

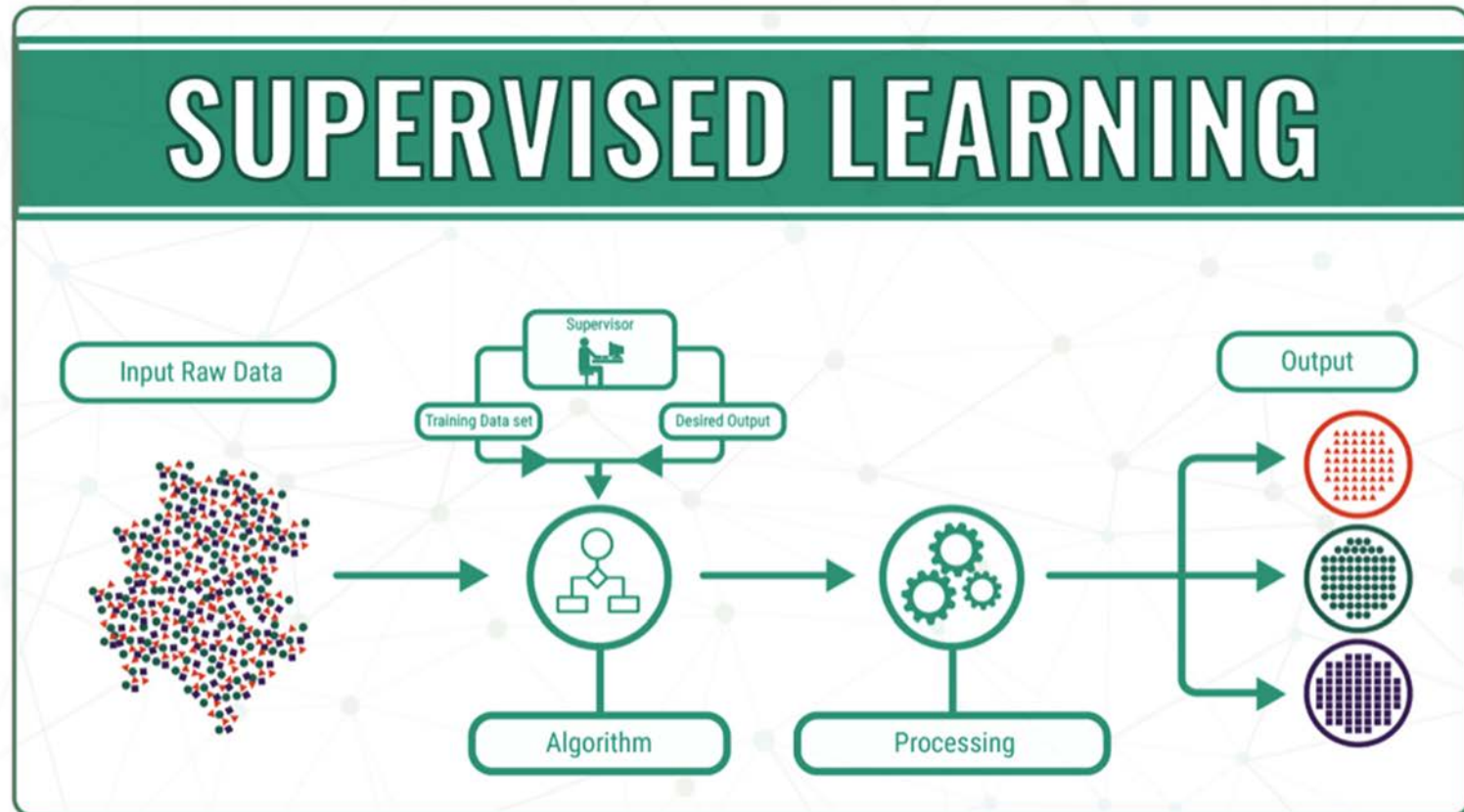
Reinforcement Learning Application Perspective

Presented by:

Siavash Fakhimi Derakhshan

TYPES OF MACHINE LEARNING

- Supervised learning
- Unsupervised learning
- Reinforcement learning

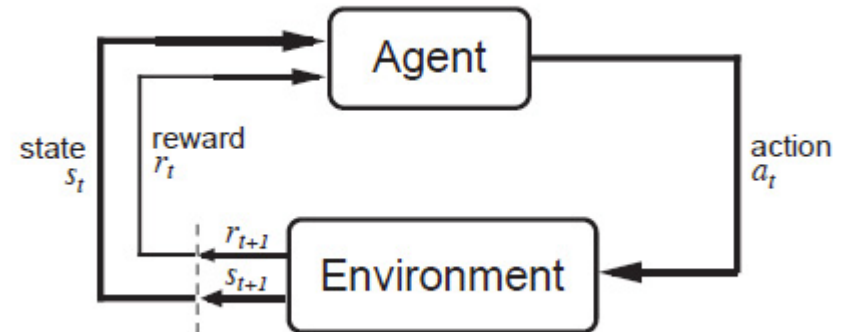


OUTLINE

- Defining an RL problem
 - Markov Decision Processes
- Solving an RL problem
 - Dynamic Programming
 - Monte Carlo methods
 - Temporal-Difference learning
- Application Perspective
 - Continuous state
 - Designing rewards
 - Adjust initial policy or initial value function
 - Reinforcement Learning and Control
 - Autonomous helicopter flight via reinforcement learning

MARKOV DECISION PROCESS (MDP)

