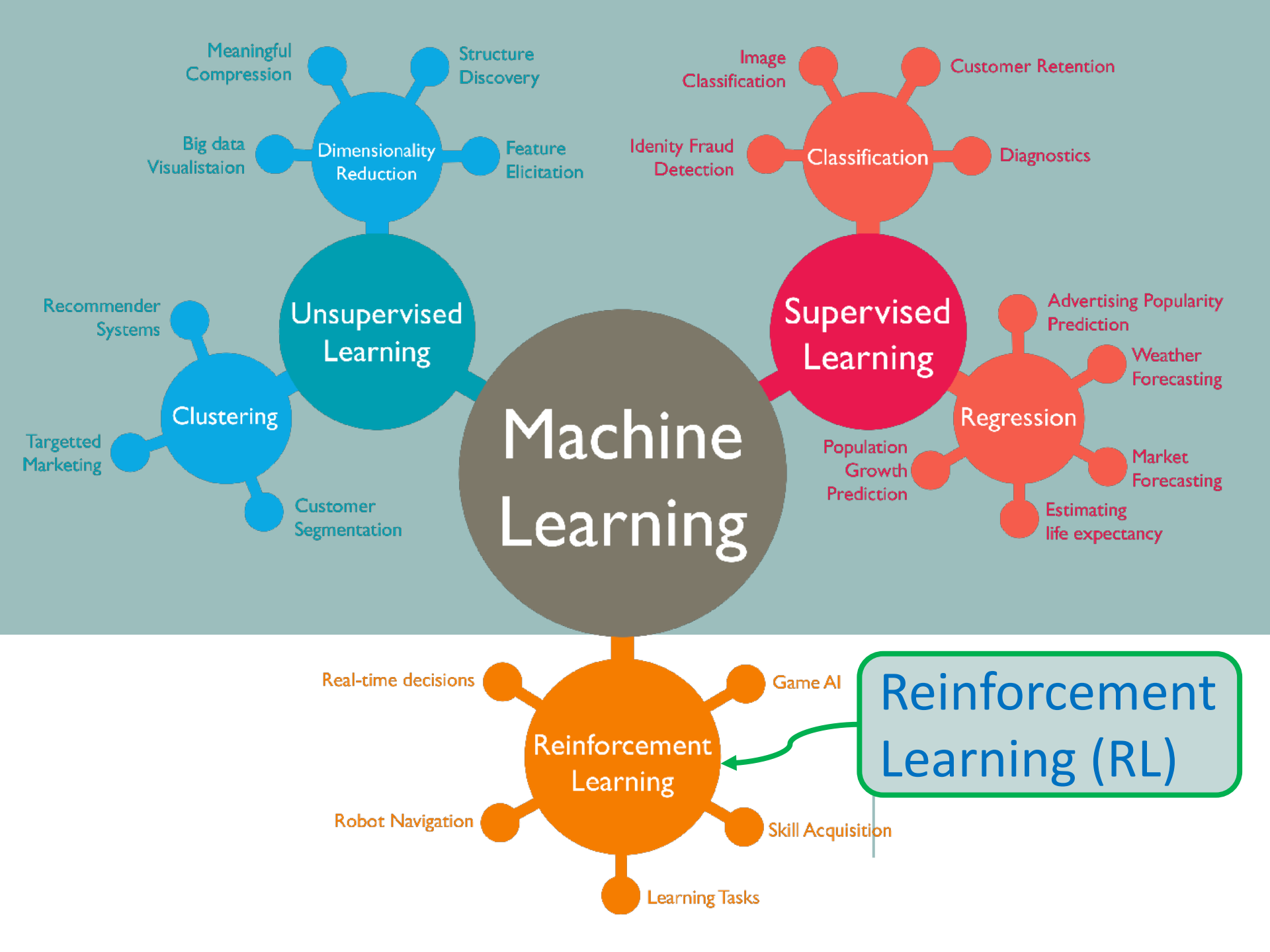

Meaningful Compression
Structure Discovery
Big data Visualistaion
Dimensionality Reduction
Feature Elicitation
Unsupervised Learning
Recommender Systems
Clustering
Targetted Marketing
Customer Segmentation
Machine Learning
Image Classification
Customer Retention
Idenity Fraud Detection
Classification
Diagnostics
Supervised Learning
Advertising Popularity Prediction
Weather Forecasting
Regression
Population Growth Prediction
Market Forecasting
Estimating life expectancy
Real-time decisions
Game AI
Reinforcement Learning
Robot Navigation
Skill Acquisition
Learning Tasks
Reinforcement Learning (RL)

