

Lecture 11

Deep Reinforcement Learning

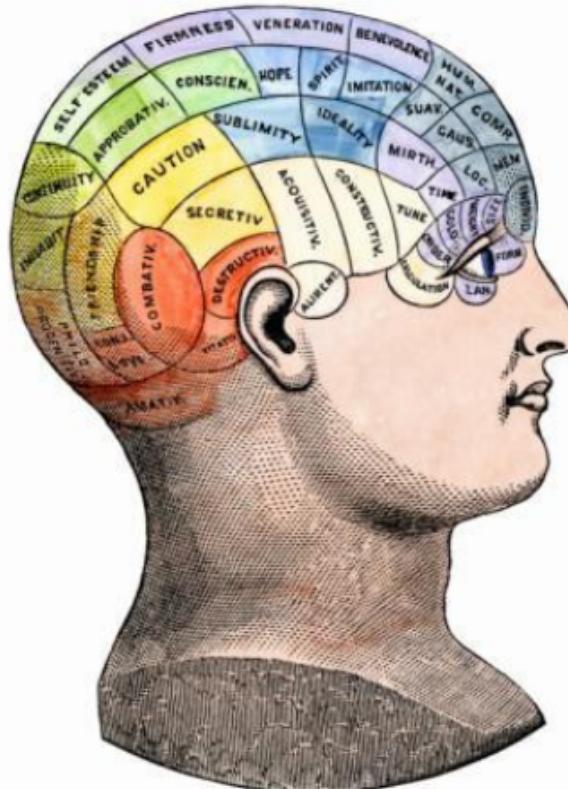
Beyond DQN

jmanero@faculty.ie.edu

MBD-EN2024ELECTIVOS-MBDMCSBT_37E89_467614

Deep Reinforcement Learning

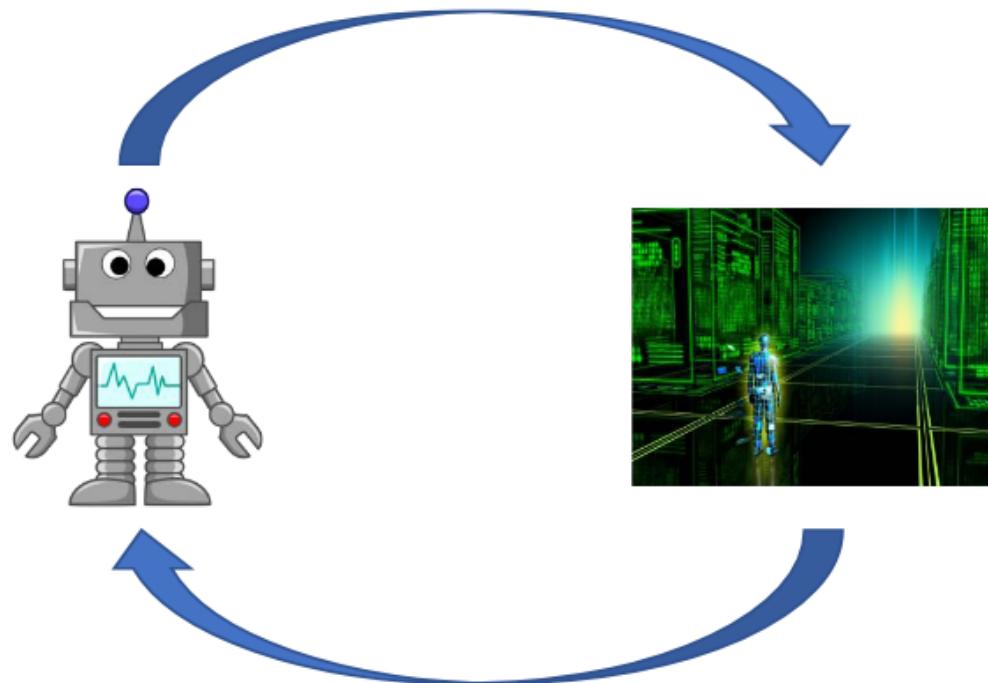
Is about learning



- Reinforcement learning = can reason about decision making
- Deep models = allows RL algorithms to learn and represent complex input-output mappings

Deep models are what allow reinforcement learning algorithms to solve complex problems end to end!

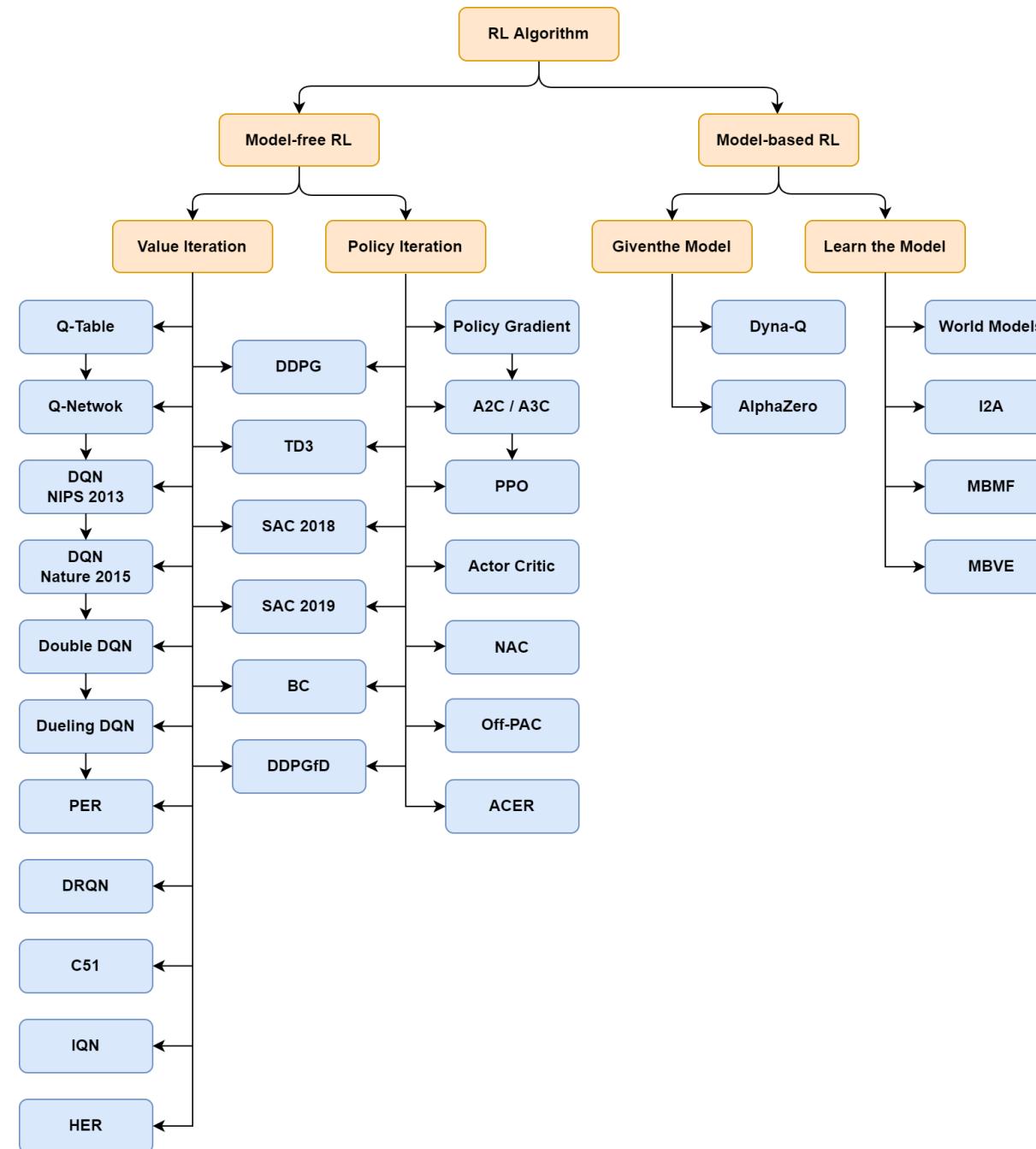
Anything you can control must be simulated



- Provides a powerful engineering tool
- Now *that* different from conventional engineering approach!
 - Before: characterize, simulate, control
 - Now: characterize, simulate, run RL
- Main role: powerful *inversion* engine
- Main weakness: still need to simulate!

DQN-DDQN

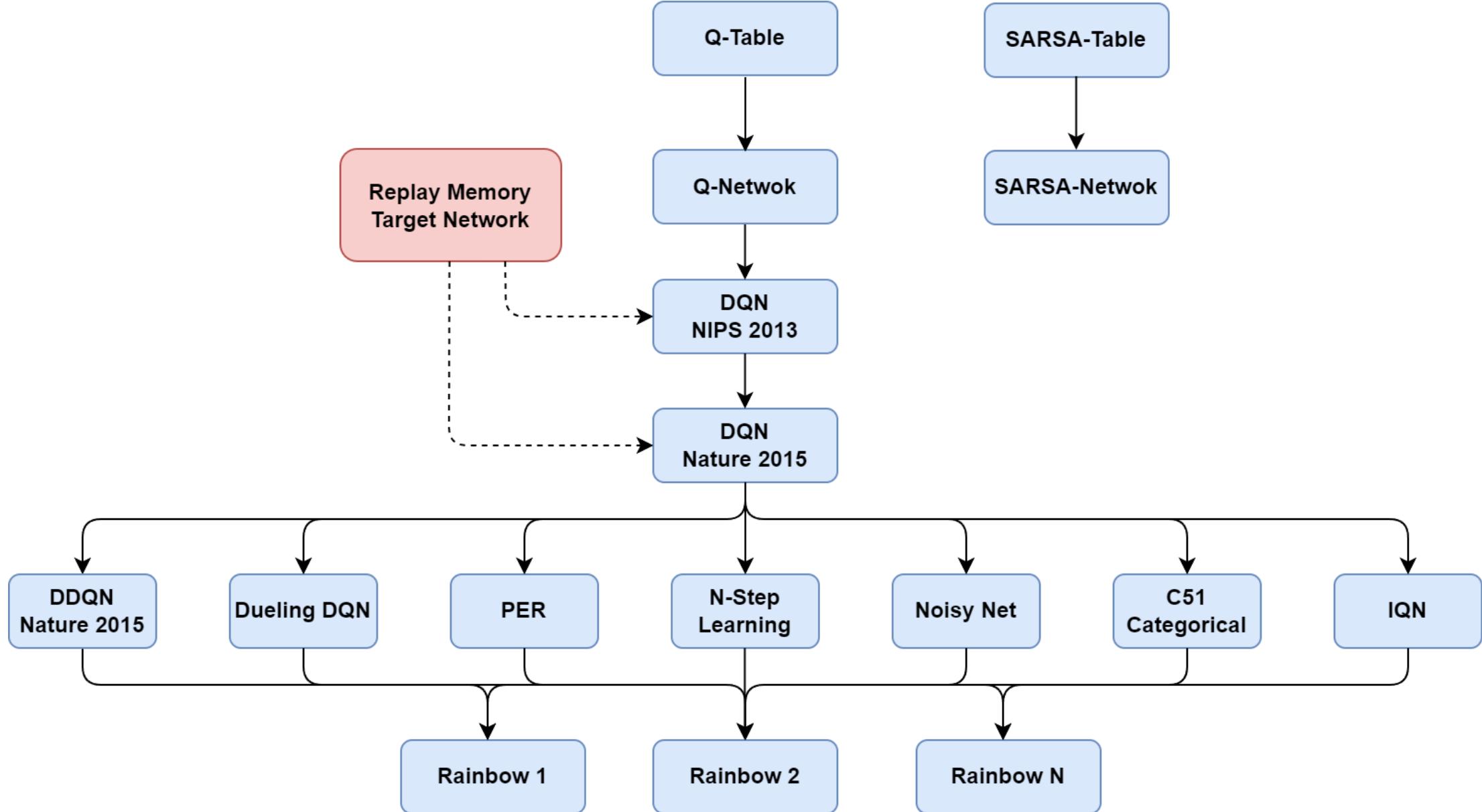
A Taxonomy of models



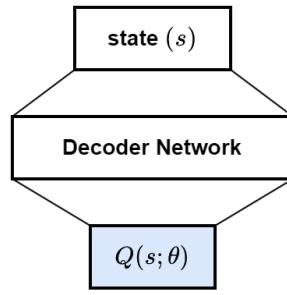
Some conclusions & advice

- Patience. Training takes time, in Lunar Lander your first intents may be close to a day.
- Use a GPU
- Try DQN and DDQN. For complex problems DDQN is better
- Look at the learning rate. Try adaptive learning rate
- Try non-standard exploration if necessary
- Run several seeds (examples) in parallel. Keep Track of your advance
- Use Hyperparameter setting to see how different parameters help to converge

