Reinforcement Learning

Pradipta Biswas

Associate Professor

Indian Institute of Science

Pradipta@iisc.ac.in, https://cambum.net/PB/

Contents

Introduction

- Learning to interact with an environment
  - Robots, games, process control
  - With limited human training
  - Where the 'right thing' isn't obvious

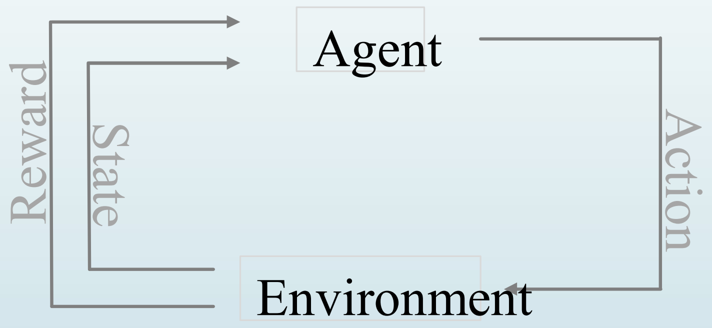
- Supervised Learning:
  - Goal: $f(x) = y$
  - Data: $[< x_1, y_1 >, \ldots, < x_n, y_n >]$

- Reinforcement Learning:
  - Goal:

    Maximize $\sum_{i=1}^{\infty} Reward(State_i, Action_i)$

  - Data:

    $Reward_i, State_{i+1} = Interact(State_i, Action_i)$



Supervised vs. Unsupervised Learning

- Supervised learning (classification)
  - Supervision: The training data (observations, measurements, etc.) are accompanied by labels indicating the class of the observations
  - New data is classified based on the training set
- Unsupervised learning (clustering)
  - The class labels of training data is unknown
  - Given a set of measurements, observations, etc. with the aim of establishing the existence of classes or clusters in the data

How Reinforcement Learning is Different

- Delayed Reward
- Agent chooses training data
- Explore vs Exploit (Life long learning)
- Very different terminology (can be confusing)
  - Reward
  - Utility
  - Policy
  - State-Action Mapping

Reinforcement Learning

- Reinforcement learning:
  - Still assume an MDP:
    - A set of states $s \in S$
    - A set of actions (per state) A
    - A model T(s,a,s')
    - A reward function R(s,a,s')
    - A discount factor γ (could be 1)
  - Still looking for a policy π(s)

  - New twist: don't know T or R
    - i.e. don't know which states are good or what the actions do
    - Must actually try actions and states out to learn

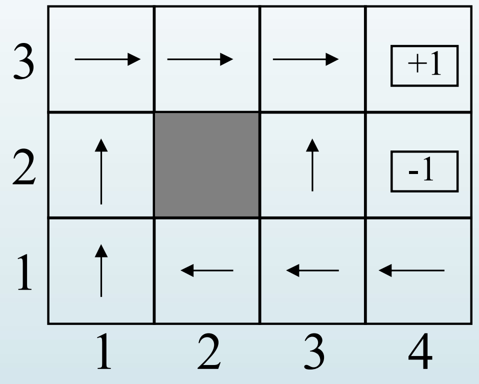
Reinforcement Learning – Set Up

M = 0.8 in direction you want to go

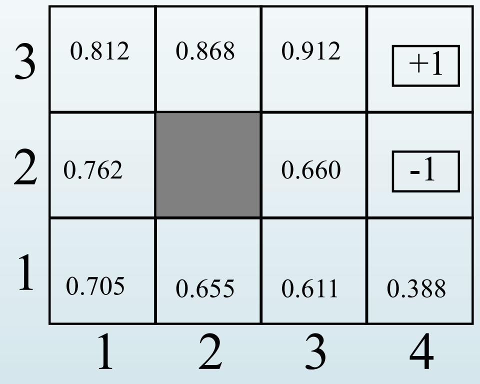
0.2 in perpendicular $<$ 0.1 left / 0.1 right

Policy: mapping from states to actions

An optimal policy for the stochastic environment:



utilities of states:



Environment $<$ Observable (accessible): percept identifies the state / Partially observable

Markov property: Transition probabilities depend on state only, not on the path to the state.
Markov decision problem (MDP).
Partially observable MDP (POMDP): percepts does not have enough info to identify transition probabilities.

Model-Based vs. Model-Free RL

- *Model based approach to RL*:
  - learn the MDP model, or an approximation of it
  - use it for policy evaluation or to find the optimal policy
- *Model free approach to RL*:
  - derive the optimal policy without explicitly learning the model
  - useful when model is difficult to represent and/or learn

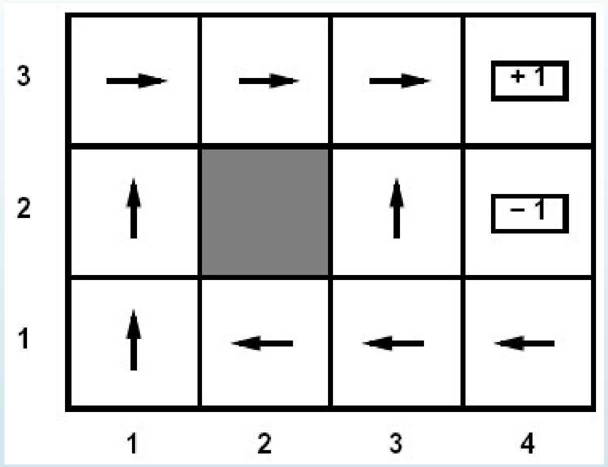
Passive vs. Active learning

- Passive learning
  - The agent has a fixed policy and tries to learn the utilities of states by observing the world go by
  - Analogous to policy evaluation
  - Often serves as a component of active learning algorithms
  - Often inspires active learning algorithms
- Active learning
  - The agent attempts to find an optimal (or at least good) policy by acting in the world
  - Analogous to solving the underlying MDP, but without first being given the MDP model

Passive Reinforcement Learning

Passive Learning

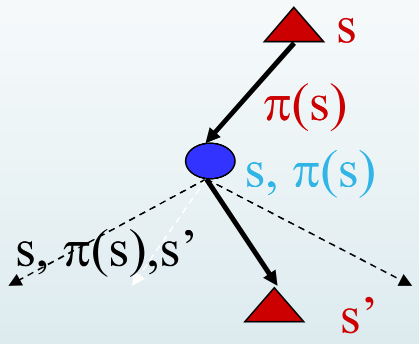
- Simplified task
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You are given a policy π(s)
  - Goal: learn the state values
  - … what policy evaluation did



- In this case:
  - Learner "along for the ride"
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - We'll get to the active case soon
  - This is NOT offline planning! You actually take actions in the world and see what happens…

