Computer Science, Engineering and Technology
Vol: 3(2), June 2025
REST Publisher; ISSN: 2583-9179
Website: https://restpublisher.com/journals/cset/
DOI: https://doi.org/10.46632/cset/3/2/11

Optimizing Decision Strategies through Advanced Learning Techniques

***Pradeep Kumar Reddy Dasari Leela**
Corresponding author Email: *pdasari1@umbc.edu*

Abstract: *This study explores an approach to refining decision- making models using adaptive learning methods. By structure data-driven strategies, the research enhances efficiency in complex scenarios. Various techniques are examined to improve adaptability and performance, with an emphasis on practical applications. The findings highlight key advancements that con- tribute to more effective and controlled learning processes.*

1. INTRODUCTON

This section introduces fundamental concepts in Reinforce- ment Learning (RL), a framework where agents learn to make decisions by interacting with an environment.

A. Core Concepts

Consider a discrete-time system, where at each time step t, the environment is in a state st and the agent selects an action at. The agent's behaviour is determined by a policy πθ, which can be deterministic (at = πθ (st)) or stochastic (πθ (at|st)). When states are only partially observable, the policy may depend on observations ot. The environment evolves based on a transition function or model, defined as a conditional probability distribution p (st+1|st, at). A sequence of states and actions forms a trajectory τ = (s1, a1... sT, at). The agent receives feedback through a reward function r(s, a), which can be known or learned (as in Inverse RL). The goal is to maximize the expected cumulative reward:

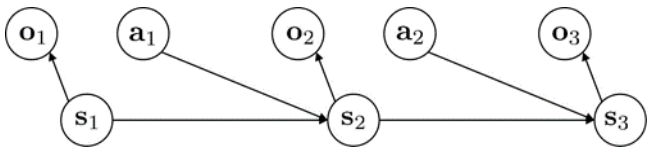

FIGURE 1. A simple Markov chain.

θ = arg max E ∼ "Σ r (st, at) #,

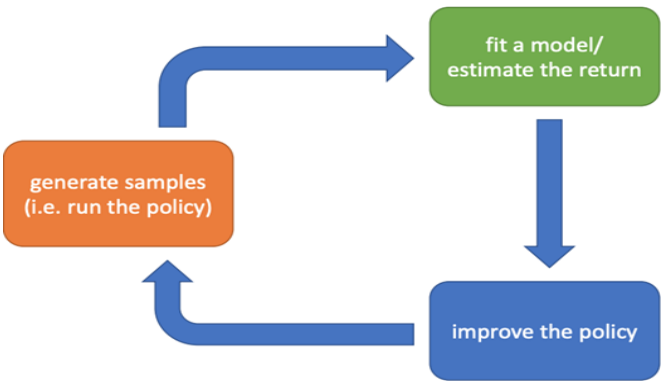

FIGURE 2. Three-step reinforcement learning pipeline.

Where the trajectory distribution is:

P (τ) = p(s) Y π (a |s) p (s|s, a).

2) Value Function
The value function estimates the expected future reward
I often assume the Markov property, where st+1 depends T only on st and at (see Fig. 1).
V π (st) = Eπ[r (set′, at′) |st], t′=t

B. Value-Based Functions
To facilitate policy optimization, two key functions are defined:

1) Q Function
The Q-function quantifies the expected future reward from taking action at in state st under policy π:
T Qπ (st, at) = Eπ[r (st′, at′) |st, at]. t′=t
With the relation V π (st) = Ea ∼π (•|s) [Qπ (st, at)]. The over- all objective becomes Es ∼p(s) [V π (s1)].

C. RL Workflow
RL involves three key stages: (1) data generation through agent-environment interaction, (2) function approximation us- ing collected samples, and (3) policy improvement based on estimated rewards. This loop continues iteratively to enhance agent performance (see Fig. 2).

2. IMITATION LEARNING

Imitation learning, also known as behavioural cloning, aims to train an agent to replicate expert behaviour from demonstrate. The agent is trained to fit a policy π (at|ot) that maps observations to actions.

Distribution Mismatch
A core challenge is the distribution mismatch between training and test time. Small mistakes can lead the agent into unfamiliar states, compounding errors and deviating from the expert trajectory, as illustrated in Fig. 3.

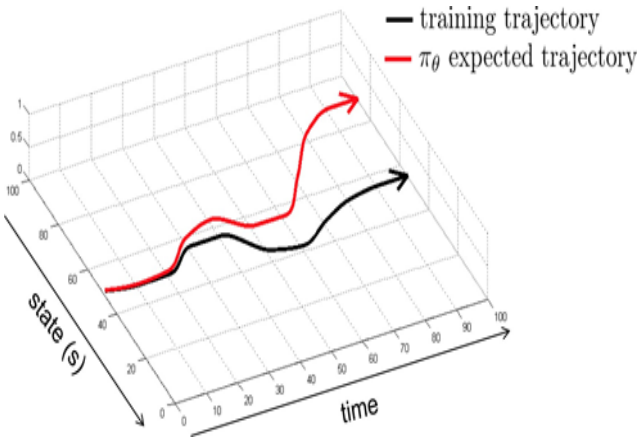

FIGURE 3. Compounding errors in behaviour cloning.

Dataset Aggregation (D Agger)
To mitigate error accumulation, Dataset Aggregation (DAg- ger) [1] iteratively augments the training dataset with agent- collected states, relabeled by a human expert.

Algorithm 1 Dataset Aggregation (DAgger)
1. Require: Expert dataset D = {(oi, ai)} N
2. while not converged do
3. Train policy πθ on D
4. Run πθ to collect observations DP
5. Reliable DP with expert actions
6. Aggregate: D ← D ∪ DP
7. return Final policy πθ

D Agger aligns the training and test distributions but relies heavily on human supervision, which can introduce noise and is not always Markova.

