane_pos: 0]`.

STATE ARRAY: [35 , 5 , 0]
35
mph
COLLISION WARNING
5 m
REWARD (r): -10

Walkthrough: Predicting Q and Calculating Loss

Step 4: The Target Calculation (The Bellman Step)

- How do we know if our prediction of `5.5` was right? We use the Bellman Equation to find the "Target" value.
- We pass the Next State (s') through the network to find the max future Q-value. Let's assume the network predicts the best future action from here has a Q-value of `2.0`.
- *Math* ($\gamma = 0.9$): `Target` $= r + \gamma \max Q(s', a') = -10 + 0.9(2.0) =$ **-8.2**

Walkthrough: Predicting Q and Calculating Loss

Step 5: The Loss & Update

- **The Error:** Our network confidently predicted Accelerating was a great idea (`+5.5`), but the reality (Target) was terrible (`-8.2`).
- **The Loss (MSE):** $Loss = (\text{Target} - \text{Prediction})^2 = (-8.2 - 5.5)^2 = 187.69$
- **The Update:** We use Backpropagation and Gradient Descent to update the network's weights. The next time it sees that state, it will predict a Q-value much closer to `-8.2` and learn to hit the brakes instead!

Walkthrough: Experience Replay

Walkthrough: Experience Replay

How much buffer do we actually need when training these agents?

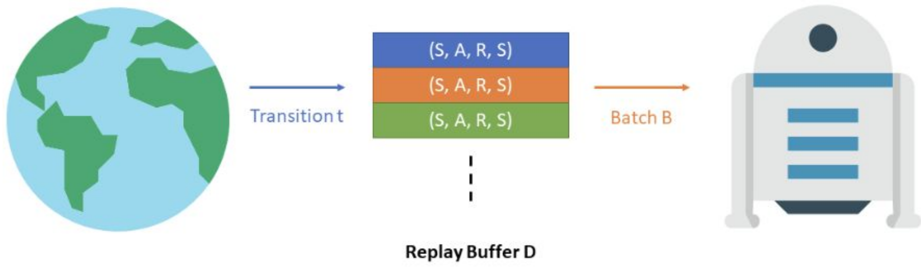
Step 1: The Correlation Problem

- If an agent learns strictly sequentially (State 1 $\rightarrow$ State 2 $\rightarrow$ State 3), the data is highly correlated.
- Neural networks struggle with this; they will overfit to the current sequence and "forget" how to handle past situations (Catastrophic Forgetting).

Walkthrough: Experience Replay

Step 2: Building the Replay Buffer

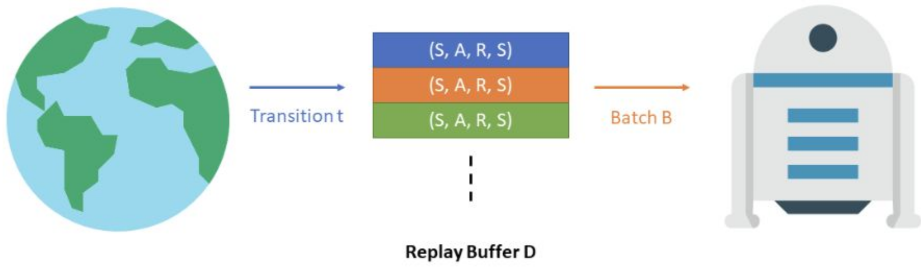
- To fix this, the agent does not learn immediately. Instead, it saves every single interaction, a tuple of $(State, Action, Reward, NextState)$, into a massive database called the **Replay Buffer**.



Walkthrough: Experience Replay

Step 3: The Random Batch

- When it is time to update the network weights, the agent reaches into the buffer and randomly pulls out a "mini-batch" (e.g., 256 scattered memories) to train on. This breaks the time correlation and stabilizes the math!