Model-Based Learning

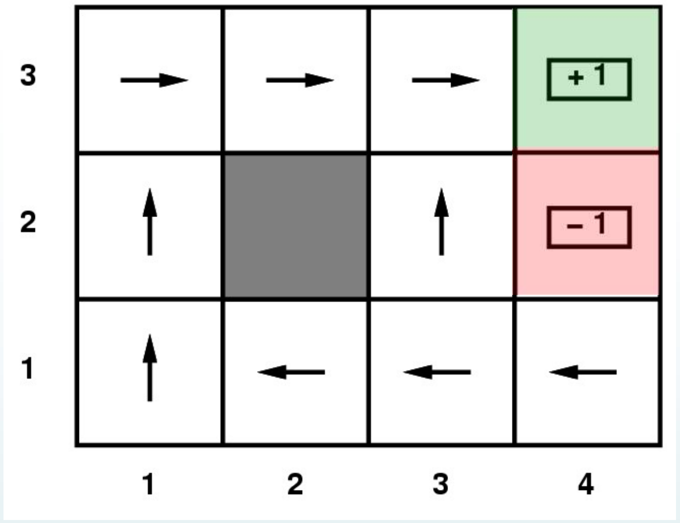
- Idea:
  - Learn the model empirically through experience
  - Solve for values as if the learned model were correct
- Simple empirical model learning
  - Count outcomes for each s,a
  - Normalize to give estimate of **T(s,a,s')**
  - Discover **R(s,a,s')** when we experience (s,a,s')
- Solving the MDP with the learned model
  - Iterative policy evaluation, for example



$$U_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma U_i^{\pi}(s')]$$

Passive RL

- Estimate $U^{\pi}(s)$
- Not given
  - transition matrix, nor
  - reward function!
- Follow the policy for many epochs giving training sequences.



$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,4)$ +1
$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (3,4)$ +1
$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2)$ -1

- Assume that after entering +1 or -1 state the agent enters zero reward terminal state
  - So we don't bother showing those transitions

Approach 1: Direct Estimation

- Direct estimation (also called Monte Carlo)
  - Estimate $U^{\pi}(s)$ as average total reward of epochs containing s (calculating from s to end of epoch)
- ***Reward to go*** of a state s

  the sum of the (discounted) rewards from that state until a terminal state is reached
- Key: use observed ***reward to go*** of the state as the direct evidence of the actual expected utility of that state
- Averaging the reward-to-go samples will converge to true value at state
- Convert the problem into supervised learning problem
  - Learn / Estimate utilities from the list of utilities in training sequences
  - Can average utilities for one sequence if the same state appears multiple times

Direct Estimation

- Converge very slowly to correct utilities values (requires more sequences than perhaps necessary)

- Doesn't exploit Bellman constraints on policy values

$$U^{\pi}(s) = R(s) + \beta \sum_{s'} T(s, \pi(s), s')\, U^{\pi}(s')$$

  - It is happy to consider value function estimates that violate this property badly.

How can we incorporate the Bellman constraints?

Approach 2: Adaptive Dynamic Programming (ADP)

- ADP is a model based approach
  - Follow the policy for awhile
  - Estimate transition model based on observations
  - Learn reward function
  - Use estimated model to compute utility of policy

$$U^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s')\, U^{\pi}(s')$$

learned

- How can we estimate transition model T(s,a,s')?
  - **Simply the fraction of times we see s' after taking a in state s.**

Adaptive DP (ADP)

Use the constraints (state transition probabilities) between states to speed learning.

Solve

$$U(i) = R(i) + \sum_j M_{ij} U(j)$$

= value determination.

No maximization over actions because agent is passive unlike in value iteration.

using DP

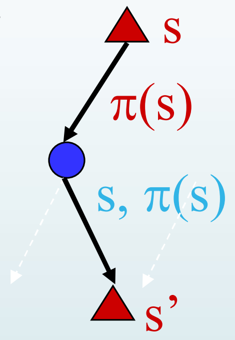
Too many linear equations to solve

One equation for each state →Large state space

e.g. Backgammon: 10^{50} equations in 10^{50} variables

Approach 3: Temporal-Difference Learning

- Use the observed transitions to adjust the values of the observed states so that they agree with the constraint equations.
- Learn from every experience!
  - Update U(s) each time we experience (s,a,s',r)
  - Likely s' will contribute updates more often
- Temporal difference learning
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!



Sample of U(s): $sample = R(s, \pi(s), s') + \gamma U^{\pi}(s')$

Update to U(s): $U^{\pi}(s) \leftarrow (1-\alpha)U^{\pi}(s) + (\alpha)sample$

Same update: $U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(sample - U^{\pi}(s))$

Temporal Difference (TD) Learning

Do ADP backups on a per move basis, not for the whole state space.

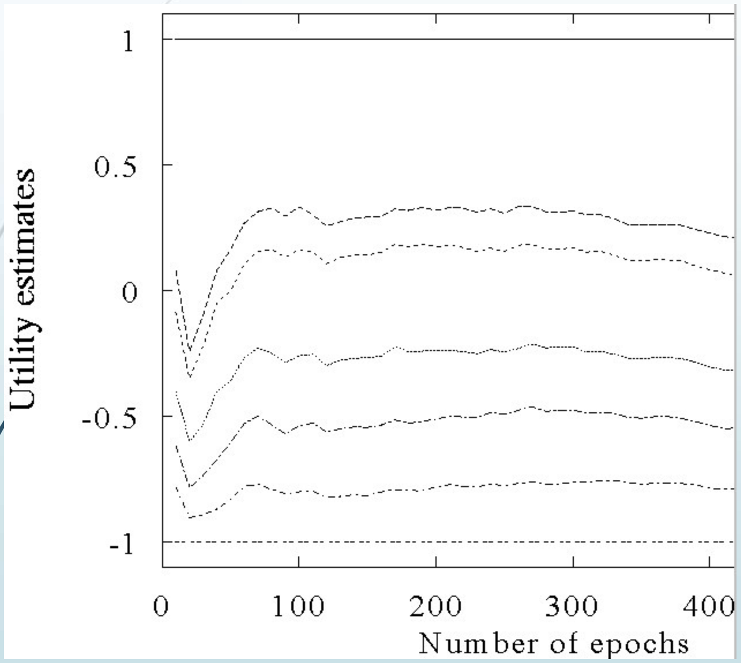
$$U(i) \leftarrow U(i) + \alpha[R(i) + U(j) - U(i)]$$

Thrm: Average value of U(i) converges to the correct value.