Failure Modes of Imitation Learning

1) Non-Markova Behaviour

Human decisions often depend on historical context, leading to a mismatch when fitting Markovian policies. One approach is to use recurrent neural networks (e.g., LSTM) to encode history, as shown in Fig. 4. However, longer histories can introduce causal confusion, where the agent learns spurious correlations. For example, an agent might associate braking with a dashboard light rather than an obstacle.

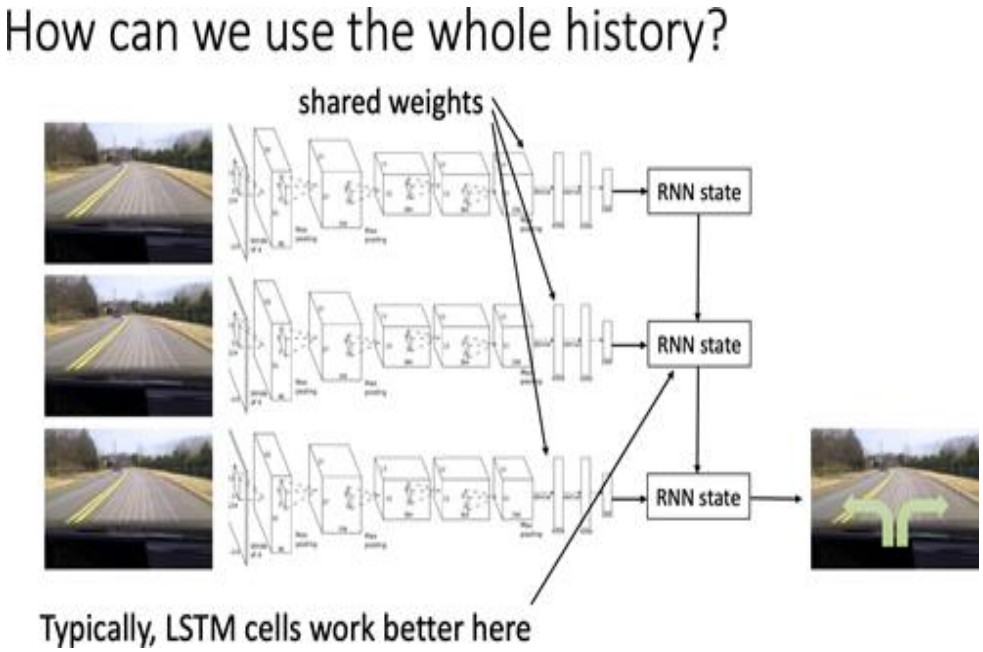

FIGURE 4. Addressing non-Markovian data with RNN.

2) Multimodal Expert Behavior

Experts may exhibit multiple valid behaviors in the same state (e.g., avoiding a tree by steering left or right). A unimodal policy (e.g., Gaussian) may average actions, leading to unsafe behavior (e.g., going straight into the tree). Solutions include using mixtures of Gaussians, latent variable models, or autoregressive discretization.

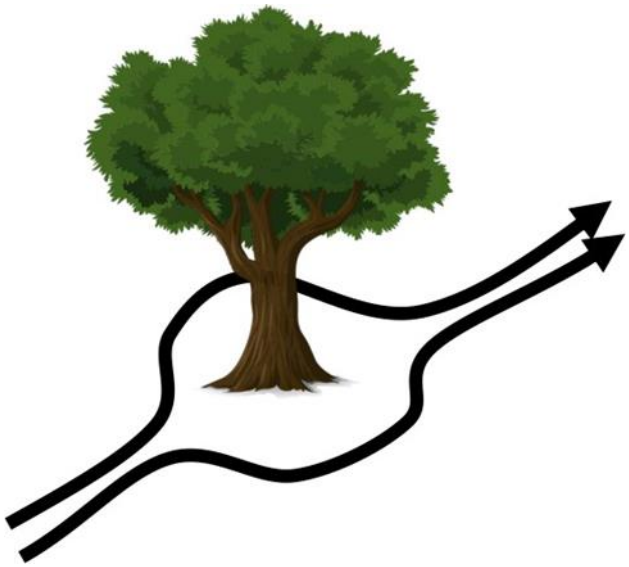

FIGURE 5. Averaging multimodal actions can result in failure.

Theoretical Error Analysis

Let $\pi*$ be the expert policy and assume the imitation policy $\pi\theta$ has error rate ϵ on training distribution ptrain(s). The state distribution under $\pi\theta$ becomes:

Pθ (st) = (1 − ϵ) tptrain (st) + (1 − (1 − ϵ) t) pmistake (st)
The total variation divergence yields:
|pθ (st) − ptrain(st)| ≤ 2ϵt
Thus, the expected number of mistakes grows as:
E [# mistakes 2] ∈ O (ϵT) Imitation learning faces challenges such as distribution
Mismatch, non-Markovian data, multimodal behaviour, and compounding errors. While DAgger addresses some of these issues, it depends on human supervision, which may be noisy or insufficient for certain state-action spaces. Future work explores learning from unlabelled or synthetic data to overcome these limitations.