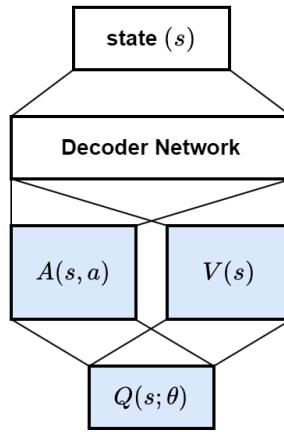
DQN & DDQN Improvements



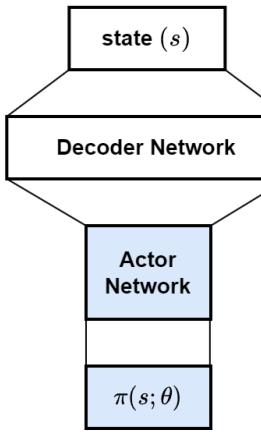
Different Neural Networks



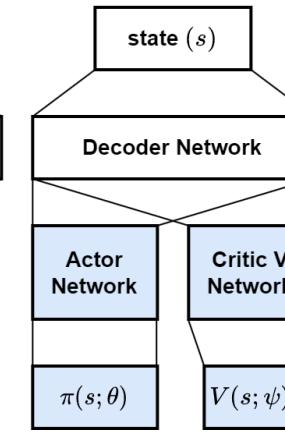
(a) Vanilla DQN



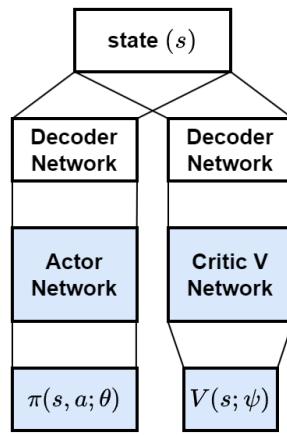
(b) Dueling DQN



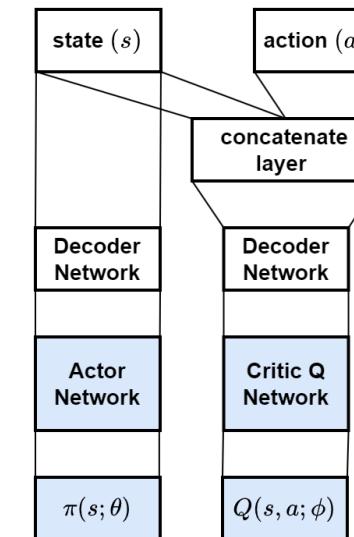
(c) PG / Reinforce



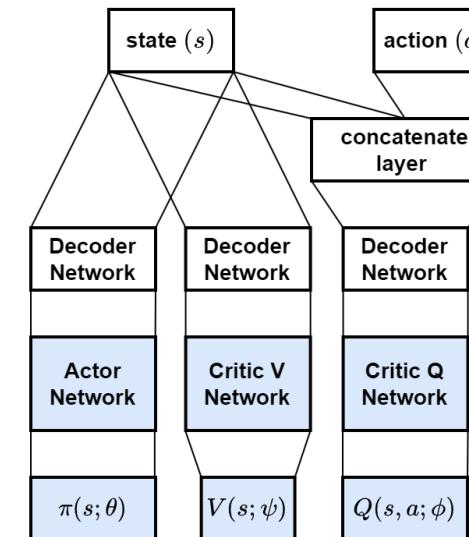
(d) Two-head Actor Critic



(e) Decoupled Actor Critic



(f) DDPG / TD3 / SAC2019

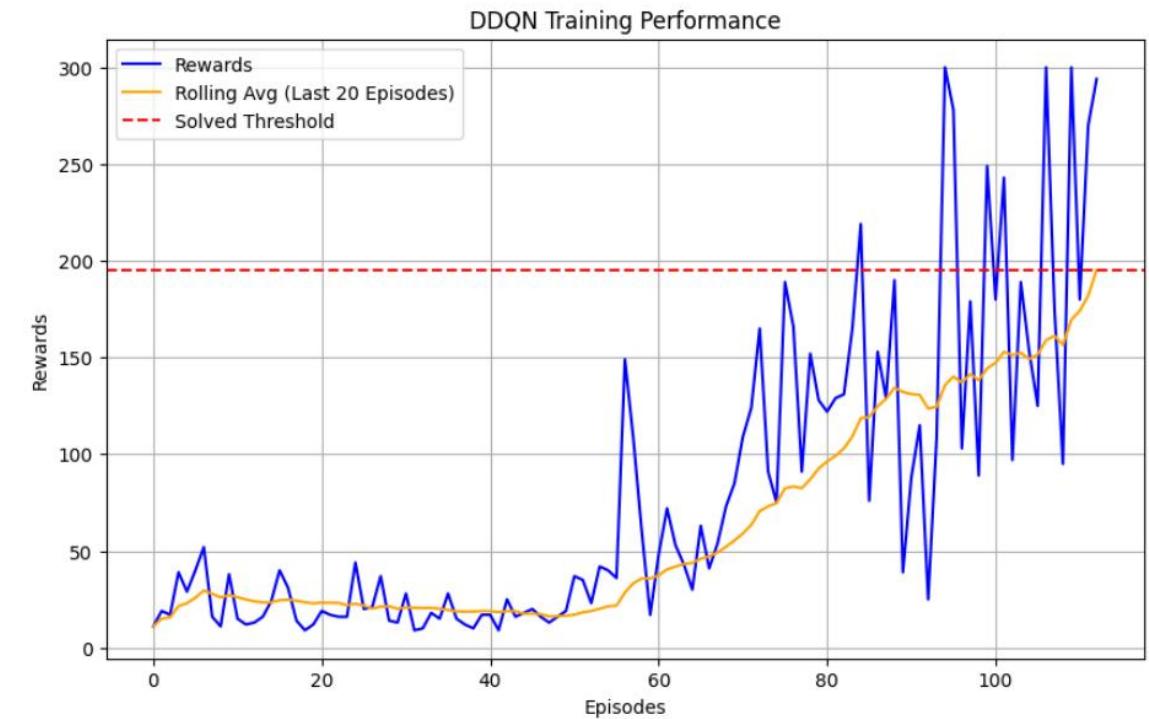
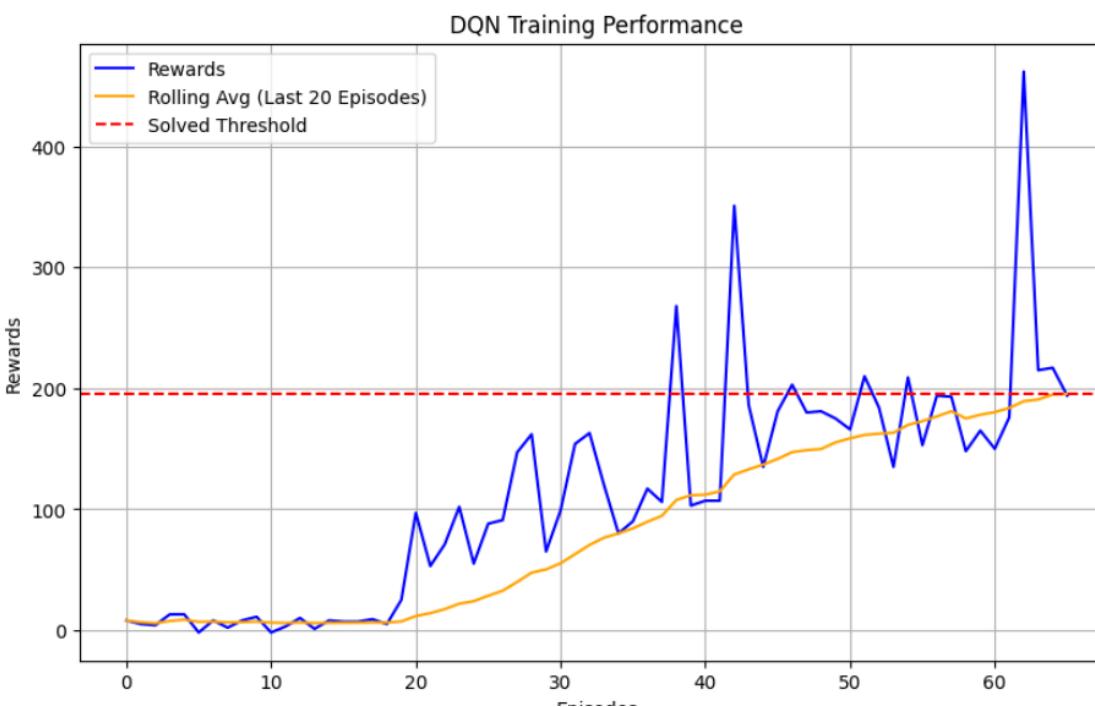


(g) SAC2018

- **Deep Q-Network (DQN)**
- **Problem Addressed:** Instability and inefficiency in traditional Q-learning with neural networks.
- **Key Innovations:**
 - Experience Replay: Breaks correlation between consecutive experiences.
 - Target Networks: Stabilizes training by maintaining a separate target network.
- **Limitations:**
 - Struggles with continuous action spaces.
 - Inefficient exploration (relying heavily on epsilon-greedy).