ROADMAP

- What is RL about?
- Application Scenario
- The Task
- Markov Decision Process
- Q-Learning
- Deep Q-Learning
- Concluding Remarks

OVERVIEW

- Supervised Learning: Immediate feedback (labels/values provided for every input).
- Unsupervised Learning: No feedback (no labels/values to use).
- Reinforcement Learning (RL): Delayed scalar feedback (a number called reward).

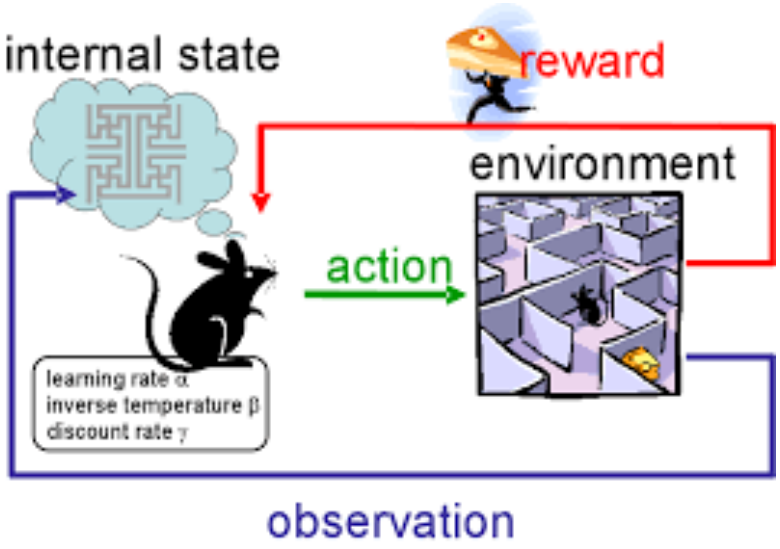
RL deals with agents that must sense & act upon their environment. It

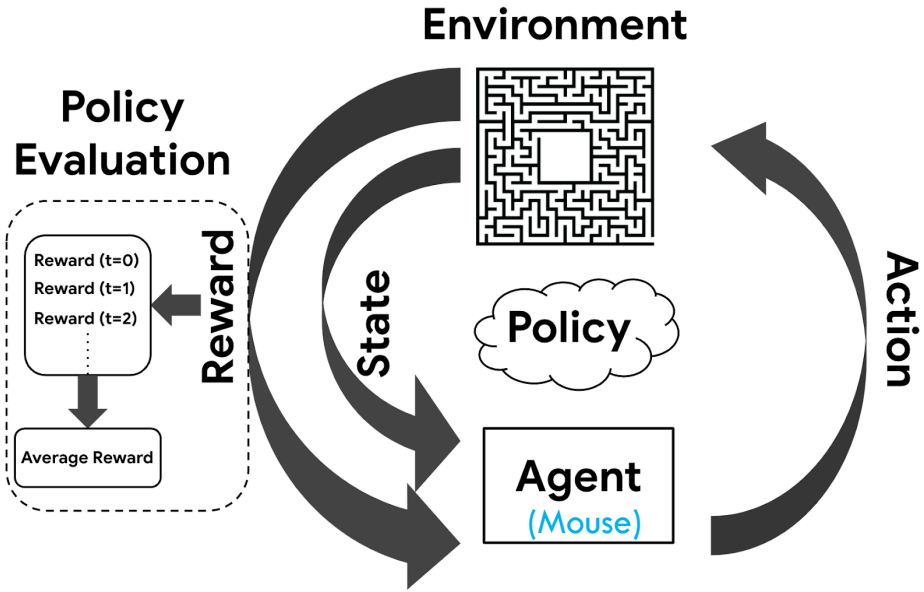
- combines classical AI and ML techniques.
- deals with very comprehensive problem settings.