- Set of states S , set of actions A , initial state S_0
- transition model $P(s,a,s')$
 - $P([1,1], \text{up}, [1,2]) = 0.8$
- reward function $r(s)$
 - $r([4,3]) = +1$
- goal: maximize cumulative reward in the long run
- policy: mapping from S to A
 - $\pi(s)$ or $\pi(s,a)$ (deterministic vs. stochastic)



WHAT YOU KNOW MATTERS

- Do you know your environment?
 - The effects of actions
 - The rewards
- If yes, you can use Dynamic Programming
 - More like planning than learning
 - *Value Iteration* and *Policy Iteration*
- If no, you can use Reinforcement Learning (RL)
 - Acting and observing in the environment
 - how to change the policy based on experience
 - how to explore the environment

ROBOT IN A ROOM

			+1
			-1
START			

actions: UP, DOWN, LEFT, RIGHT

UP

80%

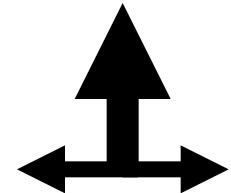
10%

10%

move UP

move LEFT

move RIGHT



- reward +1 at [4,3], -1 at [4,2]
- reward -0.04 for each step

what's the strategy to achieve max reward?

COMPUTING RETURN FROM REWARDS

- episodic (vs. continuing) tasks
 - “game over” after N steps
 - optimal policy depends on N ; harder to analyze
- additive rewards
 - $V(s_0, s_1, \dots) = r(s_0) + r(s_1) + r(s_2) + \dots$
 - infinite value for continuing tasks
- discounted rewards
 - $V(s_0, s_1, \dots) = r(s_0) + \gamma r(s_1) + \gamma^2 r(s_2) + \dots$
 - value bounded if rewards bounded

VALUE FUNCTIONS

- state value function: $V^\pi(s)$
 - expected return when starting in s and following π
- state-action value function: $Q^\pi(s,a)$
 - expected return when starting in s , performing a , and following π
- useful for finding the optimal policy
 - can estimate from experience
 - pick the best action using $Q^\pi(s,a)$

- Bellman equation

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^\pi(s')] = \sum_a \pi(s, a) Q^\pi(s, a)$$

OPTIMAL VALUE FUNCTIONS

- there's a set of *optimal* policies
 - V^π defines partial ordering on policies
 - they share the same optimal value function

$$V^*(s) = \max_{\pi} V^\pi(s)$$

- Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^*(s')]$$

- having $Q^*(s,a)$ makes it even simpler

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

TYPES OF REINFORCEMENT LEARNING

- Search-based: evolution directly on a policy
 - E.g. genetic algorithms
- Model-based: build a model of the environment
 - Then you can use dynamic programming
- Model-free: learn a policy without any model
 - Temporal difference methods (TD)
 - Q-learning
 - Actor-critic learning

OUTLINE

- Defining an RL problem
 - Markov Decision Processes
- Solving an RL problem
 - Dynamic Programming
 - Monte Carlo methods
 - Temporal-Difference learning
- Application Perspective
 - Continuous state
 - Reinforcement Learning and Control
 - Autonomous helicopter flight via reinforcement learning

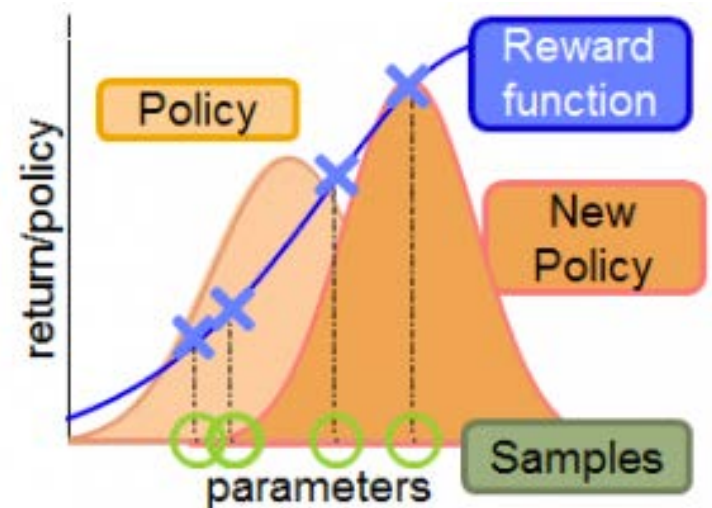
DYNAMIC PROGRAMMING

- main idea

- use value functions to structure the search for good policies
- need a perfect model of the environment

- two main components

- policy evaluation: compute V^π from π
- policy improvement: improve π based on V^π



POLICY EVALUATION/IMPROVEMENT

- policy evaluation: $\pi \rightarrow V^\pi$

- Bellman eqn's define a system of n eqn's
- could solve, but will use iterative version

$$V_{k+1}(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V_k(s')]$$

- start with an arbitrary value function V_0 , iterate until V_k converges

- policy improvement: $V^\pi \rightarrow \pi'$

$$\begin{aligned} \pi'(s) &= \arg \max_a Q^\pi(s, a) \\ &= \arg \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^\pi(s')] \end{aligned}$$