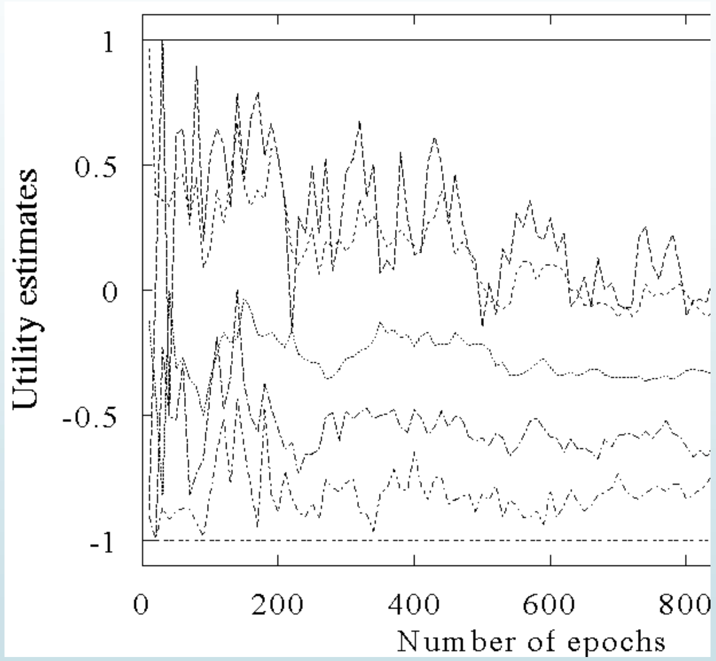
Thrm: If α is appropriately decreased as a function of times a state is visited ($\alpha=\alpha[N[i]]$), then U(i) itself converges to the correct value

Comparing Convergence between ADP and TD

ADP



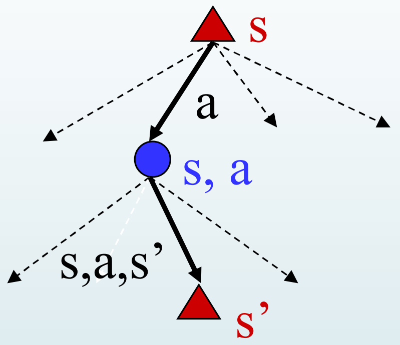
TD



- **Tradeoff:** requires more training experience (epochs) than ADP but much less computation per epoch
- Choice depends on relative cost of experience vs. computation

Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation
- We want to turn values into a (new) policy



$$\pi(s) = \arg\max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma U^*(s') \right]$$

- Idea: learn Q-values directly
- Makes action selection model-free too!

Passive RL: Comparisons

- Monte-Carlo Direct Estimation (model free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints
  - Converges slowly
- Adaptive Dynamic Programming (model based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of updates)
- Temporal Difference Learning (model free)
  - Update speed and implementation similiar to direct estimation
  - Partially exploits Bellman constraints---adjusts state to ‘agree’ with observed successor
    - Not ***all*** possible successors as in ADP
  - Convergence in between direct estimation and ADP

Active Reinforcement Learning

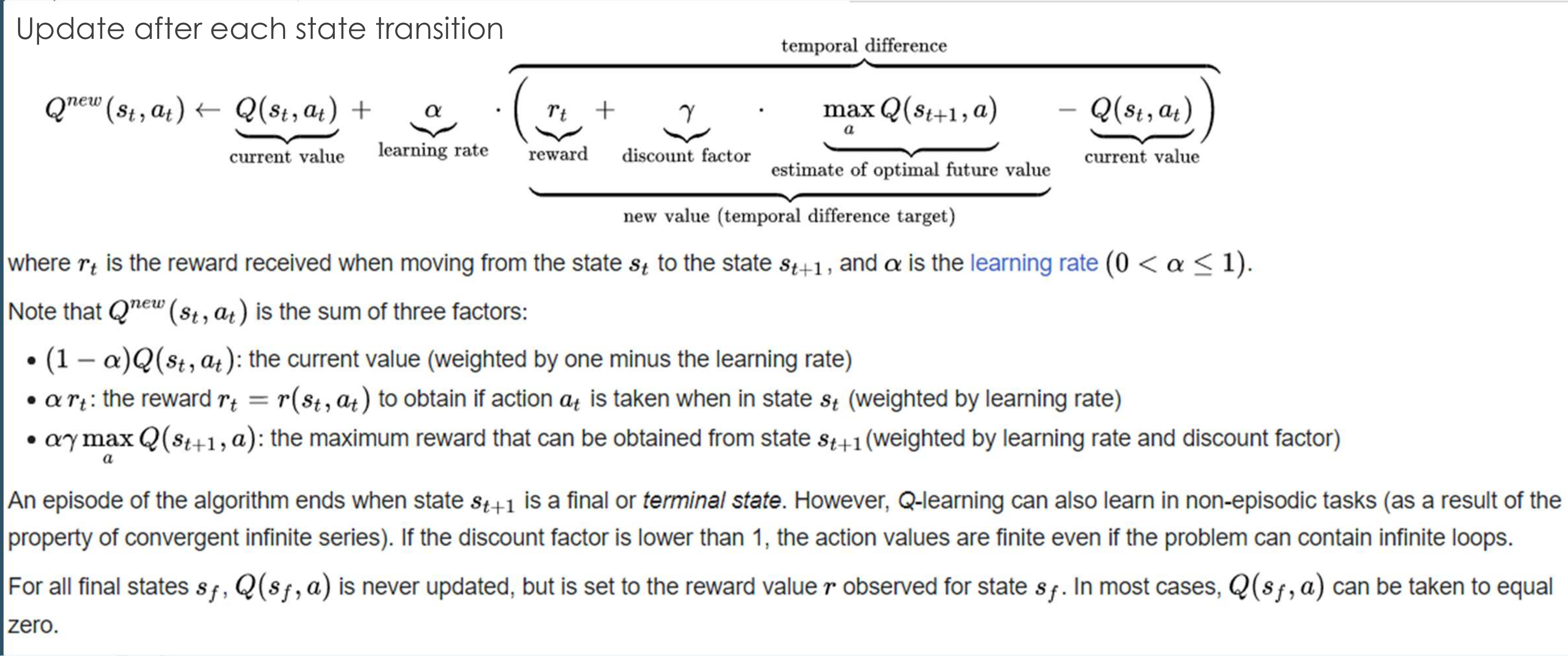
Difference with Passive RL

- A passive learning agent has a fixed policy that determines its behaviour. An active agent must decide what action to take
- An active agent requires outcome probabilities of ALL ACTIONS rather than for a fixed policy as used in passive RL
- An active agent EXPLORES the world
- Trade off between Exploration and Exploitation
  - Sticking to only known world ensures stability but may lead to sub-optimal solution
  - Exploring new opportunities lead to improve the present situation

Q-Learning

- Define a new function Q (a, s)
- Relationship with Utility: $U(s) = \max_a Q(a, s)$
- Constraint Equation for Equilibrium: $Q(a, S) = R(S) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s')$
- TD Update for Utility: $U^\pi(s) \leftarrow U^\pi(s) + a(R(s) + \gamma U^\pi(s') - U^\pi(s))$
- Q learning with TD update: $Q(a, S) \leftarrow Q(a, S) + a(R(S) + \gamma \max_{a'} Q(a', s') - Q(a, s)$

Q - Learning



Update after each state transition

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{current value}}\bigg)}^{\text{temporal difference}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $(0 < \alpha \leq 1)$.