Application Examples:

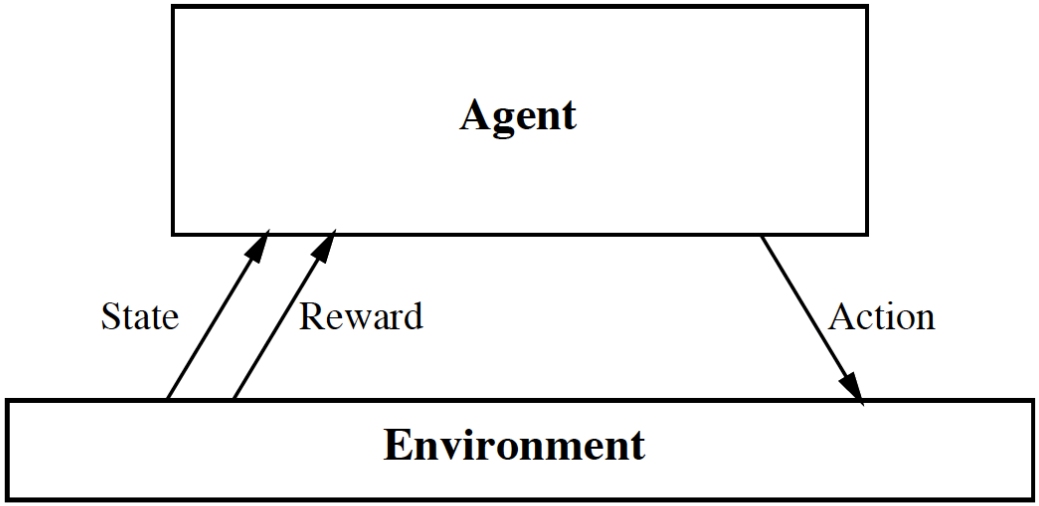
- AWS Deep Car Racer
- Cleaning robot and Robot-soccer
- Investment in shares (investment decision)
- Game Playing
- Critical control: Cart-pole Balancing, drone control, etc.
- and so on

THE APPLICATION SCENARIO





THE BIG PICTURE



Action:

State: $s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$

Reward:

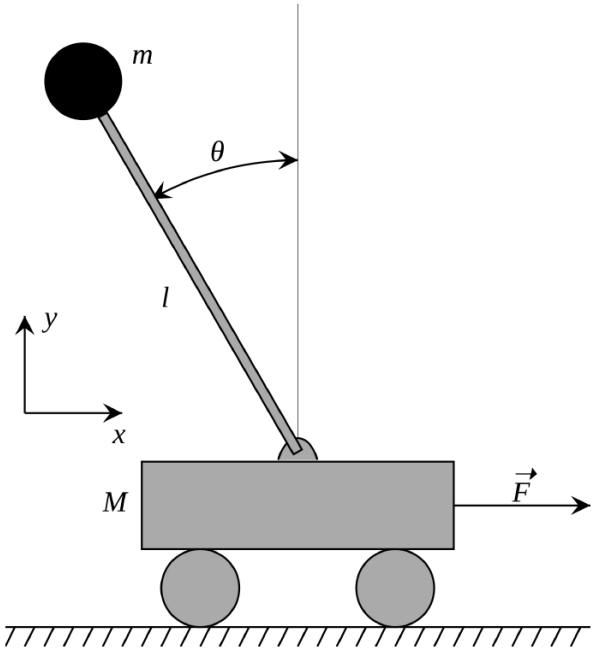
Your action influences the state of the world which determines its reward

APPLICATION EXAMPLES:

CART-POLE PROBLEM (SEE THIS VIDEO)