030

031



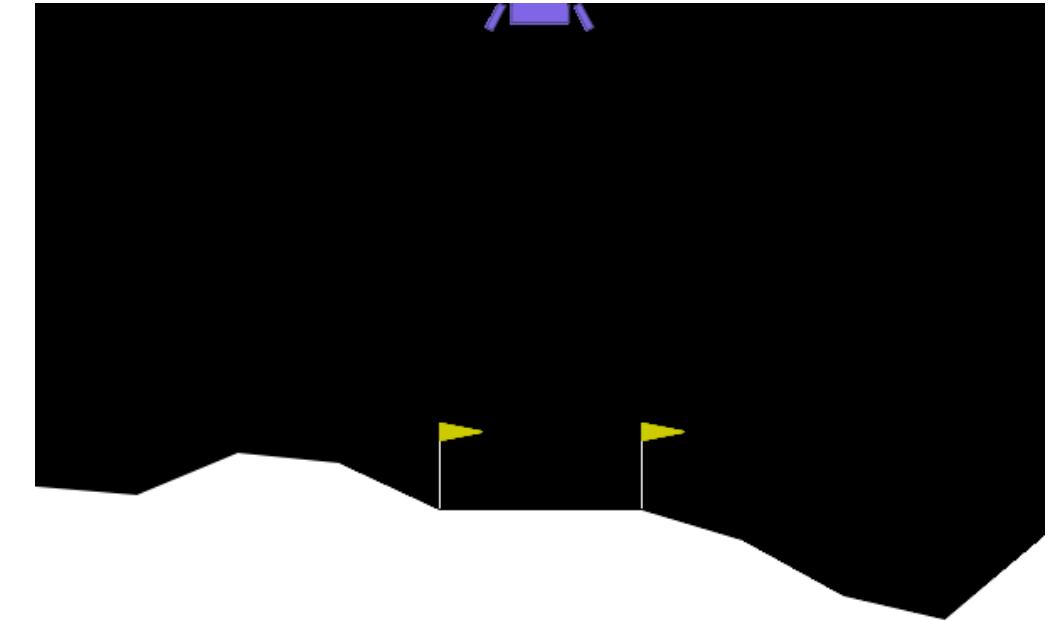
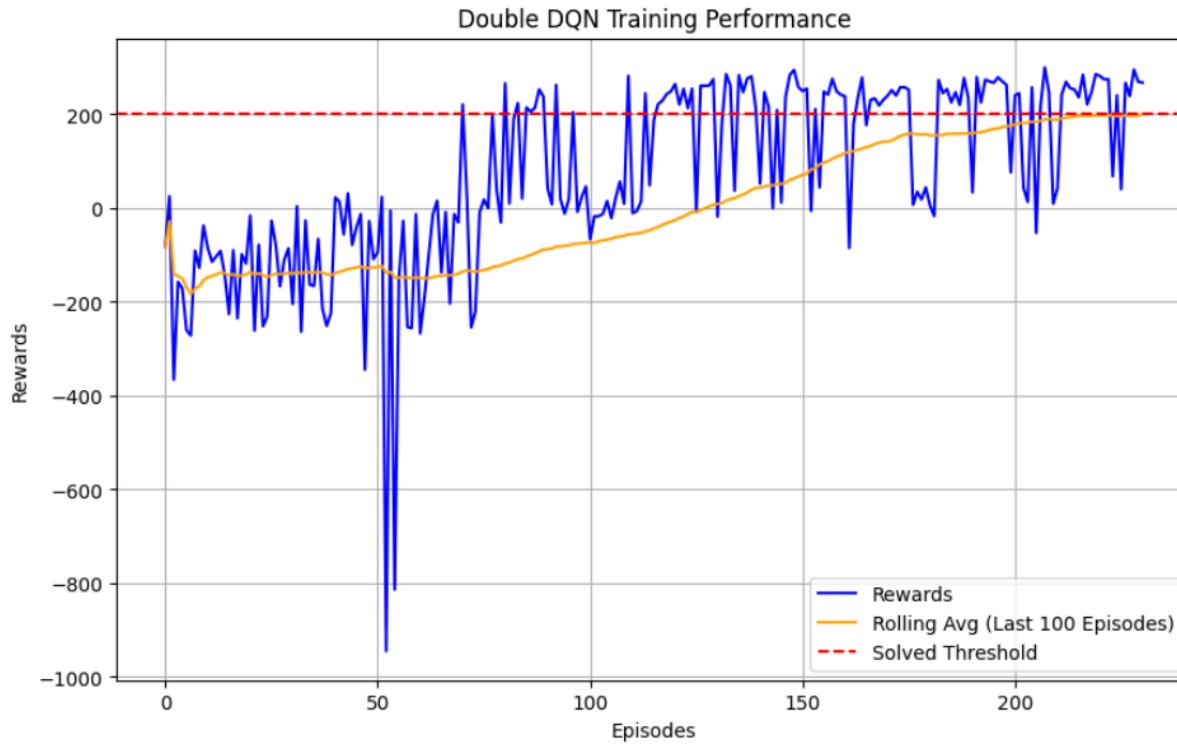
CARTPOLE is a simple environment for Function approximation, we can see the training performance of DQN compared to DDQN

```

21 # Double DQN target calculation
22 def experience_replay_with_ddqn(model, target_model, batch_size, gamma, tau, step):
23     if len(replay_buffer) < batch_size:
24         return
25
26     states, actions, rewards, next_states, dones = sample_experiences(batch_size)
27
28     # Predict Q-values for next states using both networks
29     next_q_values = model.predict(next_states, verbose=0)
30     best_actions = np.argmax(next_q_values, axis=1)
31     target_q_values = target_model.predict(next_states, verbose=0)
32
33     # Update Q-values using Double DQN formula
34     targets = rewards + gamma * target_q_values[np.arange(batch_size), best_actions] * (1 - dones)
35
36     # Update main Q-network
37     q_values = model.predict(states, verbose=0)
38     q_values[np.arange(batch_size), actions] = targets
39     model.fit(states, q_values, epochs=1, verbose=0)
40
41     # Apply soft update to target network
42     if step % retrain_steps == 0:
43         soft_update(model, target_model, tau)

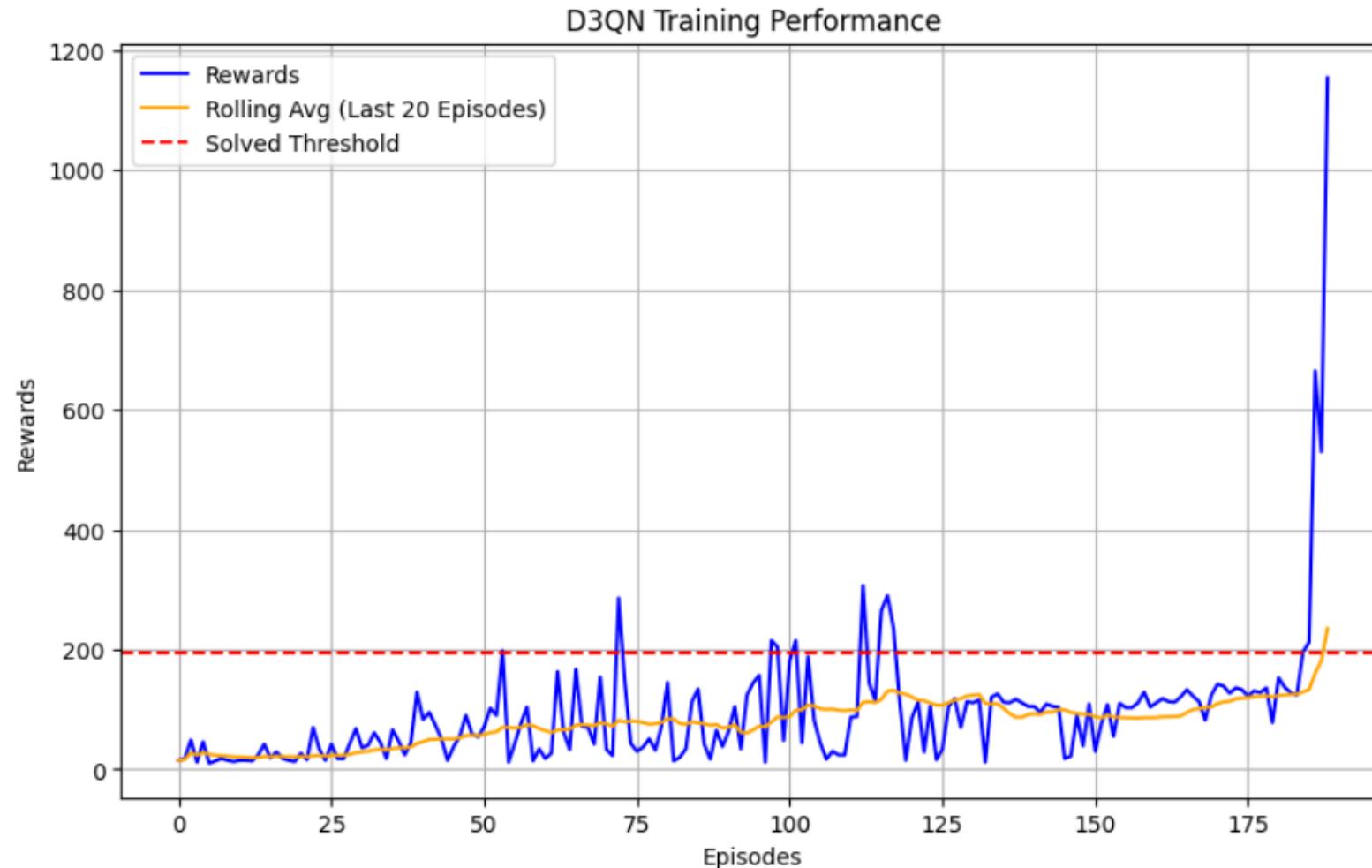
# Soft update function for target network
""" Soft Update using Polyak optimization """
def soft_update(model, target_model, tau):
    target_weights = target_model.get_weights()
    model_weights = model.get_weights()
    new_weights = [
        tau * mw + (1 - tau) * tw for mw, tw in zip(model_weights, target_weights)
    ]
    target_model.set_weights(new_weights)

```



In LUNAR LANDER the DDQN has a much better behavior than DQN as the complexity has increased greatly

- **Dueling DQN**
- **Problem Addressed:** Difficulty in differentiating the impact of specific actions on the state-value.
- **Key Innovations:**
 - Decomposes Q-values into two streams:
 - **State-value function** ($V(s)$): Represents the value of the state.
 - **Advantage function** ($A(s,a)$): Captures the importance of each action.
 - Combines: $Q(s,a) = V(s) + A(s,a)$
- **Result:** Focuses learning on actions that matter, improving efficiency



- **Prioritized Experience Replay**
- **Problem Addressed:** Uniform sampling in DQN is inefficient.
- **Key Innovations:**
 - Samples experiences based on their importance (TD-error).
 - Ensures that rare, high-error transitions are revisited more frequently.
- **Result:** Faster and more focused learning.

Prioritized Experience Replay

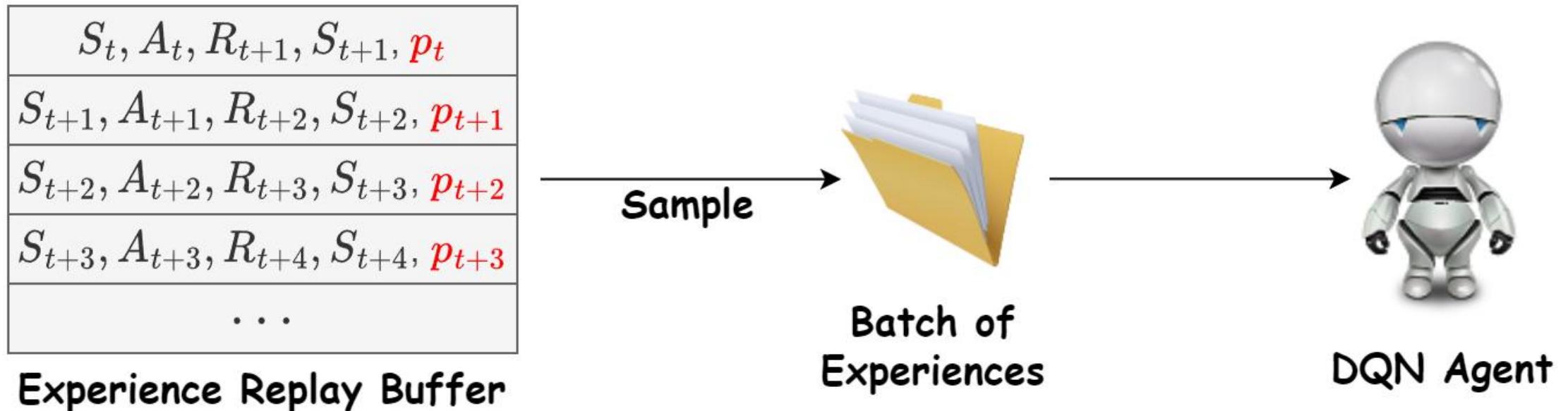
- Prioritized Experience Replay (PER) was introduced in 2015 by [Tom Schaul](#). The idea is that some experiences may be more important than others for our training, but might occur less frequently.
- Because we sample the batch uniformly (selecting the experiences randomly) these rich experiences that occur rarely have practically no chance to be selected.
- That's why, with PER, we try to change the sampling distribution by using a criterion to define the priority of each tuple of experience.
- We want to take in priority experience where there is a big difference between our prediction and the TD target, since it means that we have a lot to learn about it.
- We use the absolute value of the magnitude of our TD error

$$p_t = |\delta_t| + e$$

Magnitude of our TD
error

Constant assures that no
experience has 0 probability
to be taken.