3. POLICY GRADIENT METHODS

Reinforcement Learning (RL) can be framed as an optimization problem:
θ = arg max Eτ∼pθ (τ) "Σ r (st, at) #
Define the objective function: J(θ) = Eτ ∼πθ(τ)[r(τ)] = ∫ πθ(τ)r(τ) dτ

Policy Gradient Theorem

Using the identity ∇θπθ (τ) = πθ (τ) ∇θ log πθ (τ), I get:
∇θJ (θ) = Eτ∼πθ (τ) [∇θ log πθ (τ) r (τ)]
For trajectory τ = (s, a, s, a):
Log πθ (τ) = log p (s1) +Σ log πθ (at|st) +Σ log p (st+1|st, at)
Only πθ (at|st) depends on θ, so:
Θ τ ∼πθ (τ) θ t=1 θ t t t=1

Monte Carlo Approximation

Since the expectation is intractable, approximate using Monte Carlo with N samples:
∇θJ (θ) ≈ N ΣT ∇θ log πθ (ai, t|si, and t)! ΣTR (si,t, ai,t)
Update policy via gradient ascent:
Θ ← θ + α∇θJ (θ)

Example: Gaussian Policy

For πθ (at|st) = N (fθ (st), Σ):
Log πθ (at|st) = −2fθ (st) − at 2 + C∇
Gradient: ∇ log π (a |s) = −Σ−1 (f (s) − a) dfθ

Algorithm 2 REINFORCE Algorithm

Require: Policy πθ (a|s), learning rate α
While not converged do
Sample N trajectories {τi} from πθ

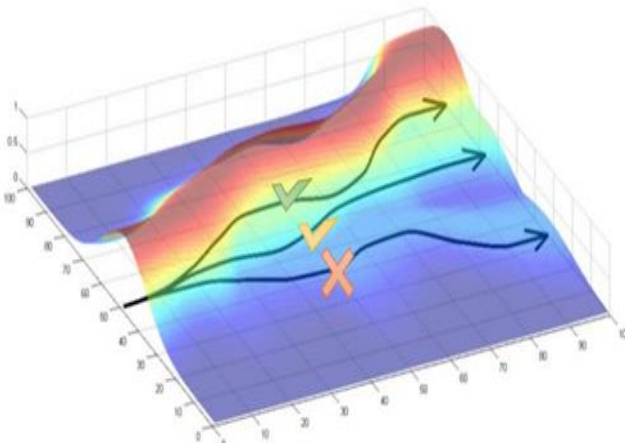

FIGURE 6. Higher-reward trajectories become more probable

Intuition behind Policy Gradient

Policy gradient increases the probability of high-reward trajectories:
∇ J (θ) ≈ 1 Σ ∇ log π (τ) r (τ)

Compared to Maximum Likelihood:
Policy gradient assigns more weight to high-reward trajectories, effectively learning to prefer them.

Policy Gradient in POMDPs

The policy gradient does not rely on the Markov property.
In POMDPs, replace states stwith observations ot:
∇θJ (θ) ≈ Ni=1∇θ log πθ (ai,t|oi,t)t r(oi,t, ai,t)

Variance Reduction Using Baselines

Causality

By causality, actions at time t cannot influence past rewards:
∇θJ(θ) ≈ N ΣΣ∇θ log πθ(ai,t|si,t) ΣTr(si,t′ , ai,t′)! i=1 t=1
This "reward-to-go" reduces variance by excluding irrele- vant past rewards. θ θ t t θ tt dθ

Drawback of Naive Policy Gradient

Naive policy gradient is sensitive to reward scaling. Adding a constant to all rewards shifts their values but affects the gradient direction, leading to high variance. Variance-reducing to stabilize learning.

4. BASELINES IN POLICY GRADIENT

To reduce variance in policy gradient methods, baselines are introduced—typically the average reward—so that only above- average trajectories contribute to learning. Let the baseline be defined as:
b = 1 Σ r (τ)
Incorporating this into the policy gradient yields:
∇ J (θ) ≈ 1 Σ ∇ log π (τ) [r (τ) – b

5. ACTOR-CRITIC ALGORITHMS

In the previous chapter, I derived the policy gradient theorem

$$\nabla_\vartheta J(\vartheta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\vartheta \log \pi_\vartheta(a_{i,t}|s_{i,t}) \right) \times \left(\sum_{t'=t} r(s_{i,t'}, a_{i,t'}) \right) \quad (1)$$

where the term Qˆi,t represents the reward-to-go, i.e., the expected cumulative reward from time t onward. While this Monte Carlo-based estimate is unbiased, it suffers from high variance. I now explore how actor-critic methods improve upon this by leveraging better approximations of Qˆi,t.

Reward-to-Go and Q-Function

A more precise form of the reward-to-go is the expected return conditioned on a state-action pair:

$$Q(s_t, a_t) = \sum_{t'=t}^{T} \mathrm{E}\left[r(s_{t'}, a_{t'}) | s_t, a_t\right]$$

Replacing Monte Carlo returns with Q (st, at) yields a refined policy gradient estimate:

$$\nabla_\vartheta J(\vartheta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\vartheta \log \pi_\vartheta(a_{i,t}|s_{i,t}) Q(s_{i,t}, a_{i,t})$$

Variance Reduction with Baselines

To further reduce variance, I subtract a baseline from Q. A common choice is the value function:

$$V(s_t) = \mathrm{E}_{a_t \sim \pi_\vartheta}[Q(s_t, a_t)]$$

This gives rise to the advantage function, which quantifies the relative value of taking action at in state st:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

Substituting this into the policy gradient, I obtain:

$$\nabla_\vartheta J(\vartheta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\vartheta \log \pi_\vartheta(a_{i,t}|s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$

Approximating the Advantage Function
To approximate Aπ, I leverage the relation

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1}}[V^{\pi}(s_{t+1})]$$
$$\approx r(s_t, a_t) + V^{\pi}(s_{t+1})$$

Which leads to a simple estimator:

$$A^{\pi}(s_t, a_t) \approx r(s_t, a_t) + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$$

Thus, estimating is key to computing efficiently

Policy Evaluation via Value Function Fitting
To assess how good a policy is, I evaluate its value function:

$$V^{\pi}(s_t) = \sum_{t'=t}^{T} E[r(s_{t'}, a_{t'}) | s_t]$$

Practically, I approximate this using Monte Carlo returns:

$$V^{\pi}(s_t) \approx \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

or over N sampled rollouts:

$$V^{\pi}(s^t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$$

Even when using single-sample returns, training a neural network to fit these estimates allows for generalization across similar states, providing a low-variance, learnable baseline to support actor-critic training