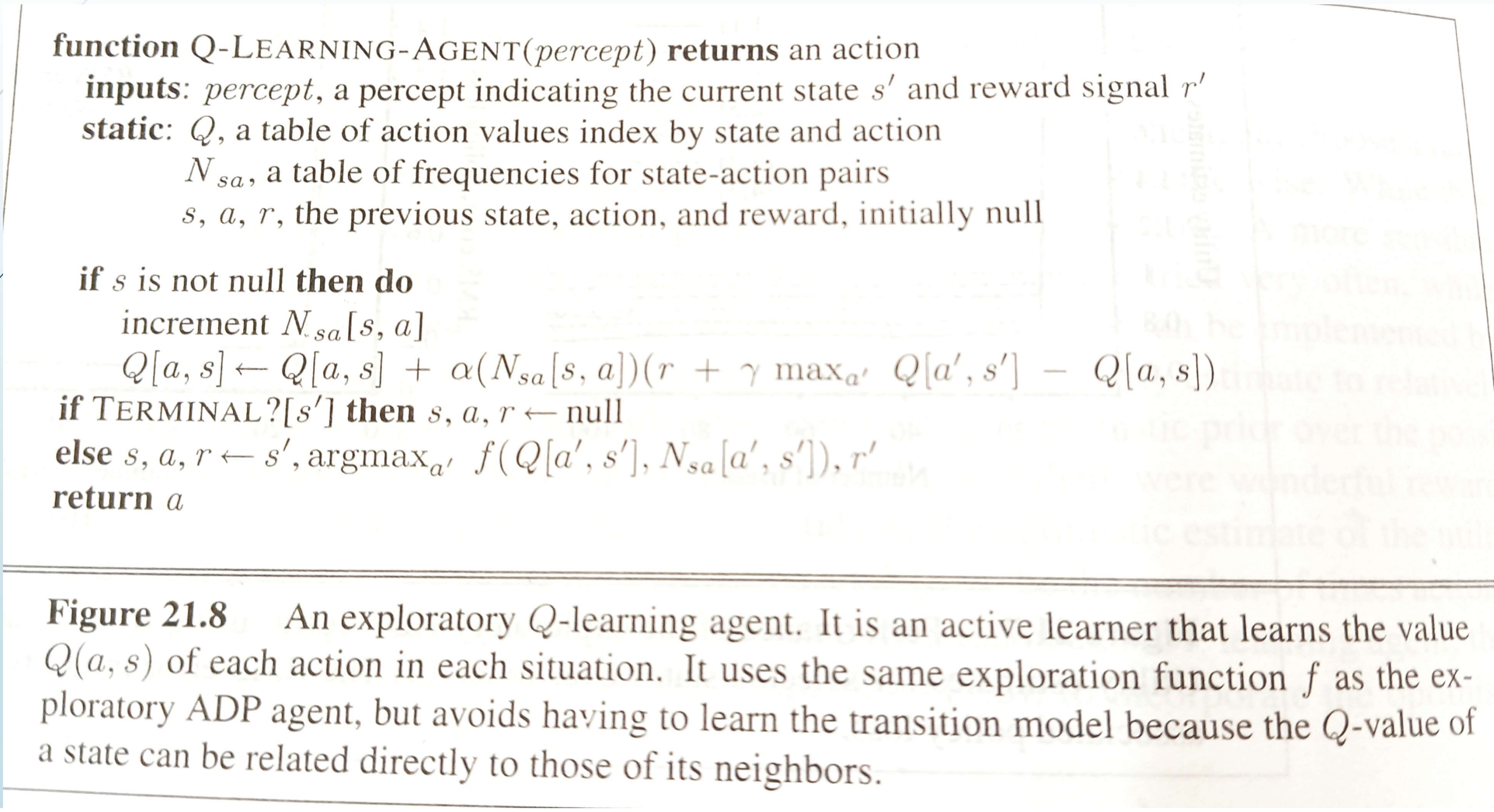
Note that $Q^{new}(s_t, a_t)$ is the sum of three factors:

- $(1-\alpha)Q(s_t,a_t)$: the current value (weighted by one minus the learning rate)
- $\alpha\, r_t$: the reward $r_t = r(s_t,a_t)$ to obtain if action a_t is taken when in state s_t (weighted by learning rate)
- $\alpha\gamma \max_a Q(s_{t+1},a)$: the maximum reward that can be obtained from state s_{t+1} (weighted by learning rate and discount factor)

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, Q-learning can also learn in non-episodic tasks (as a result of the property of convergent infinite series). If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f,a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f,a)$ can be taken to equal zero.

Q – Learning implementation



```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
  static: $Q$, a table of action values index by state and action
          $N_{sa}$, a table of frequencies for state-action pairs
          $s$, $a$, $r$, the previous state, action, and reward, initially null

  if $s$ is not null then do
     increment $N_{sa}[s, a]$
     $Q[a, s] \leftarrow Q[a, s] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[a', s'] - Q[a, s])$
  if TERMINAL?[$s'$] then $s, a, r \leftarrow$ null
  else $s, a, r \leftarrow s', \mathrm{argmax}_{a'}\ f(Q[a', s'], N_{sa}[a', s']), r'$
  return $a$
```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(a, s)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

Problem with Optimal Policy

Agent learns a model not the true environment !!