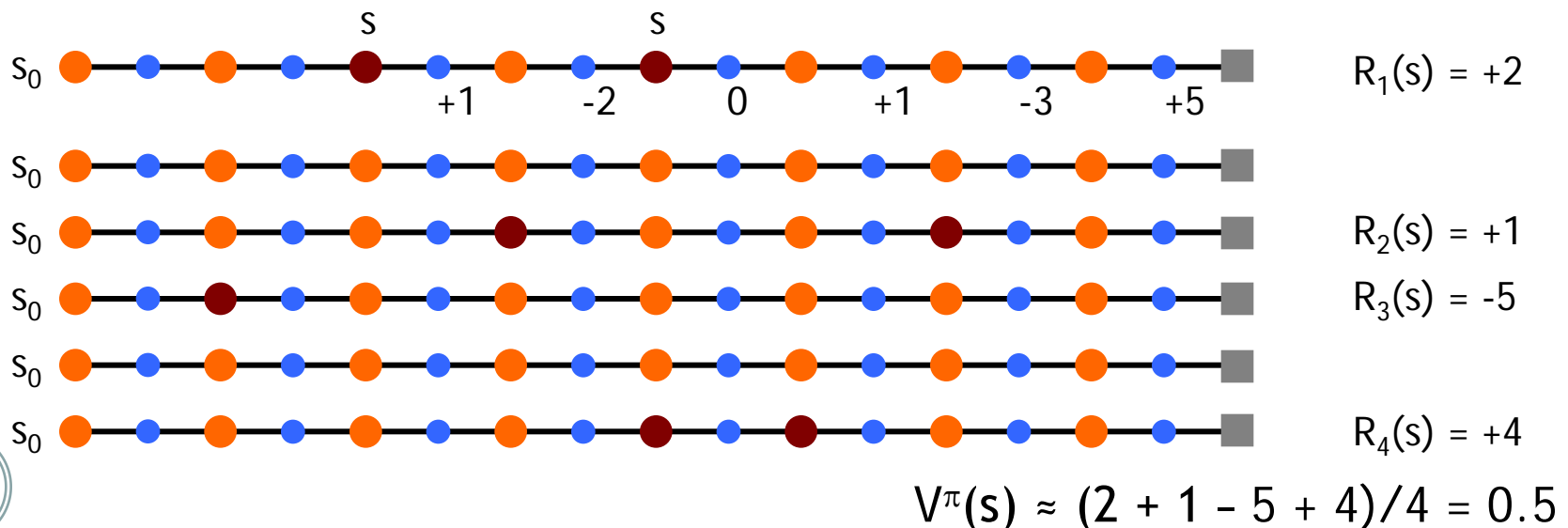
- π' either strictly better than π , or π' is optimal (if $\pi = \pi'$)

USING DYNAMIC PROGRAMMING

- need complete model of the environment and rewards
 - robot in a room
 - state space, action space, transition model
- can we use DP to solve
 - robot in a room?
 - helicopter?

MONTE CARLO POLICY EVALUATION

- want to estimate $V^\pi(s)$
 - = expected return starting from s and following π
 - estimate as average of observed returns in state s
- first-visit MC
 - average returns following the first visit to state s



MAINTAINING EXPLORATION

- deterministic/greedy policy won't explore all actions
 - don't know anything about the environment at the beginning
 - need to try all actions to find the optimal one
- maintain exploration
 - use *soft* policies instead: $\pi(s,a) > 0$ (for all s,a)
- ϵ -greedy policy
 - with probability $1-\epsilon$ perform the optimal/greedy action
 - with probability ϵ perform a random action
 - will keep exploring the environment
 - slowly move it towards greedy policy: $\epsilon \rightarrow 0$

TEMPORAL DIFFERENCE LEARNING

- combines ideas from MC and DP
 - like MC: learn directly from experience (don't need a model)
 - like DP: learn from values of successors
 - works for continuous tasks, usually faster than MC

- constant-alpha MC:

- have to wait until the end of episode to update

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$



- simplest TD

- update after every step, based on the successor

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$



MC VS. TD

- observed the following 8 episodes:

A - 0, B - 0

B - 1

B - 1

B - 1

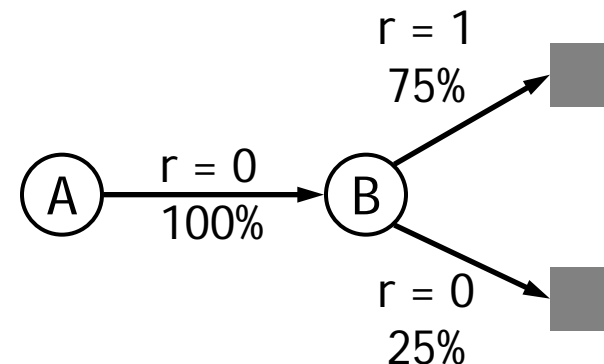
B - 1

B - 1

B - 1

B - 0

- MC and TD agree on $V(B) = 3/4$
- MC: $V(A) = 0$
 - converges to values that minimize the error on training data
- TD: $V(A) = 3/4$
 - converges to ML estimate of the Markov process



Q-LEARNING

- before: on-policy algorithms
 - start with a random policy, iteratively improve
 - converge to optimal

- Q-learning: off-policy
 - use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- Q directly approximates Q^* (Bellman optimality eqn)
- independent of the policy being followed
- only requirement: keep updating each (s,a) pair

- Sarsa

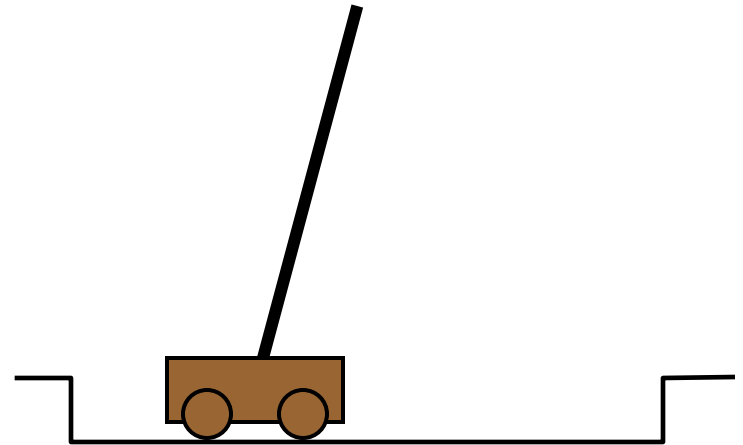
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