1) Monte Carlo Evaluation with Function Approximation

To approximate the value function, I treat it as a supervised learning problem where the target label is the cumulative return from a state. Specifically, the training set is {(si,t, yi,t)}

Batch Actor-Critic
The actor-critic framework reduces variance in policy gra- dient methods by incorporating a value function (critic). In the batch version (Alg.??), I sample state-action pairs, fit the value function, compute advantages, and update the policy via:

$$\nabla_{\theta} J(\vartheta) \approx \sum_{i} \nabla_{\vartheta} \log \pi_{\vartheta}(a_i|s_i) \hat{A}^{\pi}(s_i, a_i)$$

where $\hat{A}^{\pi}(s, a) = r(s, a) + \hat{V}_{\phi}(s') - \hat{V}_{\phi}(s)$.

Discounting for Infinite Horizons
To prevent unbounded returns in infinite horizon problems, I introduce a discount factor γ ∈ [0, 1]. The bootstrapped target while bootstrapped actor-critic estimates reduce variance at the cost of bias. A compromise is to truncate the trajectory and compute n-step returns: becomes yi,t= r(si,t, ai,t) + γV^π (si,t+1). Discounting also affects the policy gradient, where I prefer the form:

$$\nabla_{\theta} J(\vartheta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right)$$

as it yields lower variance in practice. I incorporate this into the batch actor-critic in Alg.??.GAE ttn=1n t t

Online Actor-Critic

In the online variant (Alg. ??), updates are made per interaction step without storing full trajectories. Using two networks for policy and value function (or a shared network), I update the critic with target $r + \gamma \hat{V}_\phi(s')$, ompute the advantage, and update the actor with the resulting gradient. For efficiency, parallel simulations are often employed (Fig. ??), though asynchronous setups may result in slightly outdated policies during data collection.

Critics as State-Dependent Baselines

In Monte Carlo policy gradients, the gradient estimate uses the return minus a baseline to reduce variance, maintaining unbiasedness

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \tag{4}$$

$$\left(\sum_{t'=1}^{T} r(s_{it'}, a_{it'}) - b \right) \tag{5}$$

In contrast, actor-critic methods utilize a learned critic to estimate the advantage function, reducing variance at the cost of introducing bias:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1} \sum_{t=1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \tag{6}$$

$$r(s_{it}, a_{it}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) - \hat{V}_\phi^\pi(s_{it}) \quad . \tag{7}$$

To better trade off bias and variance, a common approach is to use the critic as a state-dependent baseline, yielding an unbiased yet lower-variance gradient estimator:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \tag{8}$$

$$\sum_{t'=1} r(s_{it'}, a_{it'}) - \hat{V}_\phi^\pi(s_{it}) \quad . \tag{9}$$

Eligibility Traces and n-Step Returns

Monte Carlo and actor-critic methods reflect a bias-variance tradeoff: Monte Carlo returns are unbiased but high-variance, while bootstrapped actor-critic estimates reduce variance at the cost of bias. A compromise is to truncate the trajectory and compute *n*-step returns:

$$\hat{A}_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_\phi^\pi(s_t) + \gamma^n \hat{V}_\phi^\pi(s_{t+n}).$$

Choosing n > 1 often yields more stable learning. General Advantage Estimation (GAE) further refines this by averaging over multiple n-step returns with exponentially decaying weights:

$$\hat{A}_{GAE}(s_t, a_t) = \sum_{n=1}^{\infty} \lambda^{n-1} \hat{A}_n^\pi(s_t, a_t),$$

where λ adjusts the tradeoff between bias and variance

6. VALUE FUNCTION METHODS

An Implicit Policy

To bypass policy gradients, I can derive actions directly from the advantage function by selecting the action that maximizes it:

$$\arg\max_{a_t} A^\pi(s_t, a_t).$$

This yields an implicit, deterministic policy:

$$\pi'(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_{a_t} A^{\pi}(s_t, a_t) \\ 0, & \text{otherwise} \end{cases}$$

Though the original policy π is not explicitly used, the resulting policy is guaranteed to be at least as good under accurate advantage estimates.

Policy Iteration

1) High-Level Idea

Policy iteration repeatedly improves a policy by evaluating and maximizing the advantage function. Since advantage be expressed in terms of value functions:

$$A^{\pi}(s, a) = r(s, a) + \gamma E\,[V^{\pi}(s')] - V^{\pi}(s),$$

Approximating V π(s) suffices to guide the policy update. This method omits gradient steps and instead focuses on alternating

2) Dynamic Programming

Assuming discrete state-action spaces and known transition probabilities p(s′|s, a), I can update the value function using the standard dynamic programming (DP) approach:

$$V^{\pi}(s) \leftarrow E_{a\sim\pi(a|s)}\ r(s, a) + \gamma E_{s'\sim p(s'|s,a)}\,[V^{\pi}(s')]$$

If the policy π is deterministic, the expectation over actions disappears, yielding a simplified update:

$$V^{\pi}(s) \leftarrow r(s, \pi(s)) + \gamma E_{s'\sim p(s'|s,\pi(s))}\,[V^{\pi}(s')]$$

Alternatively, I can update values directly without main- taining an explicit policy. Since arg maxa Aπ(s, a) = arg maxa Qπ(s, a) due to Aπ(s, a) = Qπ(s, a) − V π(s), I use this equivalence in value iteration:

Algorithm 3 Policy Iteration (DP)

1. while not converged do
2. Evaluate V π(s)
3. Improve policy π ← π′

Algorithm 4 Value Iteration (DP)