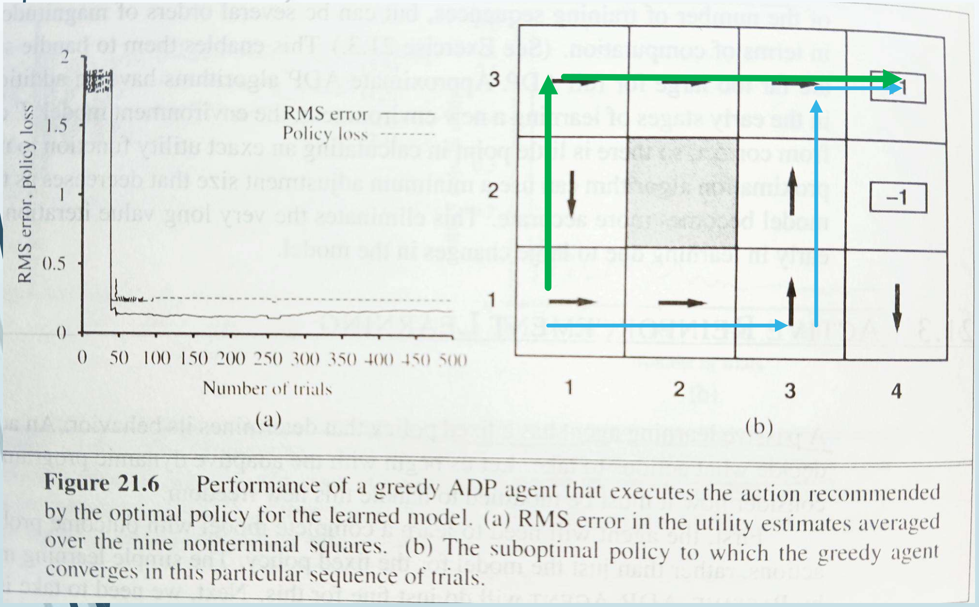


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

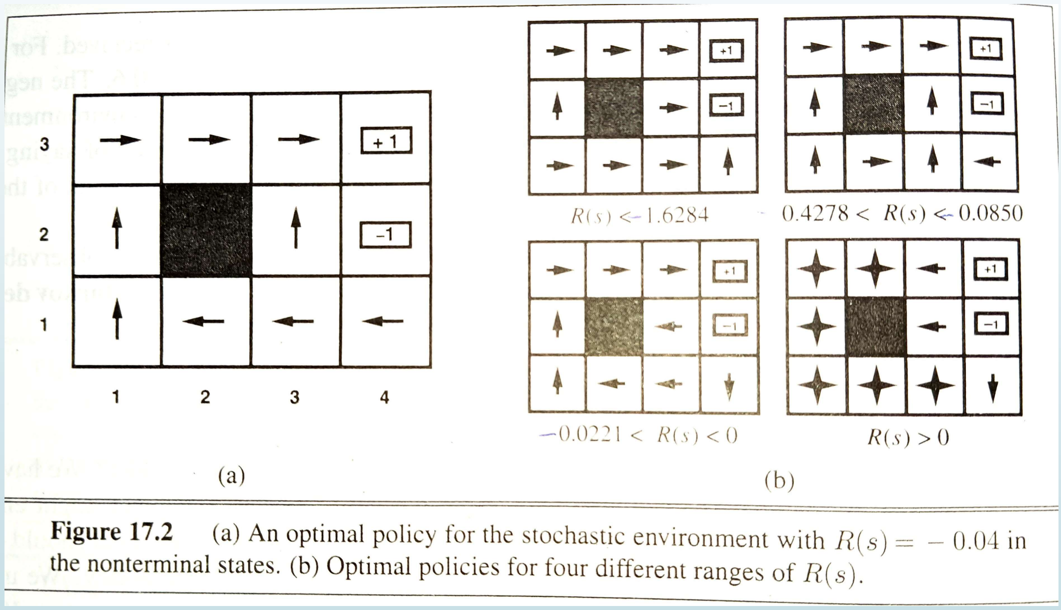


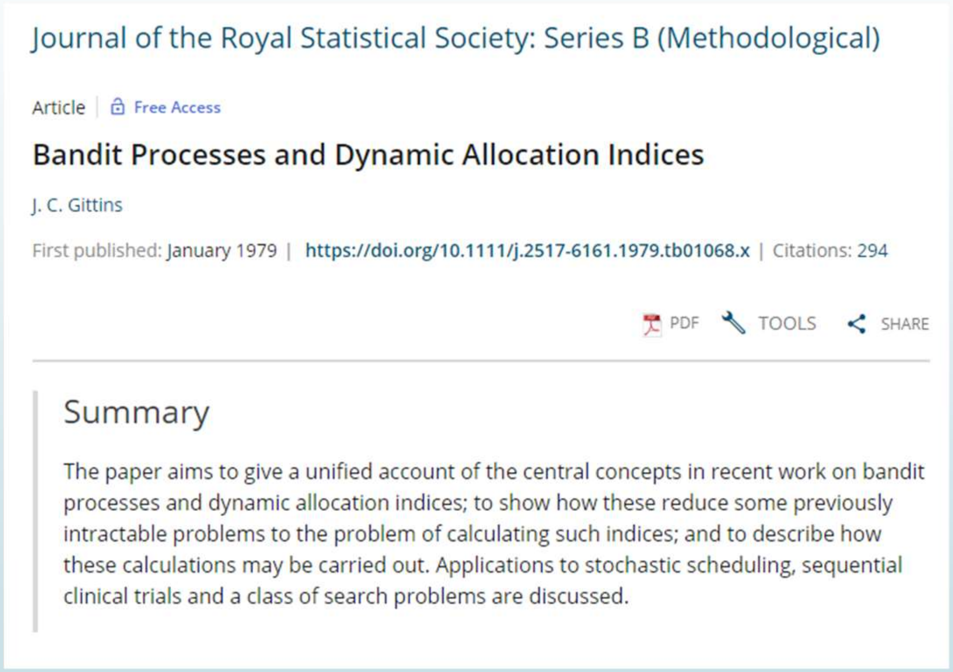
Figure 17.2 (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

Exploration Function

- Two new functions
- N(a, s) = How many times action a is executed at state s
- **Exploration function f (u, n)**
  - Increasing in u and decreasing in n
  - Greed is traded off with curiosity
  - $f(u, n) = \begin{cases} R^{+} & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$
    - where R+ is an optimistic estimate of the best possible reward obtainable in any state and Ne is a fixed parameter
    - Ensures each state-action pair will be tried at least Ne times

Quest for Optimal Exploration

- GLIE – Greedy in the Limit of Infinite Exploration
- Try each action in each state an unbounded number of times
- **Gittins index** is an effort to find optimal exploration policy
- The multi-armed bandit problem is a problem in which a fixed limited set of resources must be allocated between competing (alternative) choices in a way that maximizes their expected gain, when each choice's properties are only partially known at the time of allocation, and may become better understood as time passes or by allocating resources to the choice.


Journal of the Royal Statistical Society: Series B (Methodological)

Article | Free Access

Bandit Processes and Dynamic Allocation Indices

J. C. Gittins

First published: January 1979 | https://doi.org/10.1111/j.2517-6161.1979.tb01068.x | Citations: 294

Summary

The paper aims to give a unified account of the central concepts in recent work on bandit processes and dynamic allocation indices; to show how these reduce some previously intractable problems to the problem of calculating such indices; and to describe how these calculations may be carried out. Applications to stochastic scheduling, sequential clinical trials and a class of search problems are discussed.