Prioritized Experience Replay



$$P(i) = \frac{p_i}{\sum_k p_k}^a$$

Priority value

Hyperparameter used to reintroduce some randomness in the experience selection for the replay buffer

If $a = 0$, pure uniform randomness

If $a = 1$, only select the experiences with the highest priorities

Normalized by all priority values in Replay Buffer

As consequence, during each time step, we will get a batch of samples with this probability distribution and train our network

Prioritized Experience Replay

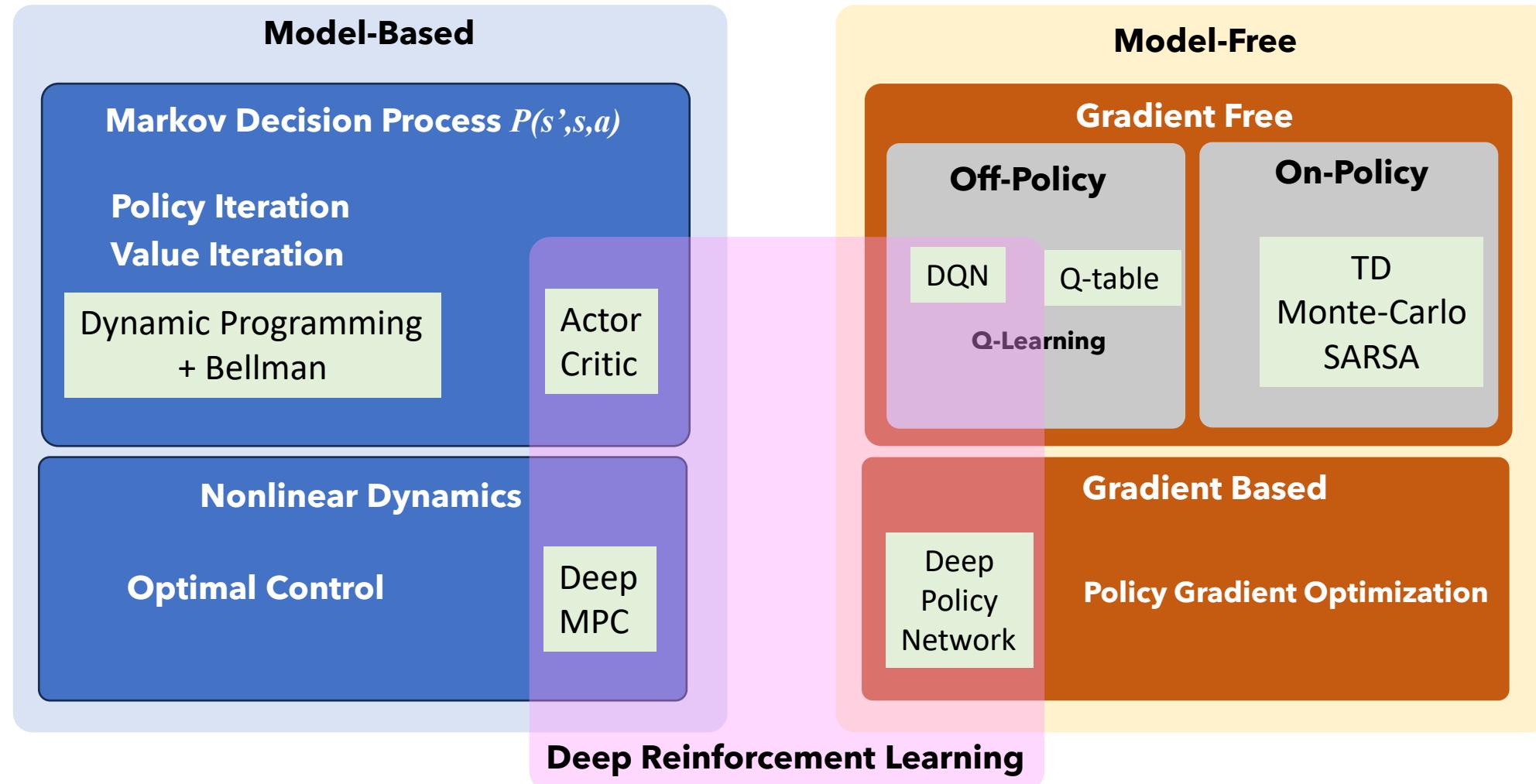
Algorithm 1 Double DQN with proportional prioritization

- 1: **Input:** minibatch k , step-size η , replay period K and size N , exponents α and β , budget T .
- 2: Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
- 3: Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
- 4: **for** $t = 1$ **to** T **do**
- 5: Observe S_t, R_t, γ_t
- 6: Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in \mathcal{H} with maximal priority $p_t = \max_{i < t} p_i$
- 7: **if** $t \equiv 0 \pmod K$ **then**
- 8: **for** $j = 1$ **to** k **do**
- 9: Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
- 10: Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
- 11: Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg \max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
- 12: Update transition priority $p_j \leftarrow |\delta_j|$
- 13: Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
- 14: **end for**
- 15: Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
- 16: From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
- 17: **end if**
- 18: Choose action $A_t \sim \pi_\theta(S_t)$
- 19: **end for**

Policy Gradient Methods

Policy Gradient Methods

Classification of RL Methods

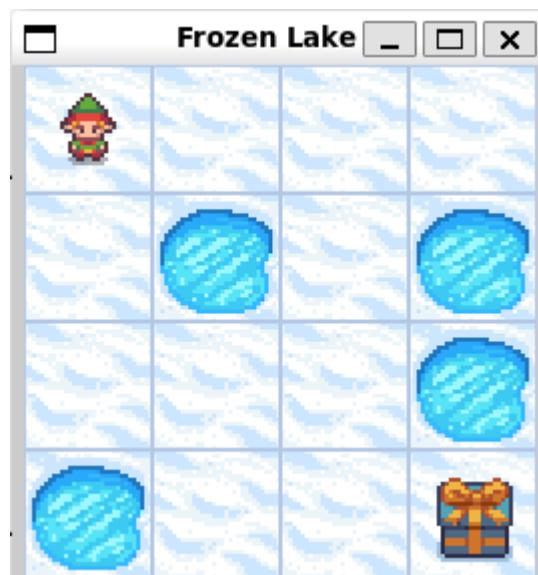


Reinforcement Learning

Value Function

Value Function Format (4x4):

S	0.656	0.729	0.656
0.656	X	0.810	X
0.729	0.810	0.900	X
X	0.900	1.000	G



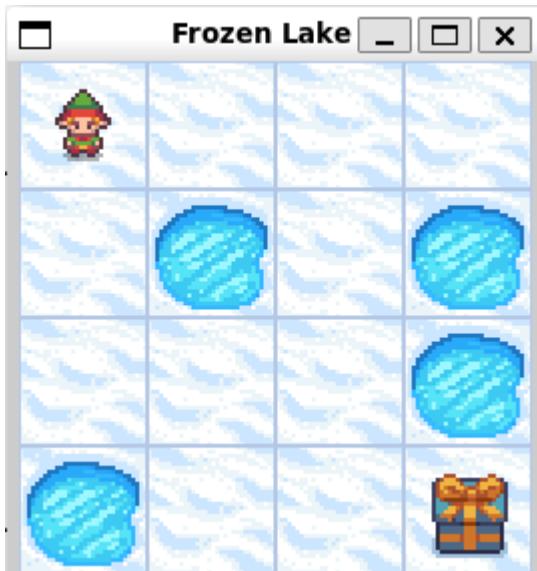
The Value function gives us an estimate of value for each possible movement from our actual state

Reinforcement Learning Policy

"Right action for each state"

Policy Format (4x4):

S → ↓ ←
↓ X ↓ X
→ ↓ ↓ X
X → → G



The Policy function tells us the best action to take on each state

Policy Gradient Explanation

Policy Gradient Methods are a class of Deep Reinforcement Learning

- Value-Based methods learn a value function to derive a policy
- Policy optimization methods learn directly a policy function π to select a function without a value function
- The policy function π_θ is parametrized by a differentiable parameter θ

Instead of learning "how good" an action is (value), **policy gradient methods learn the policy directly**. You use **gradient ascent** to improve the policy parameters θ so that the expected cumulative reward increases.

We define the **objective function**:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)]$$

where τ is a trajectory, and $R(\tau)$ is its return.

We update the policy using:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Advantages of Policy versus Value methods

- Advantages:
 - ▶ In some cases, computing Q-values is harder than picking optimal actions
 - ▶ **Better convergence properties**
 - ▶ **Effective in high dimensional or continuous action spaces**
 - ▶ Exploration can be directly controlled
 - ▶ **Can learn stochastic policies**
- Disadvantages:
 - ▶ Typically converge to a **local optimum** rather than a global optimum
 - ▶ Evaluating a policy is typically **data inefficient and high variance**

Main Policy Gradient Methods

Algorithm	Description
REINFORCE	Basic Monte Carlo policy gradient
Actor-Critic	Combines policy gradient (actor) with a value function (critic)
A2C / A3C	Synchronous / asynchronous actor-critic
TRPO	Trust region constrained PG for stability
PPO	Clipped surrogate objective; stable & widely used
DDPG	Deterministic PG for continuous action spaces
TD3	Twin critics + noise regularization (improvement on DDPG)
SAC	Soft Actor-Critic: entropy-regularized PG for better exploration

When to use Policy Gradient Methods

- Recommended when:
 - Action space is continuous (e.g., robotics)
 - You want a stochastic policy (e.g., for exploration)
 - You want to model complex behaviors directly
- Less ideal if:
 - You want very low variance learning quickly (PG methods have high variance)

The right model is for the right problem, there is not a killer model for everything

- Use Policy Gradient when:
 - The action space is continuous
 - You need stochastic decision-making
 - You care about directly optimizing performance
- Use DQN/DDQN when:
 - The action space is discrete
 - You want sample efficiency
 - You're working on games or tabular-like environments

Policy Gradient Methods

Stochastic and Deterministic

- In general, two kinds of policies:

- ▶ Deterministic policy

$$a = \pi_\theta(s)$$

- ▶ Stochastic policy

$$P(a|s) = \pi_\theta(a|s)$$

- Nice thing is that they are **smoother** than greedy policies, and so, we can compute **gradients!**
- Not new: ϵ -greedy is stochastic... but different idea. Stochastic policy is good on its own, not because it is an approx. of a greedy policy
- Any example where an stochastic policy could be better than a deterministic one?

Vanilla Policy Gradient

Given architecture with parameters θ to implement π_θ

Initialize θ randomly

repeat

 Generate episode $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T, s_T\} \sim \pi_\theta$

 Get $R \leftarrow$ long-term return for episode

for all time steps $t = 1$ to $T - 1$ **do**

$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t | s_t) R$

end for

until convergence

Substitute $\nabla_\theta \log \pi_\theta(a_t | s_t)$ with appropriate equation.

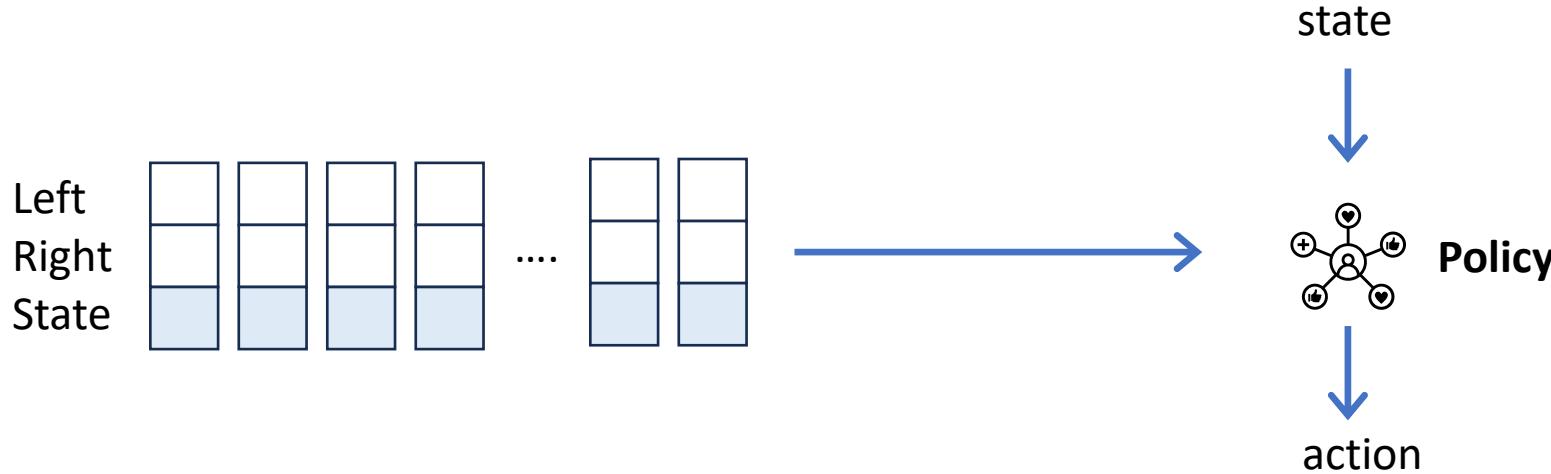
Btw, notice no explicit exploration mechanism needed when policies are stochastic (all on policy)!