1. while not converged do
2. Q(s, a) ← r(s, a) + γE[V (s′)]
3. V (s) ← maxa Q(s, a)

Algorithm 5 Fitted Value Iteration

1. while not converged do
2. yi ← maxai (r(si, ai) + γE[Vϕ(s)])
3. Update ϕ to minimize Σ (Vϕ(si) − yi)2

Algorithm 6 Fitted Q-Iteration

1. while not converged do
2. for K times do
3. yi ← ri + γ maxa′ Qϕ(s′, a′)
4. Update ϕ to minimize Σ (Qϕ(si, ai) − yi)2

Algorithm 7 Online Q-Iteration

1. while interacting with environment do
2. Take action ai and observe (si, ai, ri, s′)
3. yi ← ri + γ maxa′ Qϕ(s′, a′)
4. Update ϕ ← ϕ − α∇ϕQϕ(si, ai)(Qϕ(si, ai) − yi)

Fitted Value Iteration

Tabular methods struggle with large state spaces (curse of dimensionality). To address this, I approximate value functions with neural networks. The value function is trained to

$$L(\phi) = \frac{1}{2} \left\| V_{\varphi}(s) - \max_{a} Q^{\pi}(s, a) \right\|^2$$

1) Fitted Q-Iteration

To avoid relying on transition dynamics, I learn Qϕ directly and approximate Vϕ(s) ≈ maxa Qϕ(s, a). This supports off- policy learning and reduces variance, though it lacks convergence guarantees for non-linear approximates. The training minimizes the **Bellman Error**:

$$\epsilon = \frac{1}{2}\mathbb{E}_{(s,a)\sim\beta}\left[h_\phi(s,a) - \left(r(s,a) + \gamma \max_{a'} Q_\phi(s',a')\right)^2\right]$$

Achieving $\epsilon = 0$ yields the optimal Q-function and policy. However, due to approximation and off-policy sampling, convergence is not guaranteed in general.

2) Online Q-Iteration

Instead of batch learning, I can apply online Q-learning, updating the Q-network with each new sample immediately:

This online, off-policy version enables continual learning with improved sample efficiency but shares the same stability concerns with non-linear function approximates

Value Function Learning Theory

A natural question that arises when exploring value-based methods is whether they converge, and if so, to what. To answer this, I introduce the Bellman backup operator B:

$$BV = \max_a r_a + \gamma T_a V$$

Where ra is the reward vector for action a and Ta is the state transition matrix. The fixed point V ∗ of this operator satisfies:

$$V^* = BV^* = \max_a r(s,a) + \gamma \mathbb{E}[V^*(s')]$$

This fixed point represents the optimal value function. In the tabular case, value iteration converges to V ∗ since B is a contraction under the l∞ norm. In contrast, the non-tabular case introduces challenges. Fit- ted value iteration applies B followed by a projection operator Π onto a function space **Ω** (e.g., neural networks):

$$\Pi V = \arg\min_{V \in \Omega} \frac{1}{2}\sum_s (V(s) - V(s))^2$$

While both B and Π are contractions in their respective norms, their composition is not, thus convergence is not guaranteed. Fitted Q-iteration uses the same principle: Q ← ΠBQ, and similarly lacks convergence guarantees due to the compound Operator not being a contraction. Moreover, online Q-iteration updates Q using single samples and a bootstrapped target dependent on the same network, leading to unstable updates.

Replay Buffers and Target Networks

To reduce the correlation in sequential samples, replay buffers B store transitions and allow batch sampling for up However, the bootstrapped target yi = r + γ maxa′ Qϕ(s , a) still depends on the current network parameters ϕ, leading to biased gradients. To address this, target networks are introduced. A separate parameter set ϕ′ is used to compute targets, typically updated to avoid relying on transition dynamics, I learn Qϕ directly and approximate Vϕ(s) ≈ maxa Qϕ(s, a). This supports off- policy learning and reduces variance, though it lacks convergence guarantees for non-linear approximators. The training minimizes the **Bellman Error**:

$$\phi' \leftarrow \tau\phi' + (1-\tau)\phi$$

Combining both replay buffers and target networks yields the Deep Q-Network (DQN) algorithm, which significantly improves stability and convergence in practice.

Unifying Q-Learning Variants

The Q-learning family can be generalized by three pro- cesses: data collection, target computation, and parameter updates. In fitted Q-iteration, updates are nested; in online Q-learning, all processes run synchronously; and in DQN, target updates occur at a slower pace than data collection and training. This abstraction helps us better understand and compare these algorithmic variants.

Inaccuracy in Q-Learning

Q-values are not necessarily accurate. The reason lies in the target value. Recall that the target value y is defined as yj = rj + γ maxa′ Qϕ′ (s′, a′). The max operation in the target is the main problem, because for two random variables X1 and

X2, E [max(X1, X2)] ≥ max (E[X1], E[X2]). Therefore, when the next Q-value.