OUTLINE

- Defining an RL problem
 - Markov Decision Processes
- Solving an RL problem
 - Dynamic Programming
 - Monte Carlo methods
 - Temporal-Difference learning
- Application Perspective
 - Continuous state
 - Designing rewards
 - Adjust initial policy or initial value function
 - Reinforcement Learning and Control
 - Autonomous helicopter flight via reinforcement learning

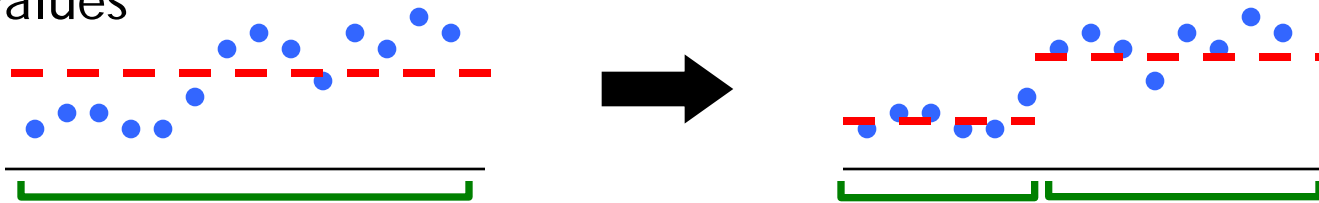
STATE REPRESENTATION

- pole-balancing
 - move car left/right to keep the pole balanced
- state representation
 - position and velocity of car
 - angle and angular velocity of pole
- what about *Markov property*?
 - would need more info
 - noise in sensors, temperature, bending of pole
- solution
 - coarse discretization of 4 state variables
 - left, center, right
 - totally non-Markov, but still works

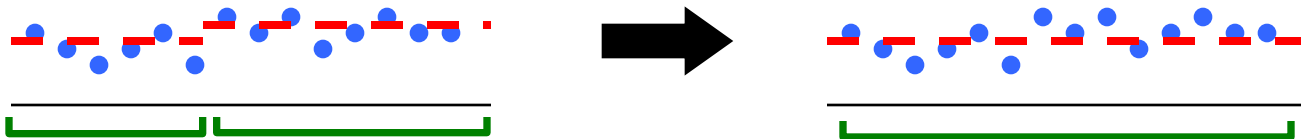


SPLITTING AND AGGREGATION

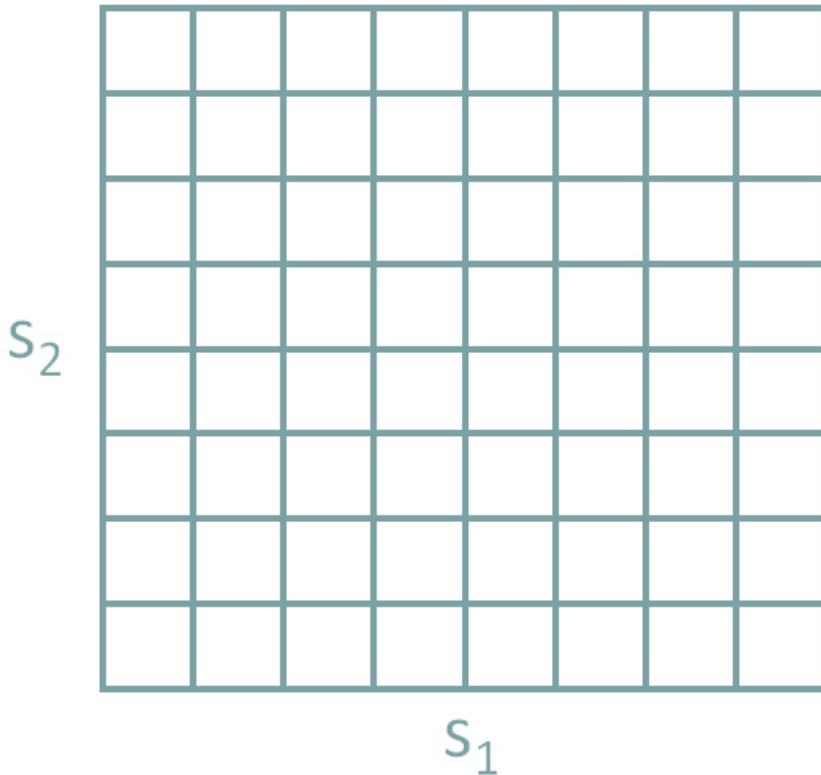
- want to discretize the state space
 - learn the best discretization during training
- splitting of state space
 - start with a single state
 - split a state when different *parts of that state* have different values



- state aggregation
 - start with many states
 - merge states with similar values