Bandit Problem

Can we have an optimal exploration function

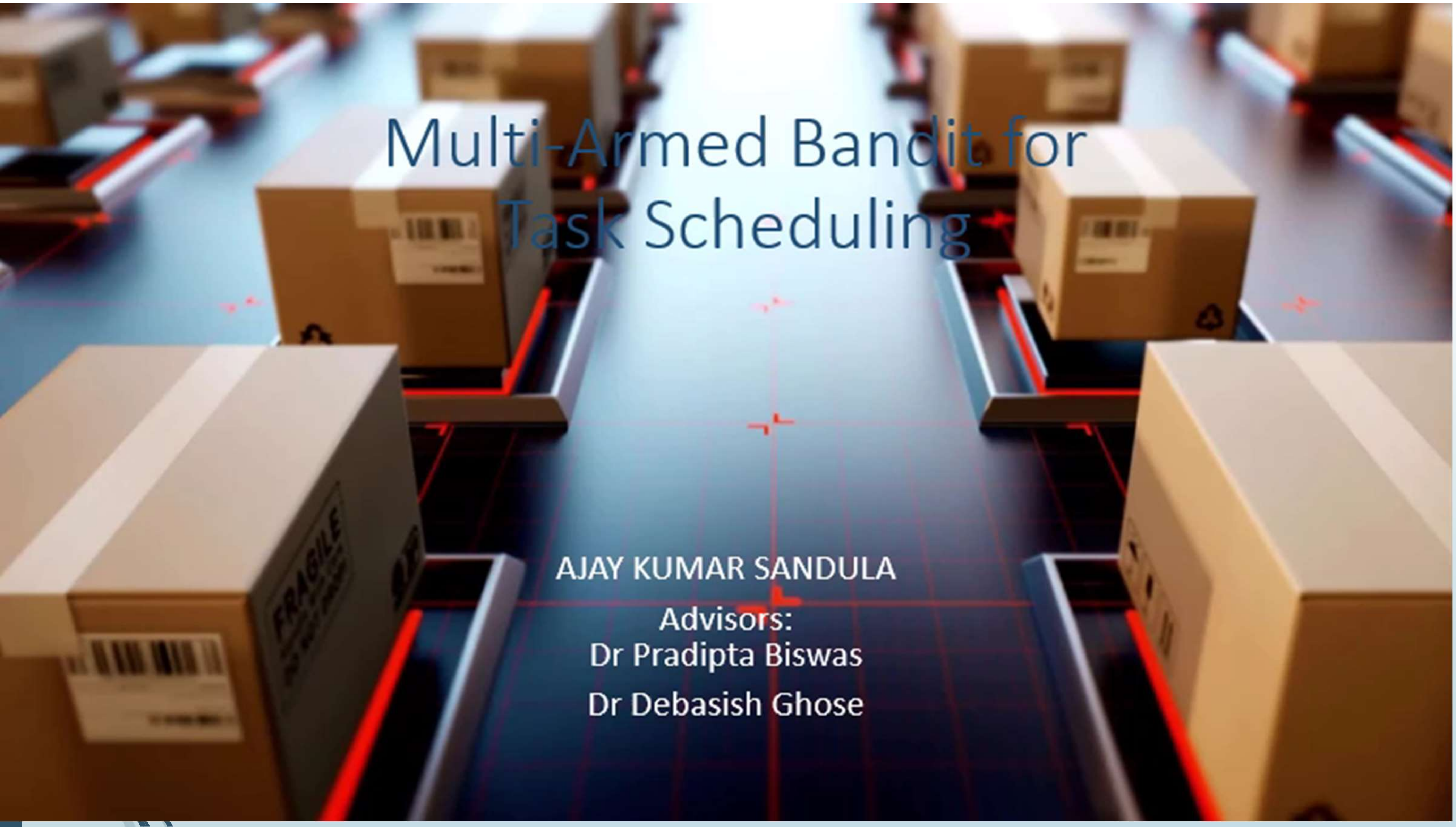
Virtual obstacle avoidance & heterogenous multi-robot coordination in mixed reality with a human in the loop

Ajay Kumar Sandula, Dr Pradipta Biswas, Arushi Khokhar, Dr Debasish Ghose

I3D Lab

RBCCPS Department

Indian Institute of Science, Bangalore

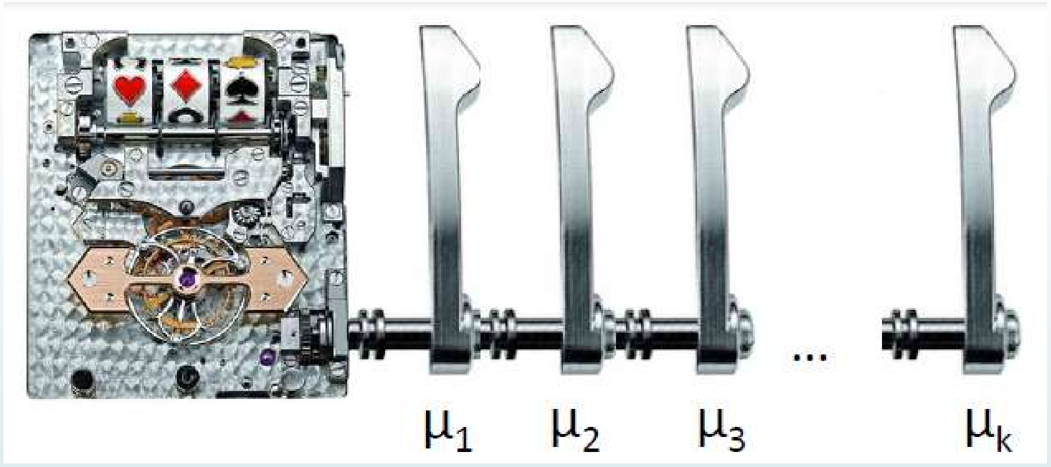


Multi-Armed Bandit for Task Scheduling

AJAY KUMAR SANDULA

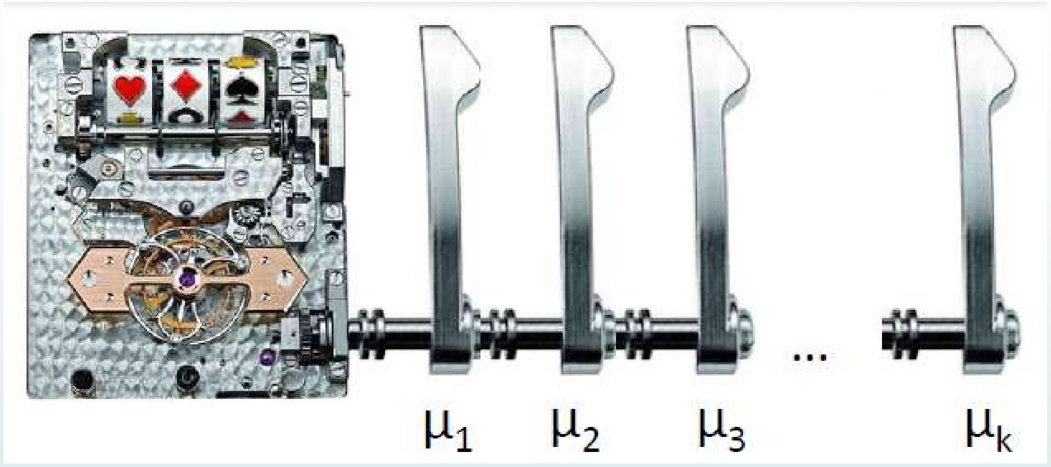
Advisors:
Dr Pradipta Biswas
Dr Debasish Ghose

k-Armed Bandit



- **Each arm *a***
  - **Wins** (reward=**1**) with fixed (unknown) prob. $\boldsymbol{\mu_a}$
  - **Loses** (reward=**0**) with fixed (unknown) prob. $\boldsymbol{1-\mu_a}$
- All draws are independent given $\boldsymbol{\mu_1} \ldots \boldsymbol{\mu_k}$
- **How to pull arms to maximize total reward?**

k-Armed Bandit



- Each **fixed base robot / service station** is a **bandit**
- Each **mobile robot** is an **arm**
- **We want to estimate the arm's probability of winning μ_a (task probability)**
- Every time we pull an arm we do an 'experiment'

Exploration vs. Exploitation

- We need to trade off **exploration** (gathering data about arm payoffs) and **exploitation** (making decisions based on data already gathered)

- **Exploration:** Pull an arm we never pulled before
- **Exploitation:** Pull an arm $\boldsymbol{a}$ for which we currently have the highest estimate of $\boldsymbol{\mu_a}$

Stochastic k-Armed Bandit

The setting:

- Set of ***k*** choices (arms)
- Each choice ***a*** is tied to a probability distribution $\boldsymbol{P_a}$ with average reward/payoff $\boldsymbol{\mu_a}$ (between [0, 1])
- We play the game for ***T*** rounds
- For each round ***t***:
  - **(1)** We pick some arm ***j***
  - **(2)** We win reward $\boldsymbol{X_t}$ drawn from $\boldsymbol{P_j}$
    - Note reward is independent of previous draws
- **Our goal is to maximize $\sum_{t=1}^{T} X_t$**
- **We don't know μ_a!** But every time we pull some arm ***a*** we get to learn a bit about $\boldsymbol{\mu_a}$