The REINFORCE method is a fundamental policy gradient algorithm in deep reinforcement learning. Here are the main points:

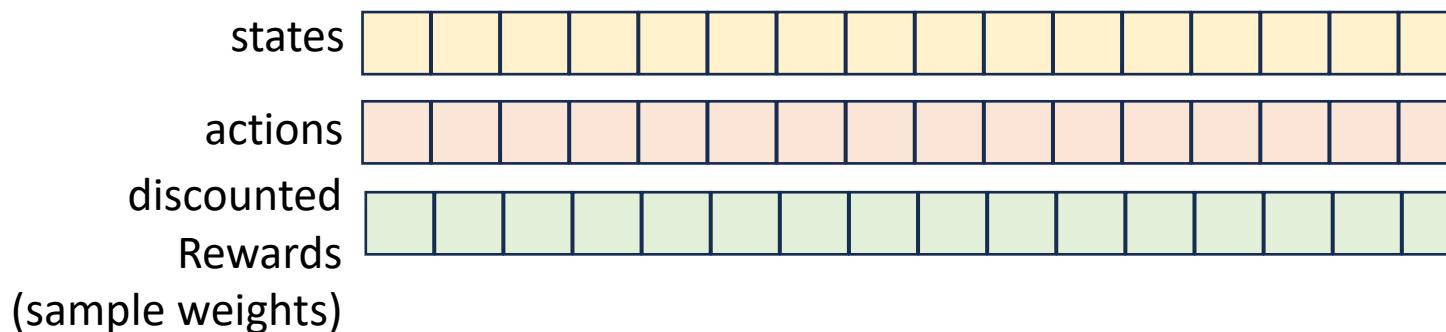
- **Policy-Based Method:**
 - REINFORCE directly parameterizes and optimizes the policy $\pi_\theta(a|s) \backslash \text{pi_theta}(a|s)$, unlike value-based methods (like DQN) which estimate value functions.
- **Monte Carlo Approach:**
 - It uses complete episodes to estimate returns — the updates happen only after an episode ends, using the total return G_t from each timestep.
- **Gradient Estimation:**
 - The core idea is to maximize expected return by computing the gradient:
 -
- **On-Policy**
 - REINFORCE learns from trajectories generated by the current policy only making it **on-policy**, and sensitive to policy changes during training

Policy Gradient Methods

REINFORCE



After each episode (Monte Carlo)



```
model.fit(states, actions, sample_weight=sample_weights, epochs=1, verbose=0)
```

Using **sparse_categorical_crossentropy** as a loss on a Softmax output layer is a trick to easily implement the policy gradient. Categorical cross entropy is defined as

$$H(p, q) = \sum_i p_i \log(q_i)$$

The **cross-entropy** between two probability distributions p (the true distribution) and q (the predicted distribution). It measures how well the predicted distribution q approximates the true distribution p

For an action a , you set $p_a = \text{advantage}$. q_a is the output of the policy network, which is the probability of taking the action a . But as we have that all other p_i are zero, thus we have

$$H(p, q) = A \cdot \log(\pi(s, a))$$

Policy Gradient Methods

REINFORCE

```
custom_opt = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, amsgrad=True, clipnorm=1.0)
model = Sequential()
model.add(Dense(self.hidden1, input_dim=self.state_size, activation='relu', dtype='float32'))
model.add(Dense(self.hidden2, activation='relu', dtype='float32'))
model.add(Dense(self.action_size, activation='softmax', dtype='float32'))
model.compile(loss='sparse_categorical_crossentropy', optimizer=custom_opt)
```

```

def train_model(self):
    """Training using sparse categorical crossentropy with sample weights"""
    if len(self.states) == 0:
        return

    episode_length = len(self.states)

    # Compute and normalize discounted rewards
    discounted_rewards = self.discount_rewards(self.rewards)
    discounted_rewards -= np.mean(discounted_rewards)
    if np.std(discounted_rewards) > 0:
        discounted_rewards /= np.std(discounted_rewards)

    # Prepare data
    states = np.array(self.states, dtype=np.float32)
    actions = np.array(self.actions, dtype=np.int32)

    # Use sample weights to weight the loss by advantages
    # For policy gradients, we want to increase probability of good actions
    # and decrease probability of bad actions
    sample_weights = discounted_rewards.astype(np.float32)

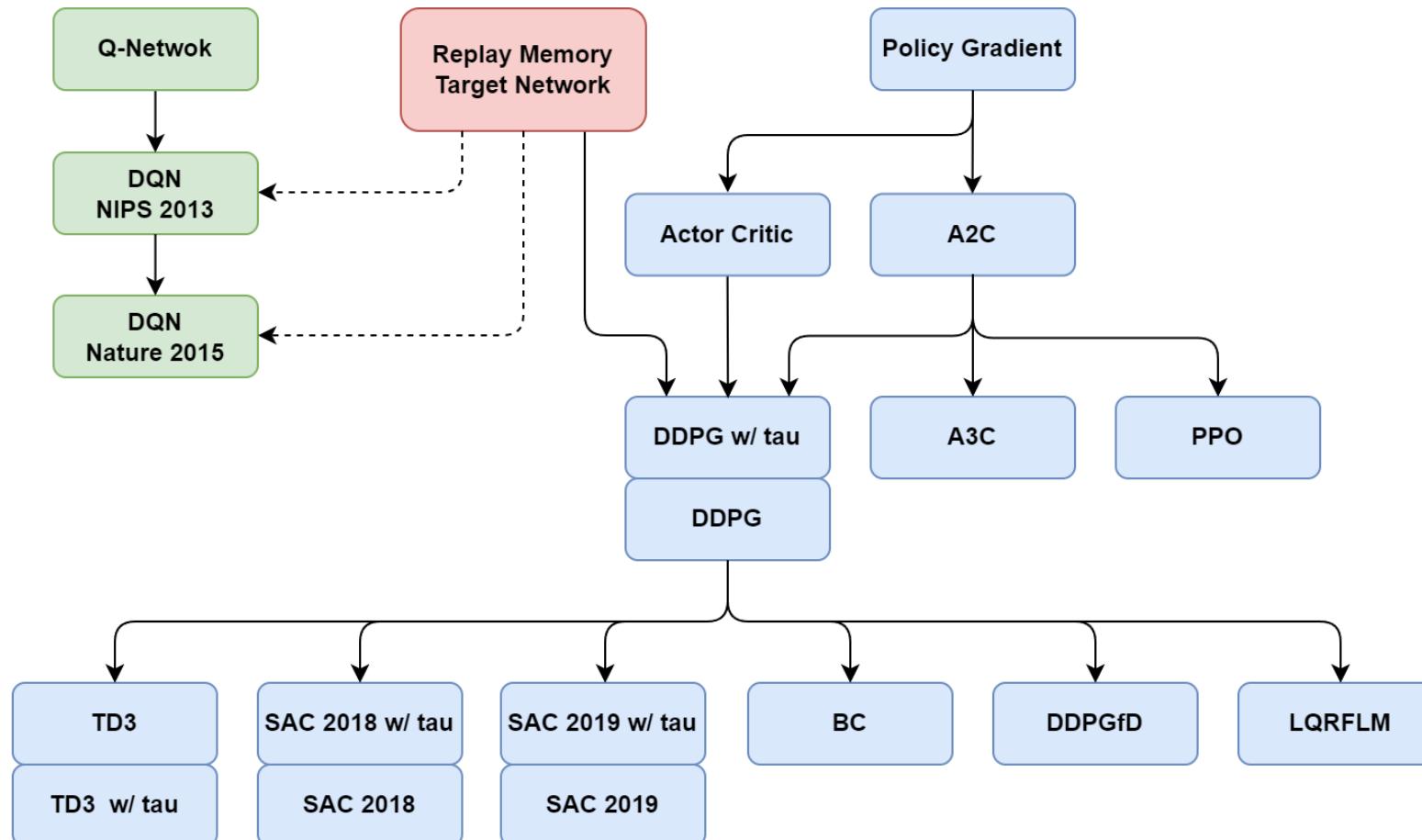
    states = states.squeeze(axis=1)
    # Train using fit with sample weights
    self.model.fit(states, actions,
                   sample_weight=sample_weights,
                   epochs=1, verbose=0)

    # Clear episode memory
    self.states, self.actions, self.rewards = [], [], []

```

Policy Gradient methods

The Rise of Policy Gradients methods



- **Actor-Critic Methods**
- **Problem Addressed:** Inefficiency of value-based methods in high-dimensional or continuous action spaces.
- **Key Innovations:**
 - Introduces two components:
 - **Actor:** Determines the policy (action selection).
 - **Critic:** Evaluates the action (value estimation).
- **Examples:**
 - Advantage Actor-Critic (A2C)
 - Asynchronous Advantage Actor-Critic (A3C)
- **Result:** Combines policy-based and value-based approaches for better stability

- **Proximal Policy Optimization (PPO)**
- **Problem Addressed:** Instability and inefficiency in policy-gradient methods.
- **Key Innovations:**
 - Uses a clipped surrogate objective function to ensure updates are not too large.
 - Avoids the need for complex trust-region optimization (as in TRPO).
- **Result:** Stable and scalable policy optimization.

- **Twin Delayed DDPG (TD3)**
- **Problem Addressed:** Overestimation bias in DDPG.
- **Key Innovations:**
 - Uses two critics and takes the minimum Q-value for updates.
 - Delays updates to the actor to reduce noise.
 - Adds noise to target actions for regularization.
- **Result:** Improved performance in continuous control.

Soft Actor-Critic (SAC)

- **Soft Actor-Critic (SAC)**
- **Problem Addressed:** Balancing exploration and exploitation in continuous spaces.
- **Key Innovations:**
 - Incorporates an entropy term into the objective function.
 - Encourages exploration by maximizing the entropy of the policy.
- **Result:** Stable and efficient for diverse continuous action tasks.

Multi-Agent Reinforcement Learning (MARL)

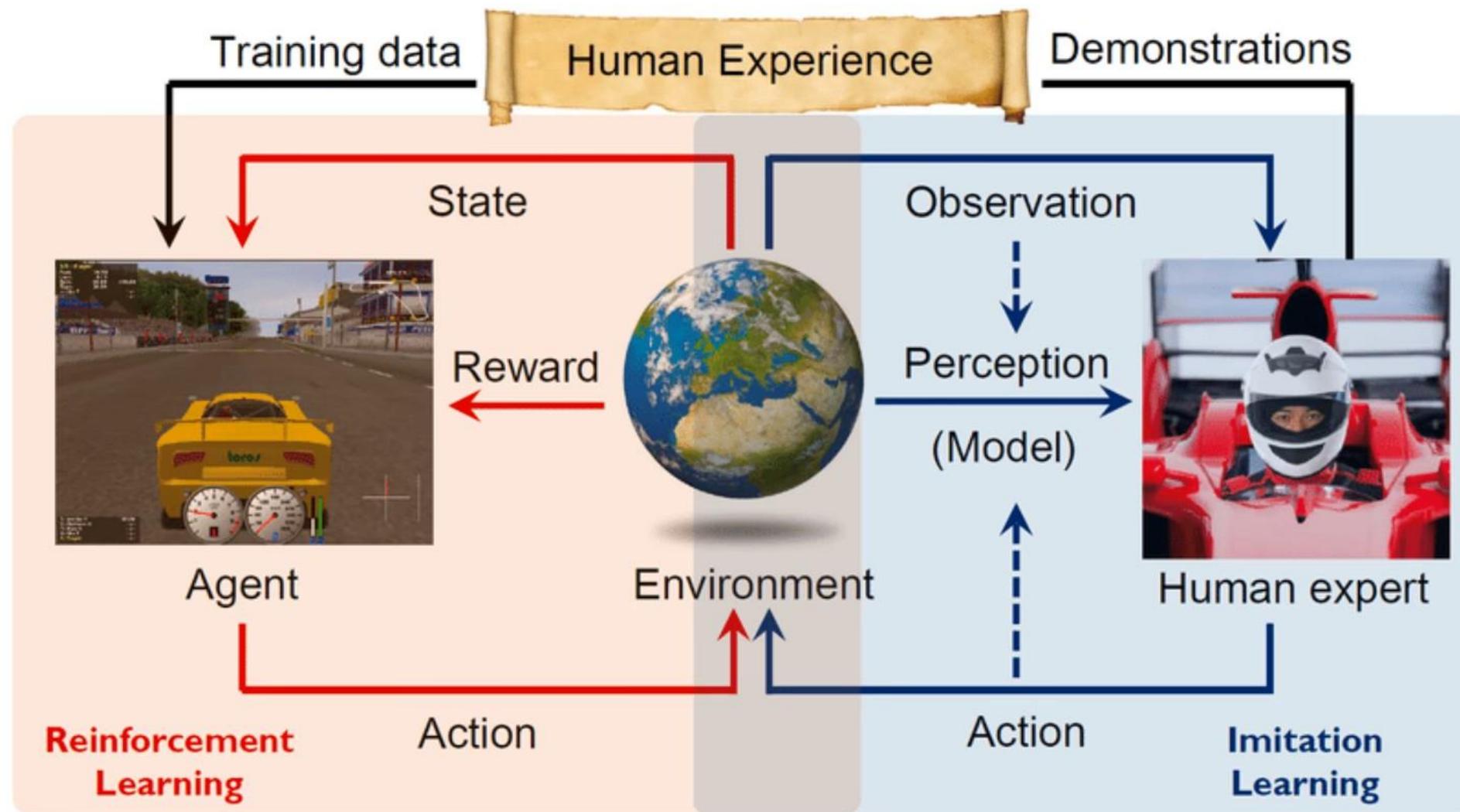
- **Multi-Agent Reinforcement Learning (MARL)**
- **Problem Addressed:** Collaboration and competition among multiple agents.
- **Key Innovations:**
 - Extensions of DQN and policy-gradient methods to handle multiple agents.
 - Examples:
 - MADDPG: Multi-agent DDPG.
 - QMIX: Value decomposition for cooperative agents.
- **Result:** Coordination among agents in complex environments