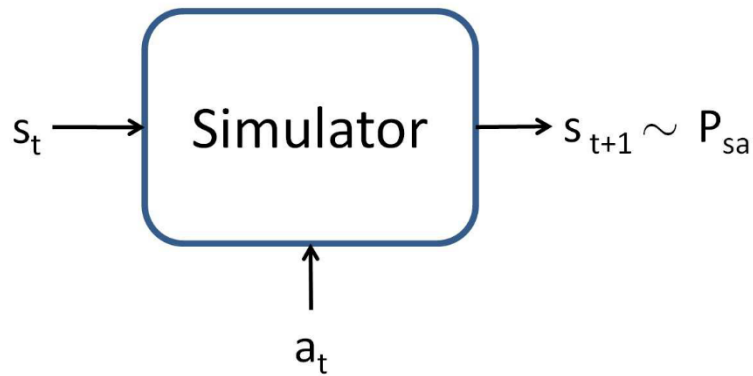
DISCRETIZATION



As a rule of thumb, discretization usually works extremely well for 1d and 2d problems. Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you're extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

being simple and quick to implement

VALUE FUNCTION APPROXIMATION



$$s_{t+1} = As_t + Ba_t$$


$$\arg \min_{A,B} \sum_{i=1}^m \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2$$

$$V(s) = \theta^T \phi(s)$$

$$V(s^{(i)}) \approx y^{(i)}$$

$$\theta := \arg \min_{\theta} \frac{1}{2} \sum_{i=1}^m \left(\theta^T \phi(s^{(i)}) - y^{(i)} \right)^2$$

DESIGNING REWARDS

- robot in a maze
 - episodic task, not discounted, +1 when out, 0 for each step
- chess
 - GOOD: +1 for winning, -1 losing
 - BAD: +0.25 for taking opponent's pieces
 - high reward even when lose
- rewards
 - rewards indicate what we want to accomplish
 - NOT how we want to accomplish it
- shaping
 - positive reward often very "far away"
 - rewards for achieving subgoals (domain knowledge)
 - also: adjust initial policy or initial value function

REINFORCEMENT LEARNING AND CONTROL

Continuous state

Designing rewards

Adjust initial policy or initial value function

Autonomous helicopter flight via reinforcement learning



**Autonomous helicopter flight
via reinforcement learning**

Andrew Y. Ng
Stanford University
Stanford, CA 94305

H. Jin Kim, Michael I. Jordan, and Shankar Sastry
University of California
Berkeley, CA 94720

AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING

Autonomous helicopter flight is widely regarded to be a **highly challenging control problem**.

Human experts can **reliably fly helicopters** through a wide range of maneuvers.

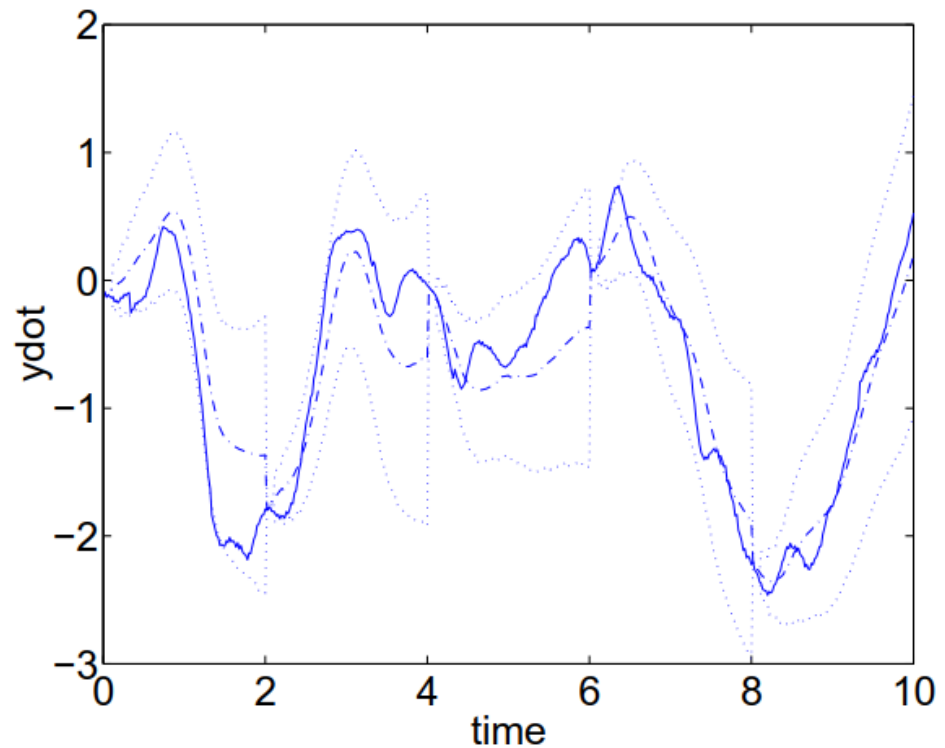
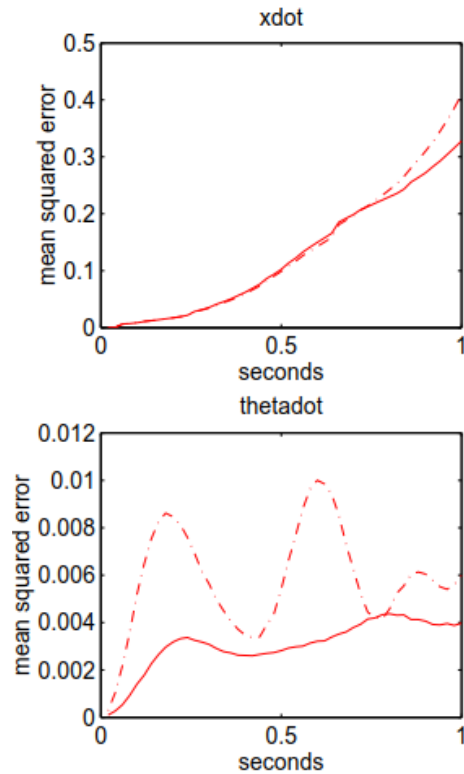
$$\begin{aligned}\dot{u} &= vr - wq + A_x u + g_x + w_u, \\ \dot{v} &= wp - ur + A_y v + g_y + D_0 + w_v, \\ \dot{w} &= uq - vp + A_z w + g_z + C_4 u_4 + D_4 + w_w, \\ \dot{p} &= qr(I_{yy} - I_{zz})/I_{xx} + B_x p + C_1 u_1 + D_1 + w_p, \\ \dot{q} &= pr(I_{zz} - I_{xx})/I_{yy} + B_y q + C_2 u_2 + D_2 + w_q, \\ \dot{r} &= pq(I_{xx} - I_{yy})/I_{zz} + B_z r + C_3 u_3 + D_3 + w_r.\end{aligned}$$

AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING

Kalman filter needed to estimate the helicopter's position and orientation, velocity and angular velocities.

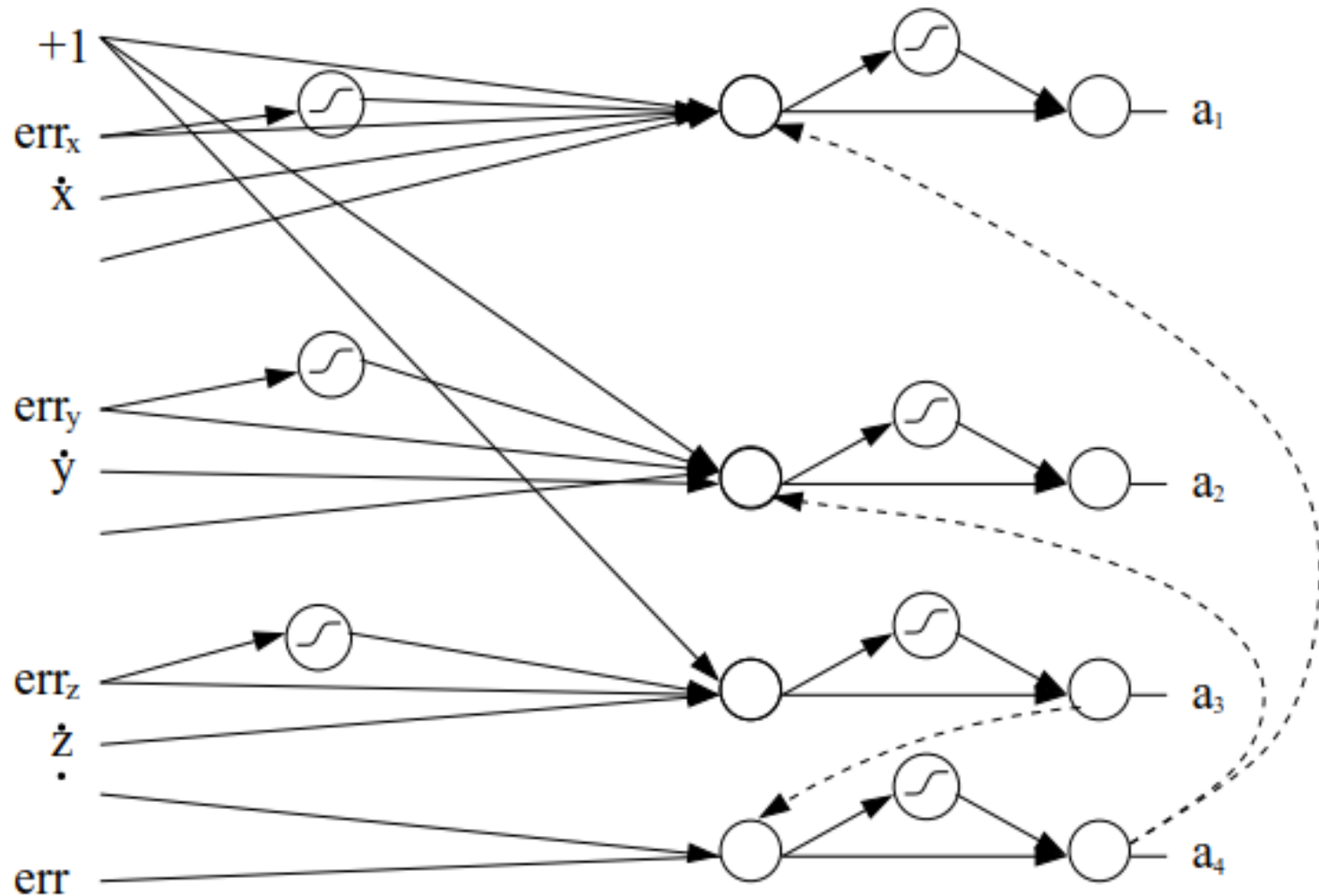
- 1) Collect data from a human pilot flying the desired maneuvers with the helicopter. Learn a model from the data.
- 2) Find a controller that works in simulation based on the current model.
- 3) Test the controller on the helicopter. If it works, we are done. Otherwise, use the data from the test flight to learn a new (improved) model and go back to Step 2

AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING



AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING

$$t_1 = w_1 + w_2 \text{err}_{x^b} + w_3 \tanh(w_4 \text{err}_{x^b}) + w_5 \dot{x}^b + w_6 \theta; a_1 = w_7 \tanh(w_8 t_1) + w_9 t_1.$$



AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING

Designing rewards

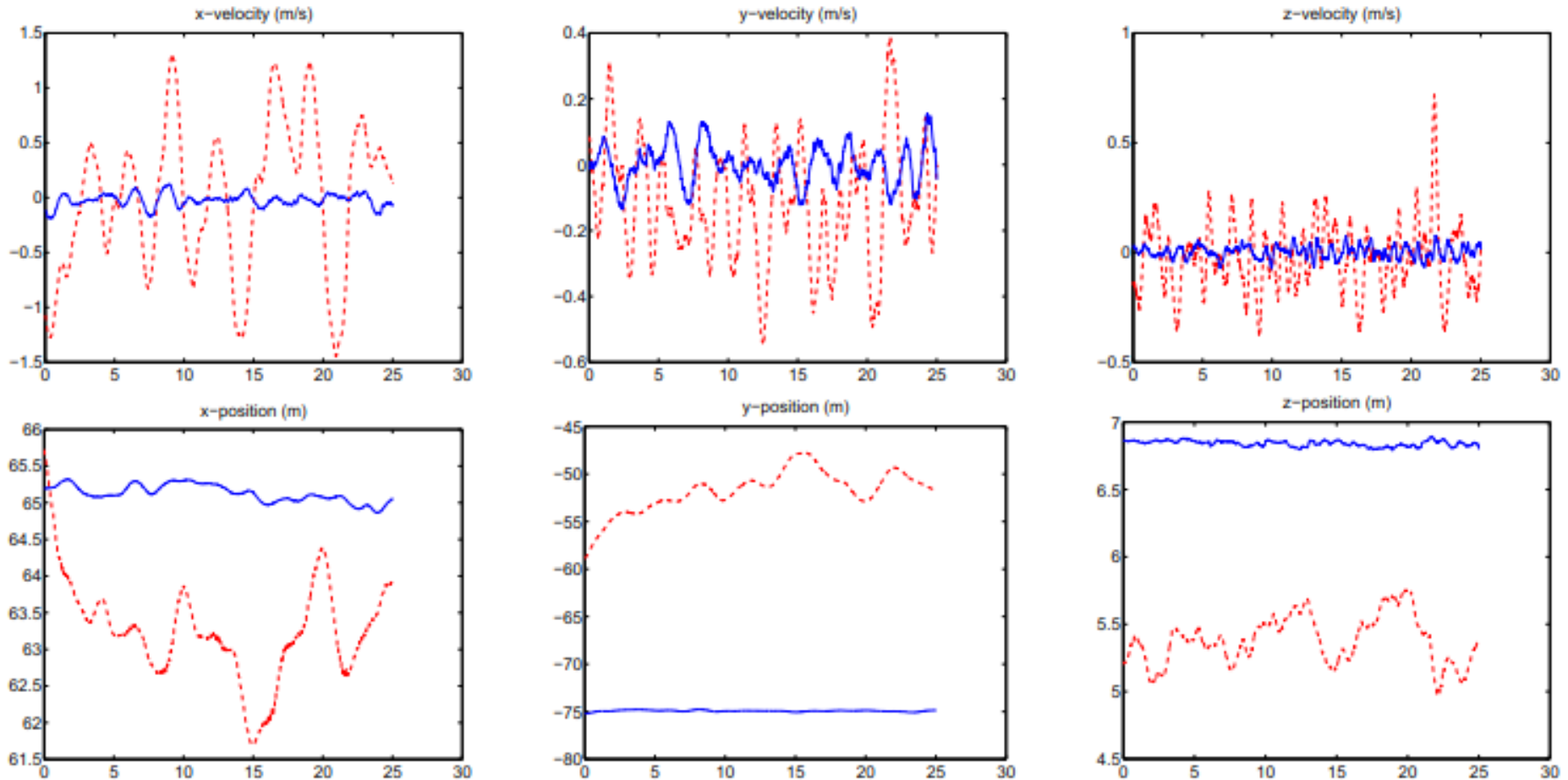
$$R(s) = -(\alpha_x(x-x^*)^2 + \alpha_y(y-y^*)^2 + \alpha_z(z-z^*)^2 + \alpha_{\dot{x}}\dot{x}^2 + \alpha_{\dot{y}}\dot{y}^2 + \alpha_{\dot{z}}\dot{z}^2 + \alpha_{\omega}(\omega-\omega^*)^2)$$

$$R(a) = -(\alpha_{a_1}a_1^2 + \alpha_{a_2}a_2^2 + \alpha_{a_3}a_3^2 + \alpha_{a_4}a_4^2)$$