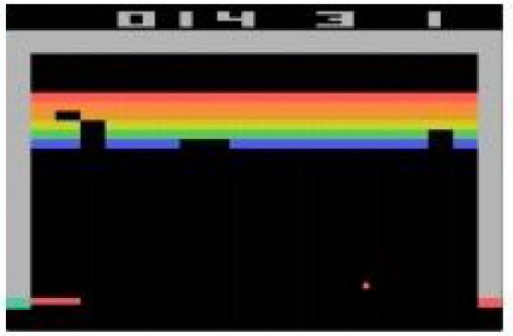
- Objective/Task: Balancing a pole on top of a movable cart

- State: Angle, angular speed, position, horizontal velocity
- Action: Horizontal force applied on the cart
- Reward: 1 at each time step if the pole is upright.



APPLICATION EXAMPLES:

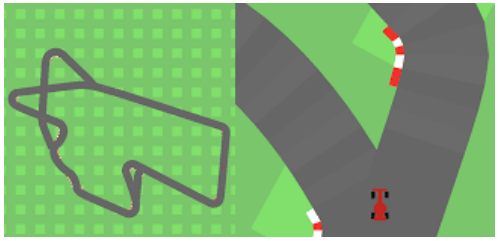
GAME PLAYING



RL Playing Breakout

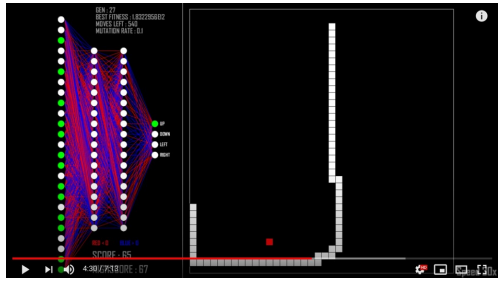
- Objective/Task: Complete the game with the highest score

- State: Raw pixel inputs of the game state
- Action: Joystick control, e.g. L, R, Up, Down
- Reward: Increase/decrease of score at each time step.



RL Car Racking
Deep RL Car Racing

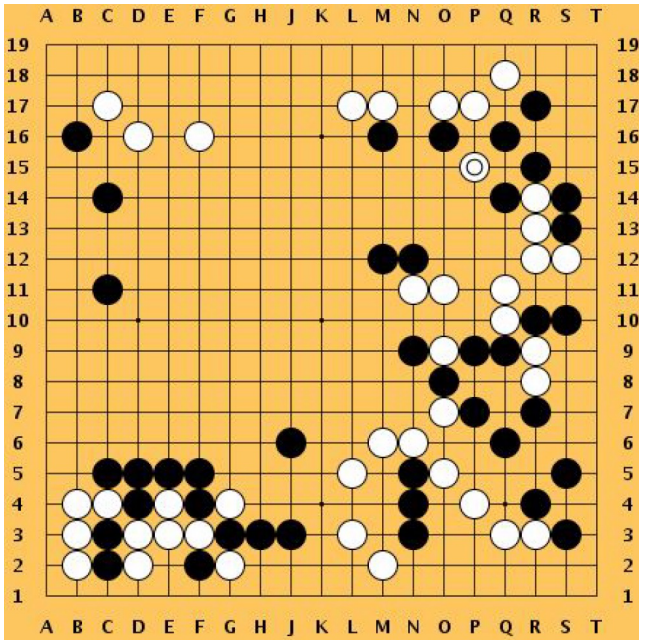
Angry bird,
Snake,
Pacman, etc.



APPLICATION EXAMPLES:
GAME GO

- Objective/Task: Win the game

- State: Stone positions or raw pixel inputs of the game state
- Action: Where to put the next stone piece down
- Reward: 1 if win at the end of the game and 0 otherwise.



SOME COMPLICATIONS

- The outcome of your actions may be uncertain.
- You may not be able to perfectly sense the state of the world.
- The reward may be stochastic.
- Reward is delayed (i.e. finding food in a maze, winning the game finally, etc.).
- You may have no clue (model) about how the world responds to your actions.
- You may have no clue (model) of how rewards are being paid off.
- The world may change while you try to learn it (dynamic behavior).
- How can you sense the whole “world”?