- 1. DQN
 - └ 2. Double DQN
 - └ 3. Dueling DQN
 - └ 4. Prioritized Experience Replay
 - └ Extensions to Continuous Spaces:
 - └ 5. DDPG → 6. TD3 → 7. SAC
 - └ Policy Improvements:
 - └ 5. Actor-Critic (A2C, A3C)
 - └ 6. PPO
 - └ 8. MARL

Imitation Learning

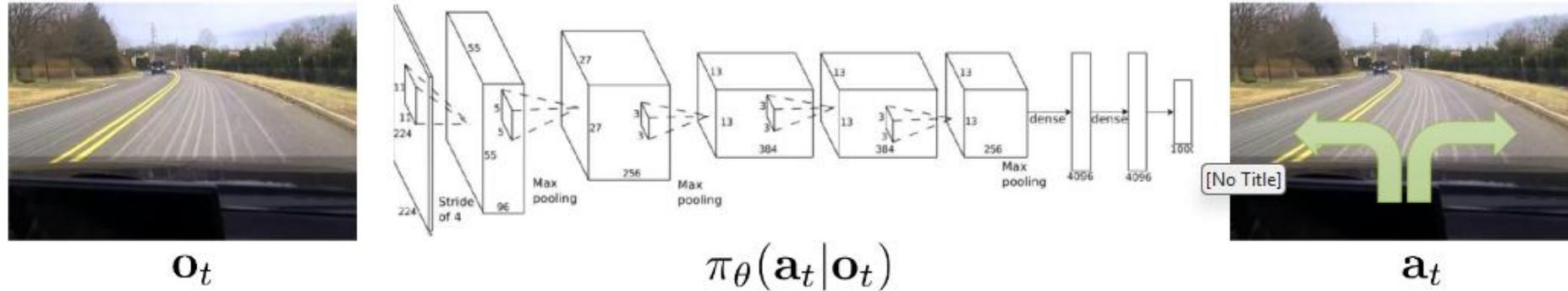
Imitation Learning

Also called behavioral cloning



Imitation Learning

A method to jump start from training data – Behavioral cloning



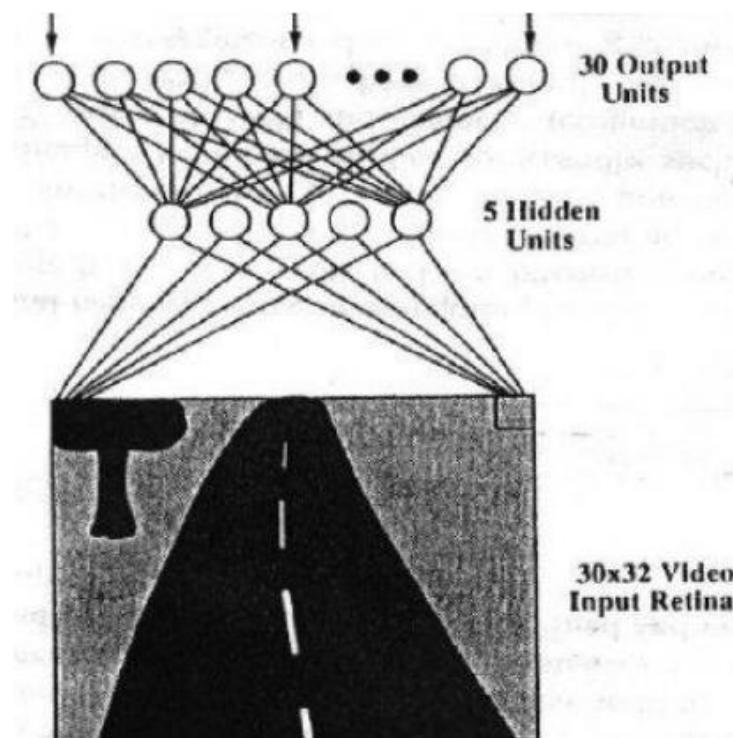
behavioral cloning

Imitation Learning

It is a method used from the beginning

ALVINN: Autonomous Land Vehicle In a Neural Network

1989

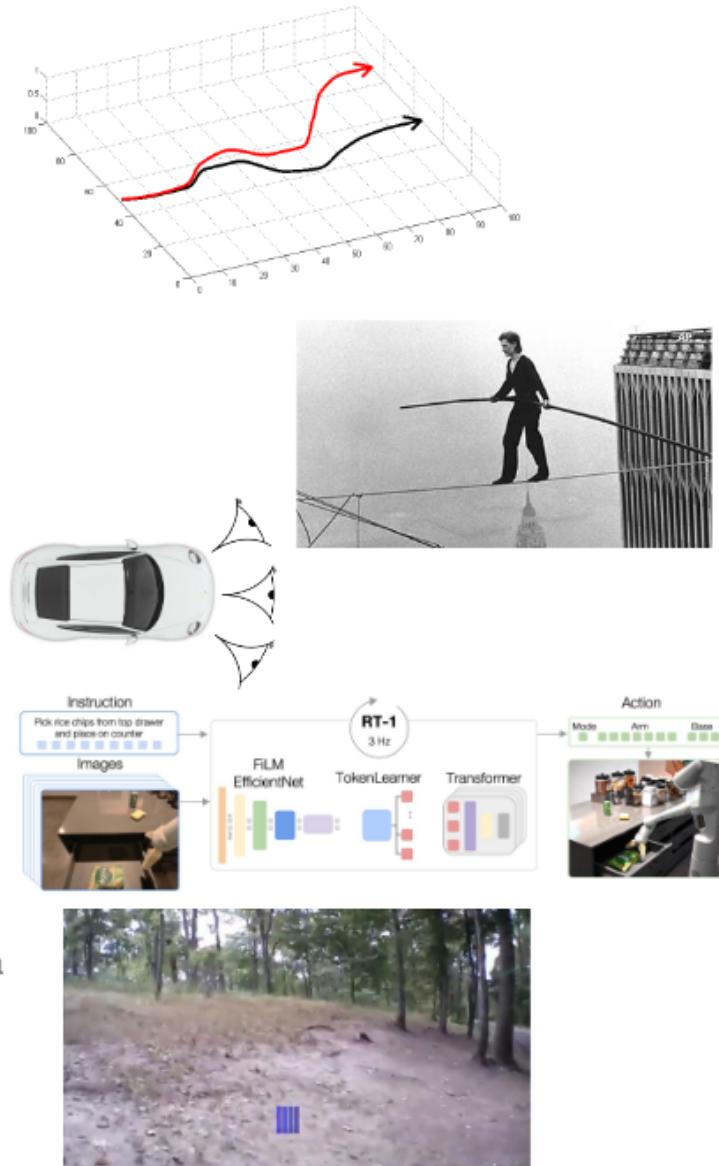
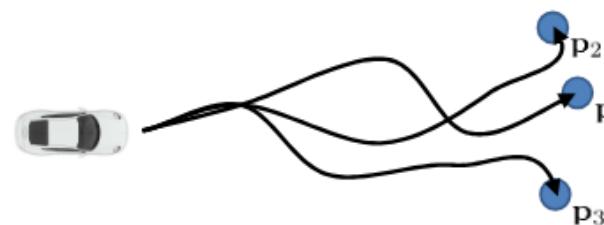


<https://youtu.be/ntlczNQKfjQ>

Imitation Learning

Good method but....

- Imitation learning via behavioral cloning is not guaranteed to work
 - This is **different** from supervised learning
 - The reason: i.i.d. assumption does not hold!
- We can formalize **why** this is and do a bit of theory
- We can address the problem in a few ways:
 - Be smart about how we collect (and augment) our data
 - Use very powerful models that make very few mistakes
 - Use multi-task learning
 - Change the algorithm (DAgger)



Imitation Learning

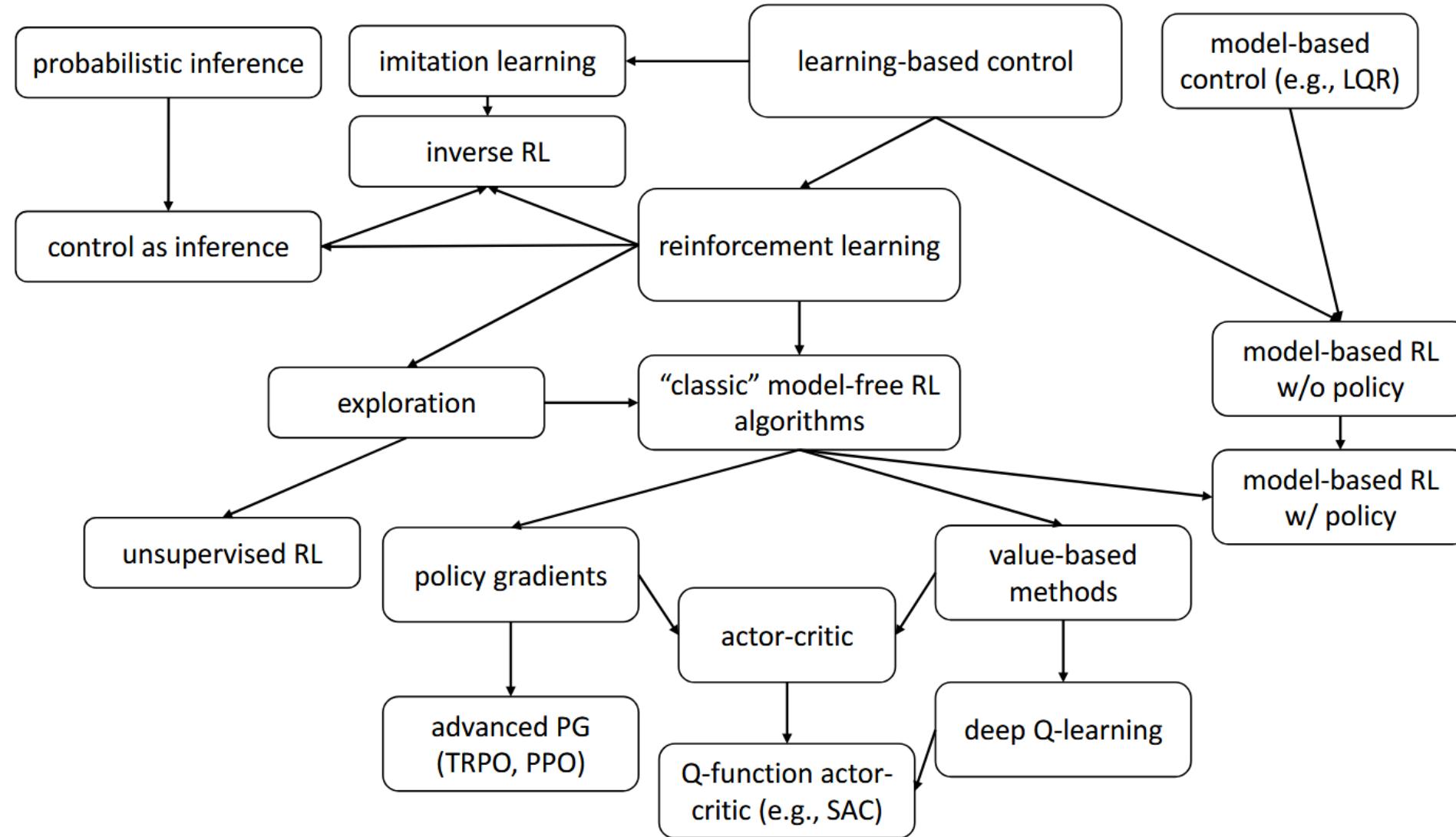
It has some issues

- **Distributional Shift (Covariate Shift):**
 - The agent may visit states not seen in expert demonstrations, leading to compounding errors because it has no guidance for those unfamiliar states.
- **Limited Generalization:**
 - Imitation learning often struggles to generalize beyond the demonstrated behaviors, especially if the demonstrations are sparse or narrow in scope.
- **Quality of Demonstrations:**
 - Performance heavily depends on the quality and optimality of expert trajectories. Poor or inconsistent demonstrations can mislead learning.
- **No Explicit Reward Signal:**
 - Unlike reinforcement learning, imitation learning does not use a reward function, making it hard for the agent to improve beyond mimicking.
- **Sample Inefficiency:**
 - It can require a large number of demonstrations to cover enough state-action pairs, especially in high-dimensional or continuous spaces.
- **Ambiguity in Actions:**
 - Multiple actions may be valid in a given state, but demonstrations usually provide only one, making it hard for the agent to learn robust, multimodal policies.