Double Q-Learning

One might notice that $\max_{a'} Q_{\phi'}(s', a') = Q_{\phi'}(s', \arg\max_{a'} Q_{\phi'}(s', a'))$. Thus, if I somehow managed to decorrelate the error from the selected action and the error from the Q-function, I could eliminate the erroneous overestimation. To achieve this, I can use two different networks

$$Q_{\phi_A}(s, a) \leftarrow r + \gamma Q_{\phi_B}\left(s', \arg\max_{a'} Q_{\phi_A}(s', a')\right)$$

$$Q_{\phi_B}(s, a) \leftarrow r + \gamma Q_{\phi_A}\left(s', \arg\max_{a'} Q_{\phi_B}(s', a')\right)$$

By using the parameters of one network for action selection and the other for value estimation, I decor relate the errors, thereby reducing overestimation bias. In practice, I often use the current and target networks as the two networks. Instead of setting the target as

$$y = r + \gamma Q_{\phi'}\left(s', \arg\max_{a'} Q_{\phi'}(s', a')\right) ,$$

I use the current network to select the action and the target network to evaluate its value:

N-Step Return Estimator

In the original definition, the target is $y_{i,t} = r_{i,t} + Q_{\phi'}(s_{i,t+1}, a_{i,t+1})$, which heavily depends on the θ Q-value estimate. When the Q-value estimate is poor, learning stalls. To resolve this, I can use the N -step return trick as in the actor-critic algorithm. The idea is to leverage the bias- variance trade off by limiting the reward accumulation to N steps:

$$y_{i,t} = \sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{i,t'} + \gamma^N \max_{a_{i,t+N}} Q_{\phi'}(s_{i,t+N}, a_{i,t+N})$$

However, this introduces an on-policy dependency, as the trajectory of rewards is generated by a specific policy. This limits the ability to fully leverage off-policy data. To mitigate this, I can either:

- Ignore the mismatch (works well in practice),
- Dynamically adapt N to maintain on-policy data,
- Use importance sampling to reweight the returns appro- priately, as described by Munos et al. [2].

Algorithm 8 Deep Deterministic Policy Gradient (DDPG)

1. while training do
2. Take action ai, observe transition (si, ai, ri, s′), store in replay buffer B
3. Sample mini-batch {sj, aj, rj, s′ } from B
4. Compute target: yj = rj + γQϕ′ (sj, μθ′ (sj))
5. Update critic: $\phi \leftarrow \phi - \alpha \nabla \Sigma (Q(s, a) - y)^2$ ϕ j j
6. Update actor: $\theta \leftarrow \theta + \beta \Sigma \nabla Q(s, \mu(s))$
7. Update target networks

Q-Learning with Continuous Actions

In Q-learning, the implicit policy is defined as:

$$\pi'(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0, & \text{otherwise} \end{cases}$$

However, arg max is intractable for continuous action spaces. To resolve this:

Option 1: Sample-based Approximation

Use random sampling over a pre-defined distribution (e.g., uniform) and approximate:

$$\max_a Q(s, a) \approx \max\{Q(s, a_1), \ldots, Q(s, a_N)\}$$

This can be improved with techniques like the Cross-Entropy Method (CEM).

Option 2: Structured Q-function

Use a Q-function that is easy to optimize analytically. One such method is the Normalized Advantage Functions (NAF) proposed by Gu et al. [3].

Option 3: Learn an Argmax-er (DDPG)

Train a separate actor network μθ(s) to approximate the action that maximizes the Q-value:

$$\mu_\theta(s) \approx \arg\max_a Q_\phi(s, a)$$

The optimization becomes:

$$\theta \leftarrow \arg\max_\theta Q_\phi(s, \mu_\theta(s))$$

The target becomes:

$$y_i = r_i + \gamma Q_{\phi'}(s_i', \mu_\theta(s_i')) \approx r_i + \gamma Q_{\phi'}(s_i', \arg\max_{a'} Q_{\phi'}(s_i', a_i'))$$

4) A Simple ϵ Bound

Assume that πθ is deterministic, i.e., at = πθ(st). From imitation learning, I say πθ′ is ϵ-close to πθ if

$$\pi_{\theta'}(a_t \neq \pi_\theta(s_t) \mid s_t) \leq \epsilon.$$

Then, the new policy's state marginal is given by:

$$p_{\theta'}(s_t) = (1 - \epsilon)^t p_\theta(s_t) + \left[1 - (1 - \epsilon)^t\right] p_{\text{mistake}}(s_t).$$

I can bound the mismatch in state distributions as:

$$\begin{aligned} |p_{\theta'}(s_t) - p_\theta(s_t)| &= \left(1 - (1 - \epsilon)^t\right) |p_{\text{mistake}}(s_t) - p_\theta(s_t)| \\ &\leq 2\left(1 - (1 - \epsilon)^t\right) \\ &\leq 2\epsilon t. \end{aligned}$$

This bound is not tight but provides a first-order approximation. Now consider a more general (possibly stochastic) policy πθ. I define closeness as:

$$|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \leq \epsilon, \quad \forall s_t$$

I use the following lemma: if |pX (x) − pY (x)| = ϵ, then there exists a joint distribution p(x, y) such that p(x) = pX (x), p(y) = pY (y), and p(x = y) = 1 − ϵ. Applying this to πθ and πθ′ , I conclude that the probability they choose different actions is bounded by ϵ. Thus, the same

$$|p_{\theta'}(s_t) - p_\theta(s_t)| \leq 2\epsilon t.$$

Now, for any function f (st), I can write:

$$\begin{aligned} E_{p_{\theta'}(s_t)}[f(s_t)] &= \sum_{s_t} p_{\theta'}(s_t) f(s_t) \\ &= \sum_{s_t} (p_\theta(s_t) + (p_{\theta'}(s_t) - p_\theta(s_t))) f(s_t) \\ &= E_{p_\theta(s_t)}[f(s_t)] + \sum_{s_t} (p_{\theta'}(s_t) - p_\theta(s_t)) f(s_t) \\ &\geq E_{p_\theta(s_t)}[f(s_t)] - \sum_{s_t} |p_{\theta'}(s_t) - p_\theta(s_t)| \\ &\quad \cdot \max_{s_t} f(s_t) \\ &\geq E_{p_\theta(s_t)}[f(s_t)] - 2\epsilon t \cdot \max_{s_t} f(s_t). \end{aligned}$$