Walkthrough: Experience Replay

Step 4: The Sizing Dilemma

- **If the buffer is too small:** The agent overwrites old memories too fast. It forgets how to drive in the snow because it has only seen sunny days for the last 10,000 steps.
- **If the buffer is too large:** The agent wastes compute time training on millions of "bad" actions it took days ago when its policy was terrible.

You may need to experiment with the proper size of the replay buffer for your problem!

Walkthrough: Experience Replay

Step 5: The Agentic Solution (PER)

- Modern MARL agents rarely use a simple random buffer. They use **Prioritized Experience Replay (PER)**.
- *The trick:* The agent scores its memories. It intentionally over-samples the memories where its prediction error (Loss) was the highest, forcing the network to learn from its biggest mistakes first rather than wasting time on things it already knows!

DQL vs. Policy Gradients

Limitations of Deep Q-Learning

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

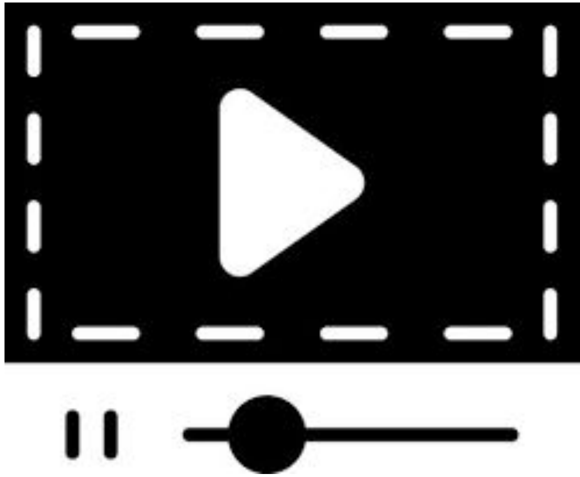
1. **Continuous Action Spaces:** DQL handles small discrete sets of actions (e.g. 'left', 'right') very well, but many problems, such as self-driving cars, require continuous values as output (e.g. 'set steering angle to 104.2°')
2. **Stochastic Policies:** DQL typically learns *deterministic* policies, 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!

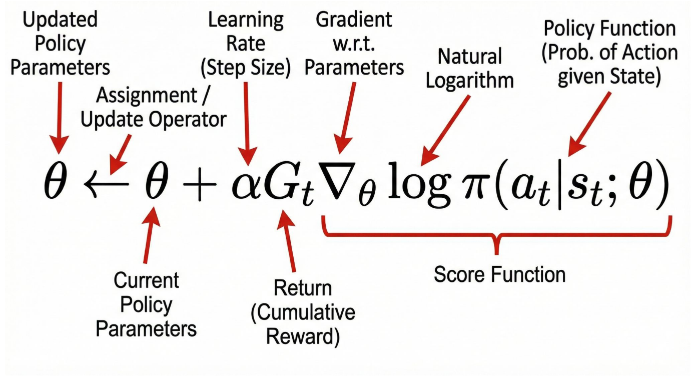


REINFORCE

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


Updated Policy Parameters
Learning Rate (Step Size)
Gradient w.r.t. Parameters
Natural Logarithm
Policy Function (Prob. of Action given State)
Assignment / Update Operator
$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi(a_t|s_t;\theta)$
Current Policy Parameters
Return (Cumulative Reward)
Score Function

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*.

Deep Q vs. Policy Gradients (Summary)

- **Deep Q-Learning (DQN):**
  - *Mechanism:* Learns the *value* of actions (Q-values) and implicitly derives a policy by choosing the max.
  - *Strengths:* Sample efficient (uses Experience Replay to reuse old data).
  - *Weaknesses:* Fails in continuous action spaces. Deterministic.
- **Policy Gradients (REINFORCE):**
  - *Mechanism:* Learns the *policy* directly (outputs probability distributions).
  - *Strengths:* Naturally handles continuous actions and learns stochastic policies.
  - *Weaknesses:* High variance, less sample efficient (on-policy: cannot easily reuse old data).

Questions?