RL Challenges

RL Challenges

A maze of methods



What is the problem with RL

Challenges with **core algorithms**:

- Stability: does your algorithm converge?
- Efficiency: how long does it take to converge? (how many samples)
- Generalization: after it converges, does it generalize?

Challenges with **assumptions**:

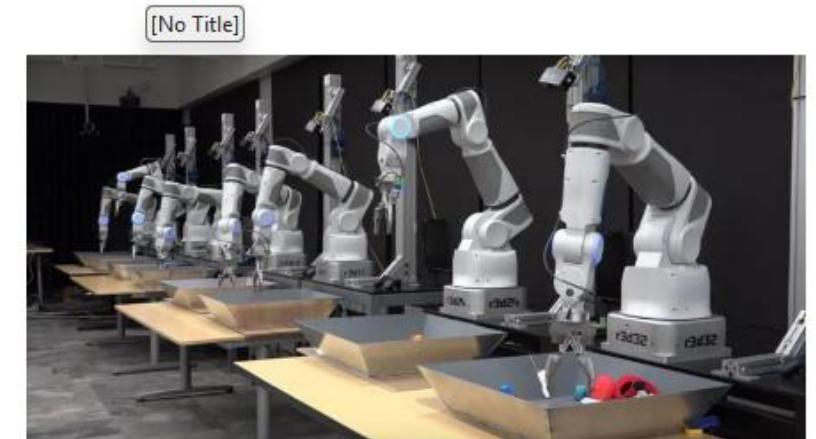
- Is this even the right problem formulation?
- What is the source of *supervision*?

What is the problem with RL

- Devising stable RL algorithms is very hard
- Q-learning/value function estimation
 - Fitted Q/fitted value methods with deep network function estimators are typically not contractions, hence no guarantee of convergence
 - Lots of parameters for stability: target network delay, replay buffer size, clipping, sensitivity to learning rates, etc.
- Policy gradient/likelihood ratio/REINFORCE
 - Very high variance gradient estimator
 - Lots of samples, complex baselines, etc.
 - Parameters: batch size, learning rate, design of baseline
- Model-based RL algorithms
 - Model class and fitting method
 - Optimizing policy w.r.t. model non-trivial due to backpropagation through time
 - More subtle issue: policy tends to *exploit* the model

The Challenge with sample complexity

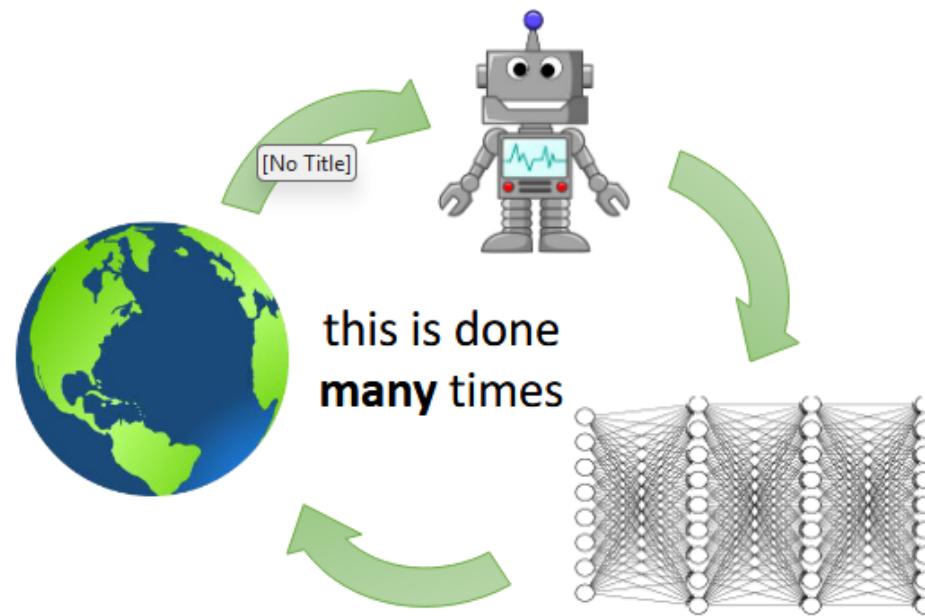
- Need to wait for a long time for your homework to finish running
- Real-world learning becomes difficult or impractical
- Precludes the use of expensive, high-fidelity simulators
- Limits applicability to real-world problems



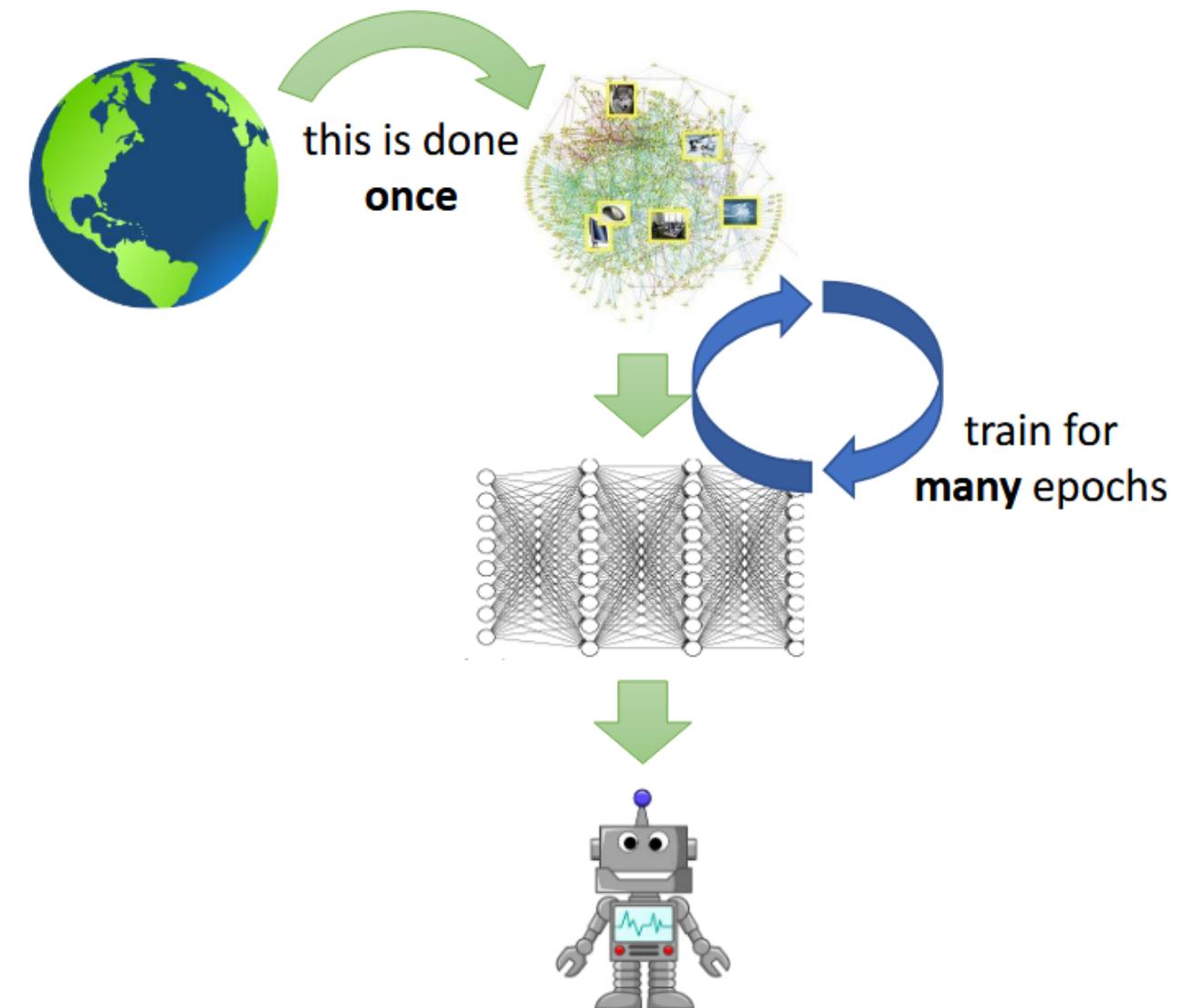
Challenges

The problem of RL

reinforcement learning

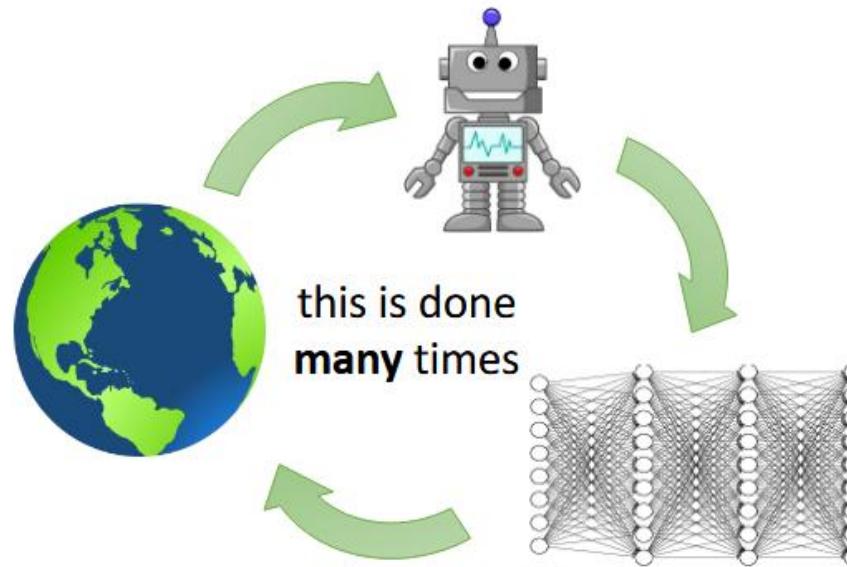


supervised machine learning

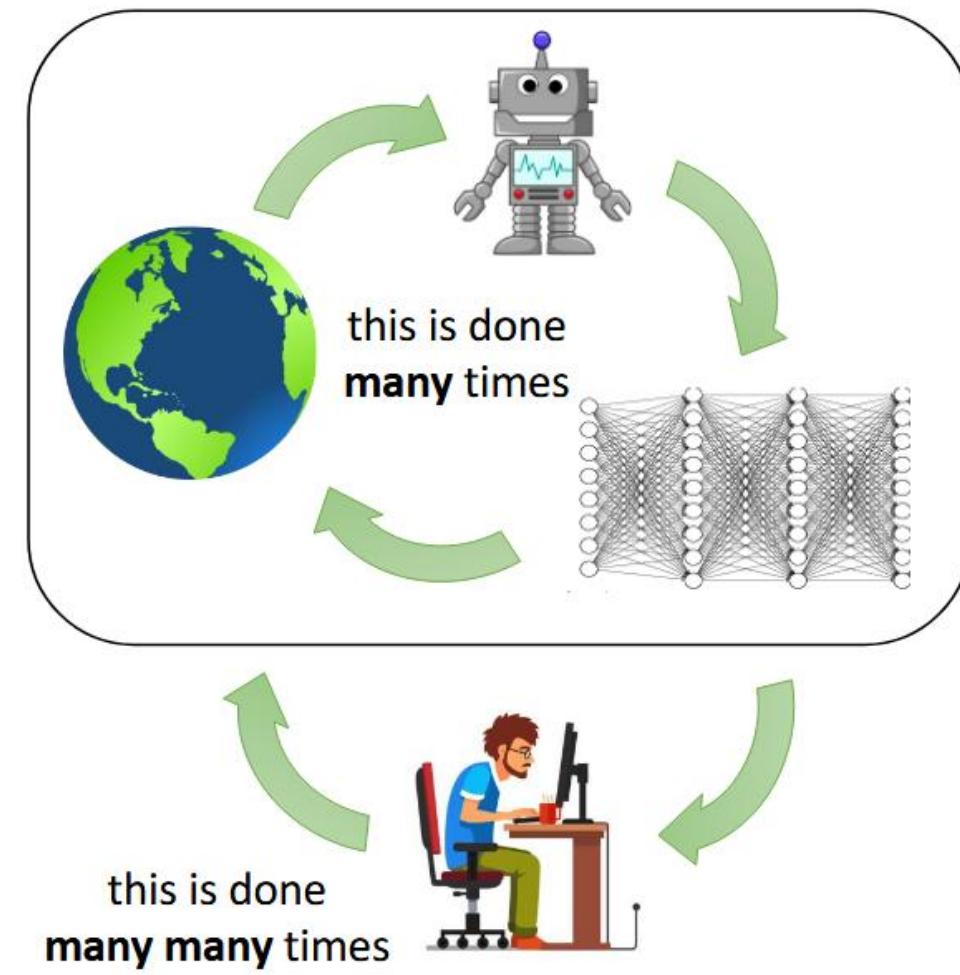


The problem of RL

reinforcement learning



actual reinforcement learning



New ways to do RL

- Demonstrations

- Muelling, K et al. (2013). Learning to Select and Generalize Striking Movements in Robot Table Tennis

Should supervision tell us **what** to do or **how** to do it?