Now apply this to the policy improvement objective:

$$\sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)$$

$$\geq \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)$$

$$- \sum_t 2\epsilon C,$$

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \tag{10}$$

$$\frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \tag{11}$$

subject to

$$|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \leq \epsilon \tag{12}$$

A More Convenient Bound - KL Divergence

A tighter and more convenient constraint is provided by KL divergence. Using Pinsker's inequality:

$$|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \leq \sqrt{\frac{1}{2} D_{KL}(\pi_\theta(\cdot \mid s_t) \| \pi_\theta(\cdot \mid s_t))}.$$

I define

$$D_{KL}(p \| q) = \mathbb{E}_{x \sim p} \left[\log \frac{p(x)}{q(x)}\right].$$

Thus, the update becomes:

$$\theta' \leftarrow \arg\max_\theta \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \tag{13}$$

$$\frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \tag{14}$$

subject to

$$D_{KL}(\pi_{\theta'} \| \pi_\theta) \leq \epsilon \tag{15}$$

Can incorporate the constraint using a Lagrangian:

$$L(\theta', \lambda) = \sum_t \mathbb{E}_{s_t \sim p_\theta} \mathbb{E}_{a_t \sim \pi_\theta} \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)$$

$$- \lambda \left(D_{KL}(\pi_{\theta'} \| \pi_\theta) - \epsilon\right)$$

I then alternate between:

- Maximizing L with respect to θ′ (e.g., via gradient ascent),
- Updating λ via: λ ← λ + α(DKL − ϵ). This method is known as dual gradient descent.

First-Order Optimization via Taylor Expansion

$$\theta' \leftarrow \arg\max_\theta \nabla_\theta \bar{A}(\theta)^T (\theta' - \theta) \quad \text{s.t.} \quad D_{KL}(\pi_{\theta'} \| \pi_\theta) \leq \epsilon.$$

From policy gradients:

$$\nabla_\theta \bar{A}(\theta) = \sum_t \mathbb{E}_{s_t \sim p_\theta, a_t \sim \pi_\theta} \gamma^t \nabla_{\theta'} \log \pi_{\theta'}(a_t \mid s_t) A^{\pi_\theta}(s_t, a_t).$$

This leads to:

$$\vartheta' \leftarrow \arg\max_{\vartheta} \nabla_{\vartheta} J(\vartheta)^T (\vartheta' - \vartheta) \quad \text{s.t.} \quad D_{KL}(\pi_{\vartheta'} \| \pi_{\vartheta}) \leq \epsilon.$$

Contrast this with the standard gradient ascent step:

$$\vartheta' \leftarrow \arg\max_{\vartheta} \nabla_{\vartheta} J(\vartheta)^T (\vartheta' - \vartheta) \quad \text{s.t.} \quad \|\vartheta' - \vartheta\|^2 \leq \epsilon.$$

whose solution is:

$$\vartheta' \leftarrow \vartheta + \sqrt{\frac{\epsilon}{\|\nabla_{\vartheta} J(\vartheta)\|^2}} \nabla_{\vartheta} J(\vartheta).$$

This enforces a spherical constraint in parameter space, rather than in policy space, which is not ideal. I aim reflects the policy mismatch.

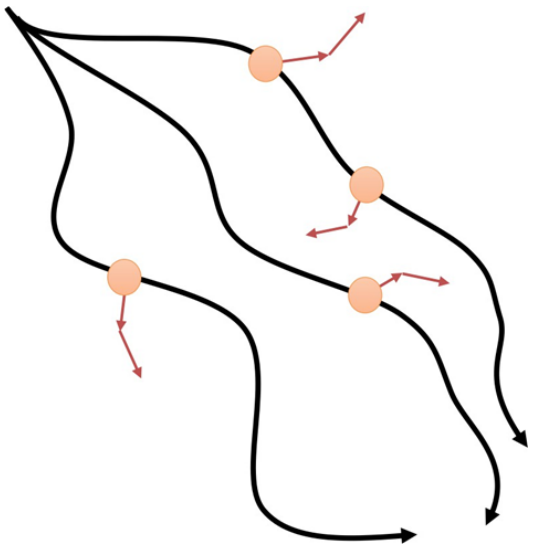

FIGURE 8. Dyna: synthetic rollouts from past states

short rollouts provide additional training data. The generalized Dyna algorithm is shown in Algorithm 10. Short synthetic rollouts minimize error accumulation while enhancing sample efficiency by reusing past states for training

Local and Global Models

In the context of LQR, a constrained control optimization problem can be reformulated into an unconstrained one, al- lowing us to minimize cumulative cost terms of the form:

$$\min_{u_1,\ldots,u_T} \sum_{t=1}^{T} c(x_t, u_t),$$

with each xt+1 = f (xt, ut). Solving this via backpropagation requires computing gradients of both the dynamics and cost functions with respect to state and control inputs. For complex systems, exact models are unavailable, so local approximations are used. By linearizing the nonlinear dynamics around a nominal trajectory, I obtain a local model where f (xt, ut) ≈ Atxt + Btut. These Jacobians, At = df and Bt = df , are estimated using samples from the system.

Local Models

Local models exploit LQR's structure by iteratively updat- ing a linear approximation around the current trajectory. If the true dynamics are stochastic and modeled as p(xt+1|xt, ut) = N (f (xt, ut), Σ), the system can still be approximated linearly within each iteration of iLQR. The resulting feedback control
law is given by:

$$u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t,$$

where xˆt, uˆt, Kt, and kt are outputs of the iLQR optimization. To introduce variability and avoid deterministic rollouts, I can inject Gaussian noise into the controller, yielding p(ut|xt) = N (Kt(xt − xˆt) + kt + uˆt, Σt), where Σt is often set to Q−1 . This approach allows for robust exploration during training. short rollouts provide additional training data. The generalized Dyna algorithm is shown in Algorithm 10. Short synthetic rollouts minimize error accumulation while enhancing sample efficiency by reusing past states for training.