Epsilon-greedy Policy

- At each time step, the agent selects an action
- The agent follows the greedy strategy with probability 1 – epsilon
- The agent selects a random action with probability epsilon
- With Q-learning, the greedy strategy is the action a that maximises Q given S_{t+1}

```
For t=1:T
    Set $\varepsilon_t = O(1/t)$
    With prob. $\varepsilon_t$: Explore by picking an arm chosen uniformly at random
    With prob. $1 - \varepsilon_t$: Exploit by picking an arm with highest empirical mean payoff
```

Theorem **[Auer et al. '02]**

For suitable choice of ε_t it holds that $R_T = O(k \log T) \Rightarrow \frac{R_T}{T} = O\left(\frac{k \log T}{T}\right) \to 0$

Issues with Epsilon Greedy

- **“Not elegant”**: Algorithm explicitly distinguishes between exploration and exploitation
- Exploration makes **suboptimal choices** (since it picks any arm with equal likelihood)

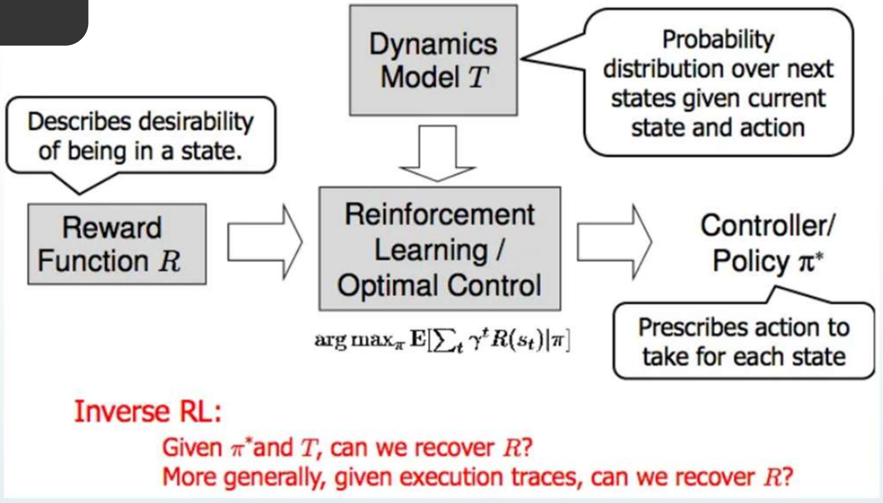
- **Idea:** When exploring/exploiting we need to **compare** arms
- Confidence interval based selection

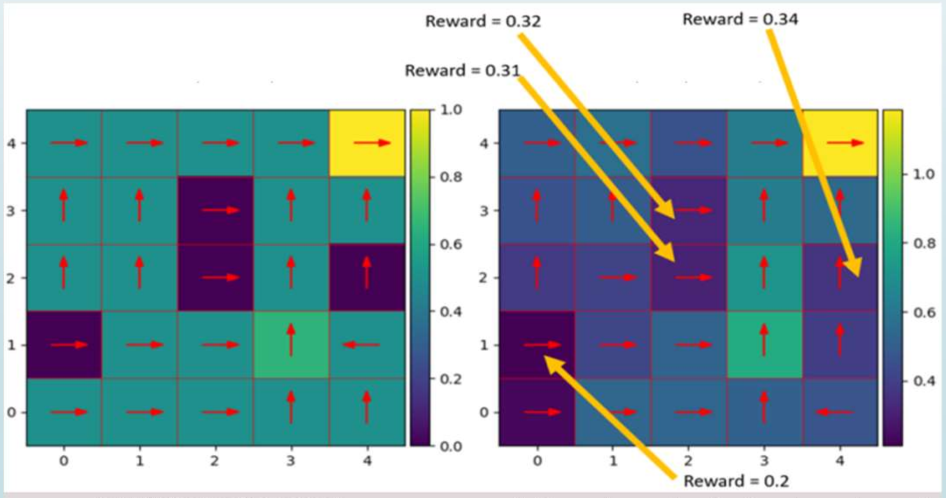
Deep Q-Networks (DQN)

- It is common to use a function approximator Q(s, a; θ) to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Uses epsilon-greedy policy to select actions
- We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

Inverse Reinforcement Learning

- In RL, it is necessary to manually tweak the rewards until the desired behaviour is observed.
- A better way is to learn the rewards by observing expert demonstrations (or policy).
- Thus IRL learns the underlying reward distribution using expert demonstrations/trajectories (human demonstrations).
- Expert policy π^* and transition probability matrix T is given from expert demonstrations (or can be obtained using supervised learning, etc.)
- IRL is then implemented using π^* and T to learn the reward distribution.
- For a robotic agent (autonomous navigation) or hand motion (for pointing task) it is easier to estimate the initial policy and transition matrix. Challenge is to estimate initial policy for eye gaze movement which has high uncertainty.
- Recently sampling-based approaches being used to obtain reward distribution using IRL for tasks with high uncertainty.

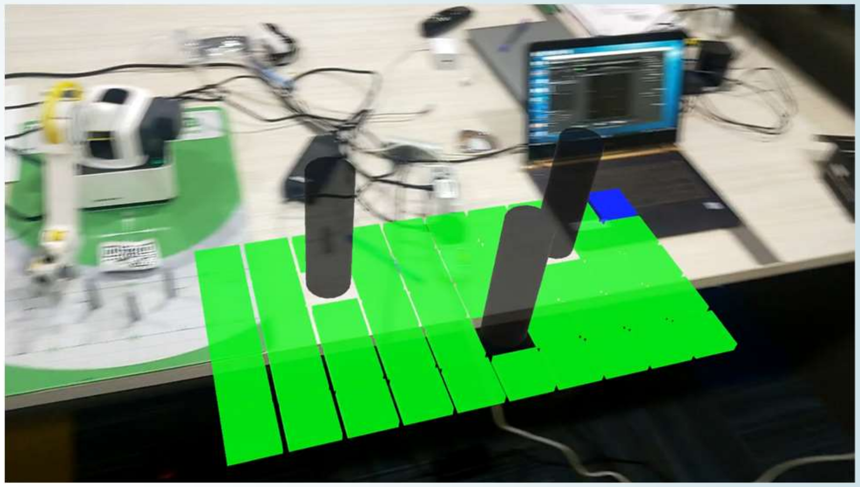

Dynamics Model T
Probability distribution over next states given current state and action
Describes desirability of being in a state.
Reward Function R
Reinforcement Learning / Optimal Control
Controller/ Policy π^*
$\arg\max_\pi \mathbf{E}[\sum_t \gamma^t R(s_t)|\pi]$
Prescribes action to take for each state
Inverse RL:
Given π^* and T, can we recover R?
More generally, given execution traces, can we recover R?