- Language

- Andreas et al. (2018). Learning with latent language

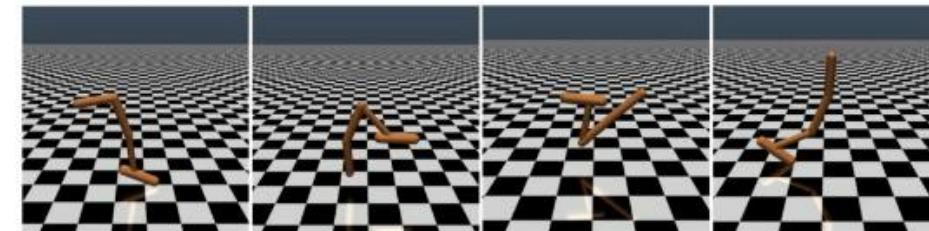
Human description:
move to the star

Inferred description:
reach the star cell



- Human preferences

- Christiano et al. (2017). Deep reinforcement learning from human preferences

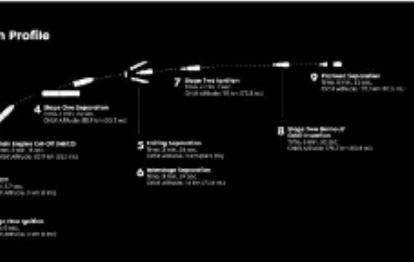
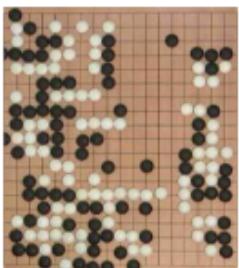


Challenges

Moravec's Paradox

Moravec's paradox seems like a statement about AI
but it is actually a statement about the physical universe

“easy” universes



Why?

“hard” universes



We are all prodigious olympians in perceptual and motor areas, so good that we make the difficult look easy. Abstract thought, though, is a new trick, perhaps less than 100 thousand years old. We have not yet mastered it. It is not all that intrinsically difficult; it just seems so when we do it.

- Hans Moravec

The main lesson of thirty-five years of AI research is that the hard problems are easy and the easy problems are hard. The mental abilities of a four-year-old that we take for granted – recognizing a face, lifting a pencil, walking across a room, answering a question – in fact solve some of the hardest engineering problems ever conceived.

- Steven Pinker

Dealing with the unexpected



How do we engineer a system that can deal with the unexpected?

- Minimal external supervision about what to do
- Unexpected situations that require adaptation
- Must discover solutions autonomously
- Must “stay alive” long enough to discover them!

- Humans are extremely good at this
- Current AI systems are extremely bad at this
- RL *in principle* can do this, and nothing else can

Challenges

Dealing with the unexpected



RL **should** be really good in the
“hard” universes!

➤ RL *in principle* can do this, and nothing else can

But we rarely study this kind of setting in RL research!

“easy” universes

success = high reward
 (“optimal control”)

closed world, rules
are known

lots of simulation

Main question: can RL
algorithms **optimize**
really well

“hard” universes

success = “survival”
 (“good enough control”)

open world, everything
must come from data

no simulation (because
rules are unknown)

Main question: can RL
generalize and **adapt**

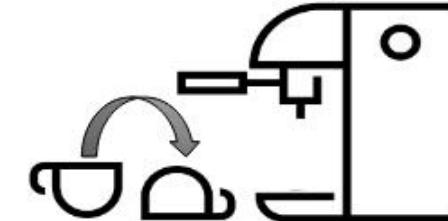
Wrap-up

Dealing with the unexpected

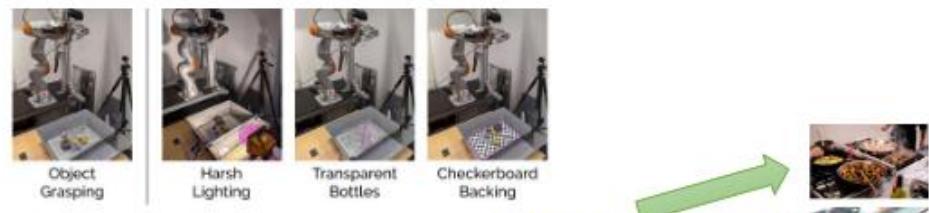
How do we tell RL agents **what we want them to do?**



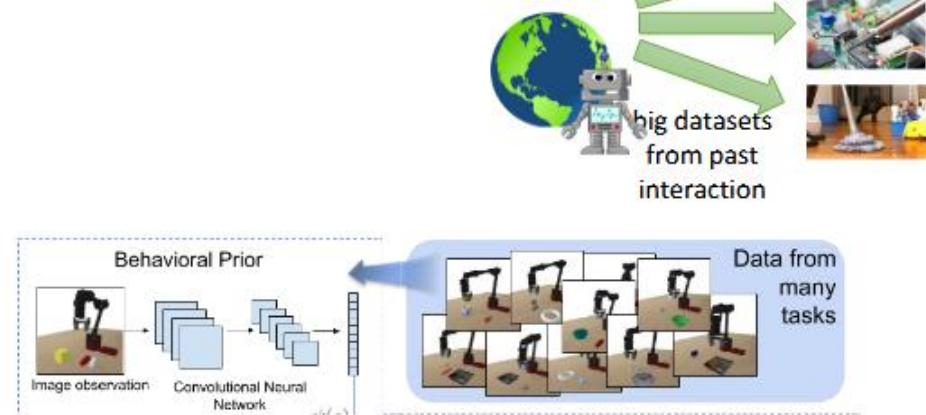
How can we learn **fully autonomously** in continual environments?



How do remain **robust** as the environment changes around us?



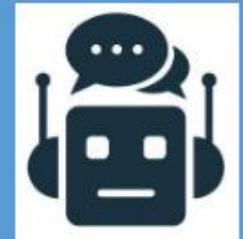
What is the right way to **generalize** using **experience & prior data**?



What is the right way to **bootstrap exploration with prior experience**?

This is not about robots

Robots are the most natural for us to think about, because they are embodied like we are



One last thing

What about using LLM?

What we want:



Can you teach me about behavior cloning?



Of course! I'd be happy to explain behavior cloning. To start, could you tell me if you've ever come across the terms "artificial intelligence" or "machine learning" before?



Yes, I've heard those terms, but I'm not exactly sure what they mean.



No problem at all! Let's take it step by step. Have you ever used a computer or a smartphone?

What we get:



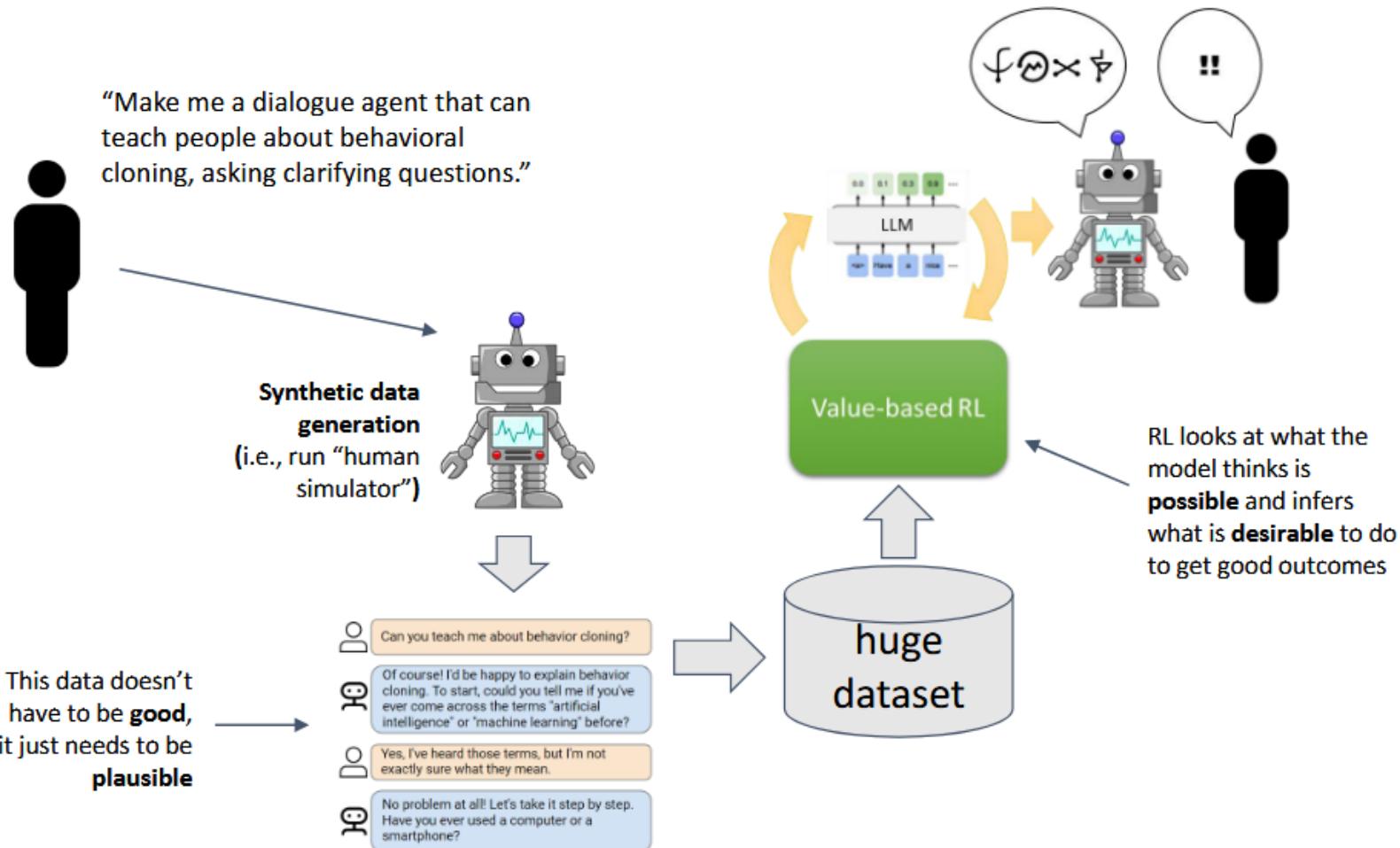
Can you explain the AI concept of behavior cloning to me? But first, I want you to ask me questions to figure out what my background in AI is, then tailor your explanation to match my level.



Certainly! Let's start by understanding your background in AI to provide an explanation that matches your level of knowledge. Please answer the following questions:

1. Have you ever studied or worked with artificial intelligence or machine learning concepts before, even at a basic level?
2. Are you familiar with terms like "neural networks," "training data," or "supervised learning"?
3. Have you ever heard of or worked with specific AI tools or frameworks like TensorFlow, PyTorch, or scikit-learn?
4. Are you interested in a high-level overview or a more detailed explanation of behavior cloning?
5. Do you have any specific examples or applications in mind where you'd like to see behavior

What about using LLM?



RL + LLM is great for tasks that are easy for LLMs to **simulate** but hard for them to **perform optimally**.

With RL, instead of emulating humans, LLMs can learn to achieve desired outcomes using their understanding of how humans behave.

**END
Session 11**