LLocal and Global Models

In the context of LQR, a constrained control optimization problem can be reformulated into an unconstrained one, al- lowing us to minimize cumulative cost terms of the form: Dynamics can be modeled through Bayesian linear regres- sion, enabling uncertainty quantification. To maintain policy stability, the updated controller must remain close to previous ones. This is enforced by constraining the divergence be- tween trajectory distributions: $D_{KL}(p(\tau) \quad p(\bar{\tau})) \leq \epsilon$, ensuring smooth transitions across policy updates.

Guided Policy Search

Guided Policy Search (GPS) bridges local optimal control with global policy learning. The key idea is to use optimized local controllers (e.g., LQR policies) to generate trajectories, which serve as supervised data for training a global policy, typically a neural network. Since a single global policy may not fully capture the behaviors of all local controllers, GPS iteratively refines the local policies while encouraging align- ment with the global policy $\pi\theta$. This is done by modifying the cost to include a regularization term:

$$\tilde{c}_{k+1,i}(x_t, u_t) = c(x_t, u_t) + \lambda_{k+1} \log \pi_\theta(u_t|x_t),$$

which penalizes deviations from the global policy. The overall process is illustrated in Algorithm 11. Over time, both local and global policies are co-optimized to ensure consistency and performance.

Distillation

To efficiently generalize across tasks, reinforcement learning borrows the concept of knowledge distillation from supervised learning [4]. Instead of retaining a collection of specialized models, a single global model is trained to mimic the collective output of an ensemble by learning from their soft probability distributions. The soft targets are produced using:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)},$$

where T is a temperature parameter that smooths the output logits. In RL, this translates to policy distillation, where the global policy πAMN (a|s) is trained to match the behavior of multiple local expert policies πEi (a|s). The loss function is:

$$L = \sum_a \pi_E(a|s) \log \pi_{AMN}(a|s),$$

which enables the distilled policy to generalize across tasks efficiently. This strategy is foundational for scalable, multi- task RL and has been explored in works like [5], [6]. Similar ideas are also leveraged in Divide-and-Conquer RL, where the role of local LQR controllers is replaced with task-specific neural policies.

Combining Imitation and Reinforcement Learning

Imitation learning offers sample efficiency and stability but is limited by demonstration quality. Reinforcement learning can surpass demonstrations but suffers from exploration chal- lenges. A hybrid approach—pretraining a policy on demon- strations and fine-tuning with reinforcement learning—strikes a balance. However, care must be taken to avoid poor initial trajectories due to distribution shift.

Off-policy Reinforcement Learning

To prevent forgetting demonstrations during training, off- policy RL methods are advantageous, as they allow learning from any data source, including demonstrations reused across iterations. This approach maintains exposure to demonstration data while enabling the learned policy to outperform them, as it is not constrained to imitation. One strategy is to employ off-policy policy gradients, which use importance sampling to adjust for the mismatch between the behavior and target policies:

$$\nabla_\vartheta J(\vartheta) = \sum_{\tau \in D} \sum_{t=1}^{T} \nabla_\vartheta \log \pi_\vartheta(a_t \mid s_t) \left(\prod_{t'=1}^{t} \frac{\pi_\vartheta(a_{t'} \mid s_{t'})}{q(a_{t'} \mid s_{t'})} \right) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \right)$$

While policy gradients typically rely on on-policy data, importance sampling enables the use of off-policy data—such as demonstrations—by reweighting trajectories. The optimal importance sampling distribution for estimating Ep(x)[f (x)] minimizes variance when q(x) ∝ p(x)|f (x)|, suggesting that high-reward demonstrations bring us closer to this ideal. To model q(x), I can train a behavioral cloning policy πdemo, or if demonstrations come from multiple sources, define a fusion distribution:

$$q(x) = \frac{1}{M} \sum_i q_i(x)$$

Q-learning with Demonstrations

Since Q-learning is inherently off-policy, it can directly benefit from demonstrations without importance weighting. A practical method is to initialize the replay buffer with demon- stration data and proceed with standard Q-learning updates. This simple augmentation often yields improved performance, especially in sparse reward settings.

Imitation as Auxiliary Loss

Imitation learning maximizes the log-likelihood of expert actions:

$$\sum \log \pi_\vartheta(a|s)$$

To integrate this with reinforcement learning, a hybrid objective is used:

$$\mathrm{E}_{\pi_\vartheta}[r(s,a)] + \lambda \sum_{(s,a)\sim \mathrm{D}_{\mathrm{demo}}} \log \pi_\vartheta(a|s)$$

This formulation encourages the agent to imitate expert be- havior while still optimizing for long-term rewards.

Offline reinforcement learning

Unlike traditional RL, which relies on active environment interaction, offline RL aims to learn policies from static datasets—useful in scenarios where data collection is risky or costly, such as autonomous driving. Interestingly, offline RL can sometimes exceed the quality of its dataset by stitching together successful sub-trajectories. For example, Q-learning can converge to an optimal policy even from randomly col- lected data.

REFERENCES

[1]. S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in Proceedings of the fourteenth international conference on artificial intelligence and statistics, 2011, pp. 627–635.

[2]. R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare, "Safe and efficient off-policy reinforcement learning," in Advances in Neural Information Processing Systems, 2016, pp. 1054–1062.

[3]. S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q- learning with model-based acceleration," in International Conference on Machine Learning, 2016, pp. 2829–2838.

[4]. G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," arXiv preprint arXiv:1503.02531, 2015.

[5]. A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirk- patrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, "Policy distillation," arXiv preprint arXiv:1511.06295, 2015.

[6]. E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," arXiv preprint arXiv:1511.06342, 2015.

[7]. R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," in Advances in Neural Information Processing Systems, 2016, pp. 1109–1117.