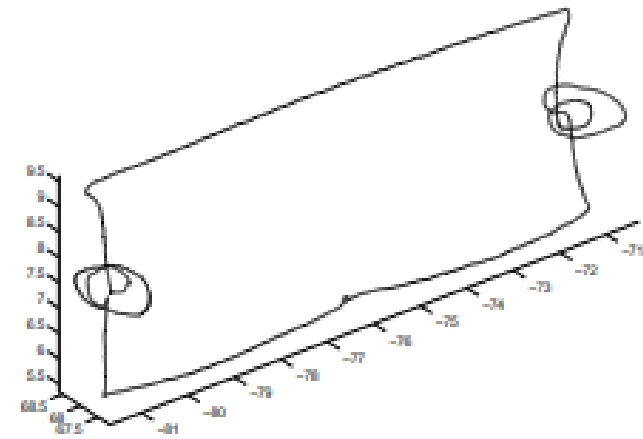
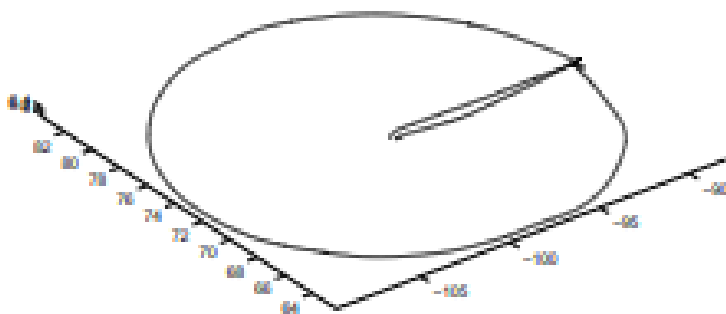
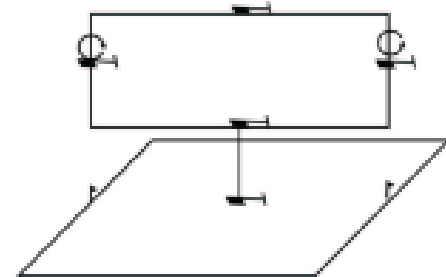
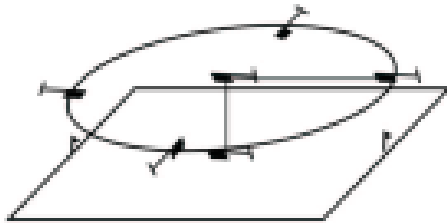
$$\hat{R}(s, a) = R(s) + \hat{R}(a)$$

$$U(\pi) = \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | \pi]$$

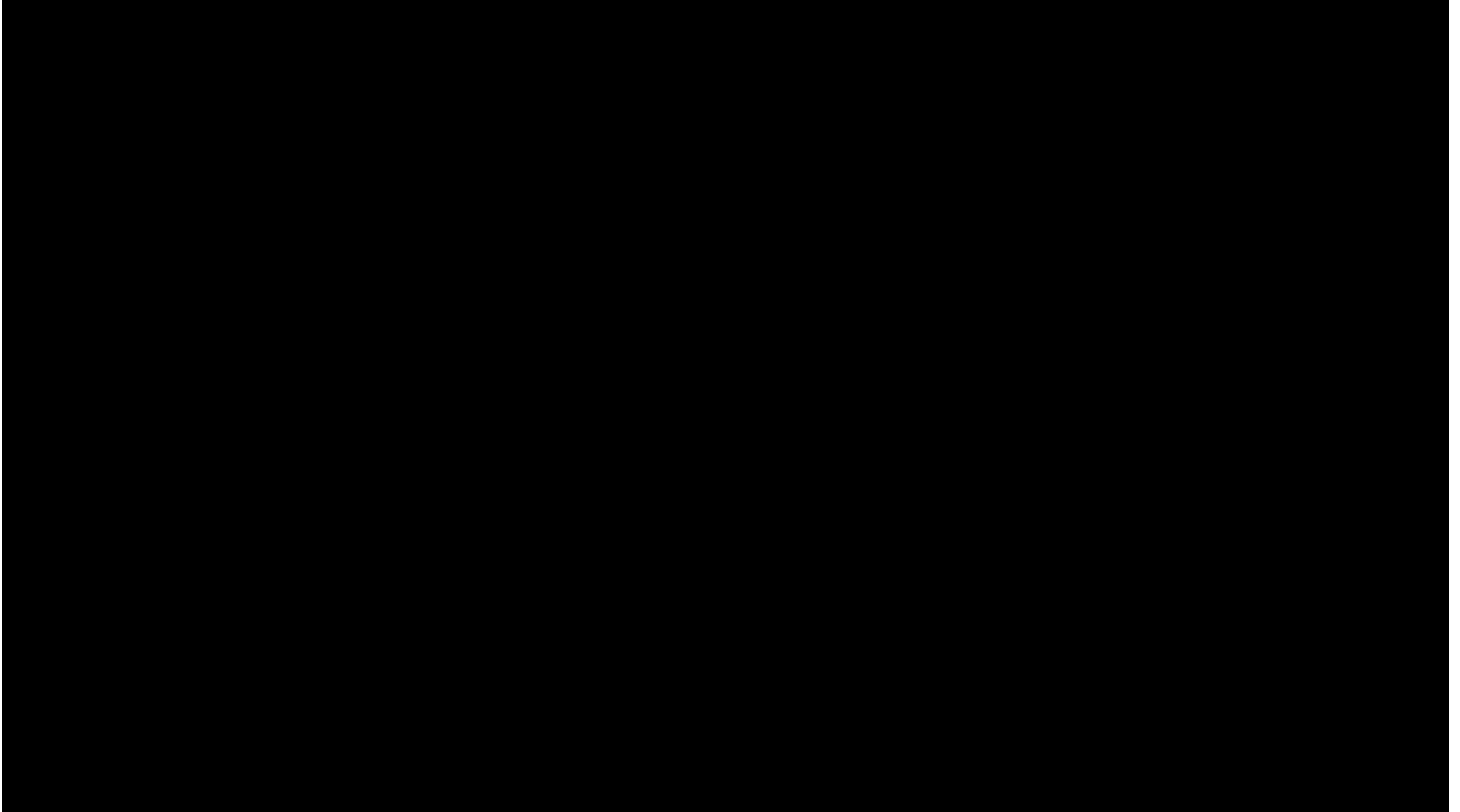
AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING



AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING



AUTONOMOUS HELICOPTER FLIGHT VIA REINFORCEMENT LEARNING



Reinforcement Learning

Application Perspective

THANKS

ANY QUESTIONS?