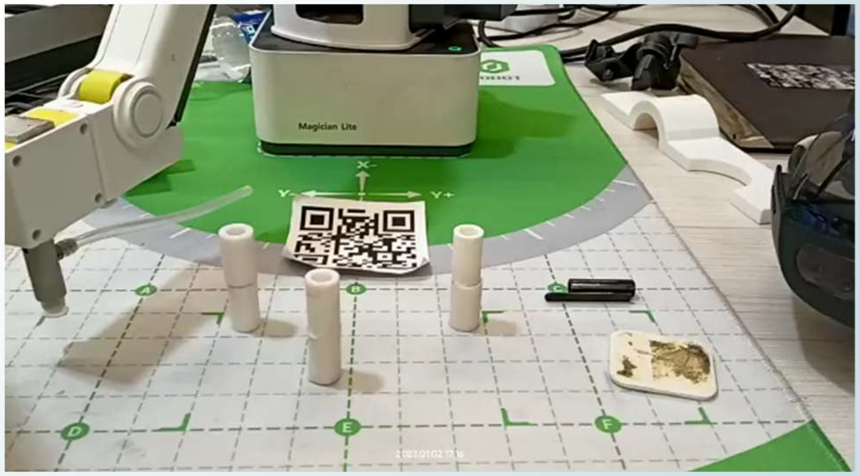


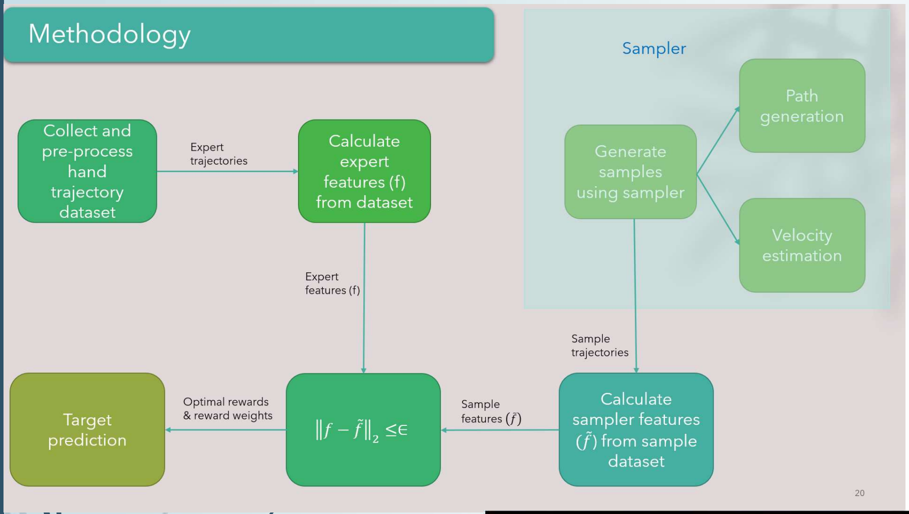
(a) Initial policy from expert (b) learned reward, policy from learned reward

Implementation of SMEIRL in 3D for Fixed Base Robot

- Robot workspace: 100x200x100mm
- 5x10x5 grid
- 150 Expert trajectories were collected, 120 for training, 30 for testing. Manually moving the robotic arm through hand, end-effector tracked by motion capture system.
- Velocity feature was used.
- SMEIRL was implemented.
- Given a start and target location, A-star was implemented using the learned rewards.
- Robot with no sensors, learns from human demonstrations and avoid obstacles.







Methodology

Sampler

Collect and pre-process hand trajectory dataset

Expert trajectories

Calculate expert features (f) from dataset

Generate samples using sampler

Path generation

Velocity estimation

Expert features (f)

Sample trajectories

Target prediction

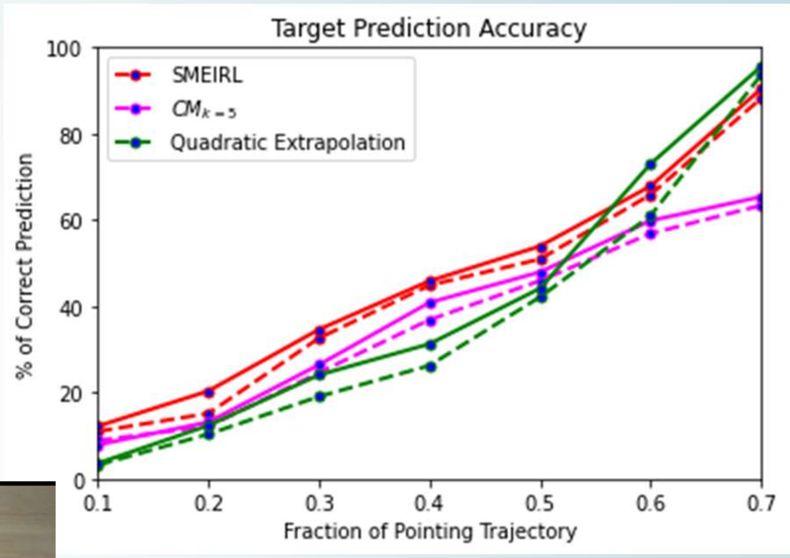
Optimal rewards & reward weights

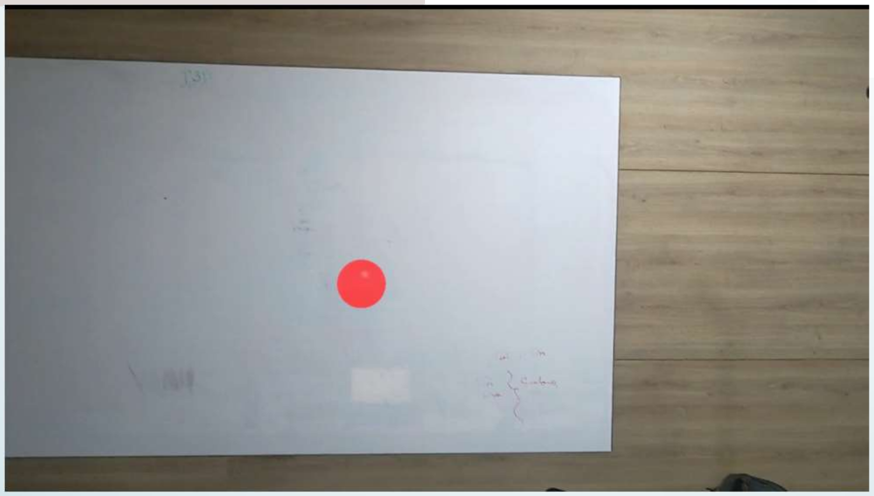
$\|f - \tilde{f}\|_2 \leq \epsilon$

Sample features ($\tilde{f}$)

Calculate sampler features ($\tilde{f}$) from sample dataset

20





Hand Movement Prediction

Summary

- Introduction to Reinforcement Learning
- Different Types
  - Passive vs Active
  - Model based vs Model Free
- Concept of Temporal Difference and Q Learning
- Exploration vs Exploitation – Multi Arm Bandit Problem
- Introducing advanced topics – Deep Q-Network and Inverse Reinforcement Learning
- Demonstrations of Multi Arm Bandit in Warehouse Simulation and IRL for Trajectory Prediction