THE TASK

$\pi : S \rightarrow A$ Action:

State: $s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$

Reward:

- To learn an optimal *policy* that maps states of the world to actions of the agent.
  - If this patch of the room is dirty, clean it.
  - If the battery is empty, recharge it.

$$\pi : S \rightarrow A$$

- So, what the agent tries to optimize? Any error or loss function?
  - A value function in terms of total future discounted reward

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \leq \gamma < 1$$

Note: Immediate reward is worth more than future reward (exponentially decay).

MARKOV DECISION PROCESS (MDP)

- The mathematical formulation (probability theory) of the RL problem

  Markov property: Current state completely characterizes the state of the environment, and that the future state is independent of the past states given the current state.

- This means that if you know the current state of a system, you can predict the future behavior of that system without needing to know its complete history. So, we just need to focus on the current state and use it to make decisions about future actions.

Defined by: (S, A, R, P, γ)

S: set of possible states A: set of possible actions

R: distribution of reward given a state-action pair (*state s, action a*)

P: probability of transition, i.e., the distribution over the next state given the current (*s,a*) pair

γ : discount factor

MARKOV DECISION PROCESS

- At time t=0, environment samples the initial state s_0
- Then, for t=0 until end
  - Agent selects action a_t
  - Environment samples reward r_t
  - Environment samples next state s_{t+1}
  - Agent receives reward r_t and next state s_{t+1}

- A policy π is a function from S to A that specifies what action to take in each state
- Objective: Find optimal policy π^* that maximizes cumulative discounted reward:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

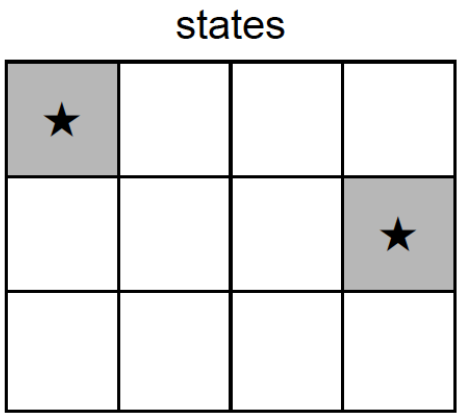
$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \le \gamma < 1$$

A SIMPLE MDP: THE MATRIX WORLD

actions = {

1. right
2. left
3. up
4. down

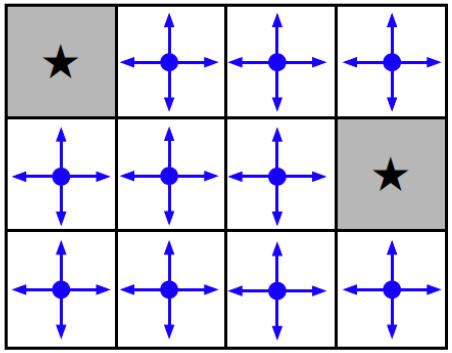
}



Set a negative "reward" for each transition (e.g. $r = -1$)

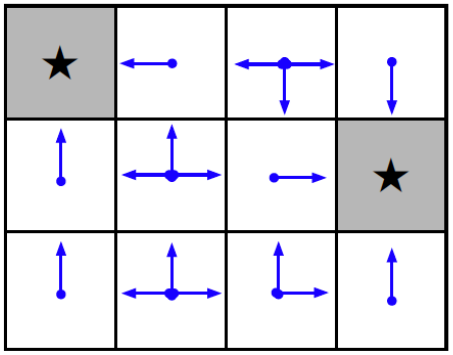
Objective: reach one of terminal states (greyed out) in least number of actions

A SIMPLE MDP: THE MATRIX WORLD



Random Policy

We want to get this ->



Optimal Policy

Optimal policy π^* that maximizes cumulative discounted reward

VALUE FUNCTION

- Let's say we have access to the optimal value function that computes the total future discounted reward $V^*(s)$
- What would be the optimal policy $\pi^*(s)$?
- Solution: Choose the action that maximizes:

$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

- Assuming that we know what the reward will be if we perform action "a" in state "s": $r(s,a)$
- And assuming that we know what the next state of the world will be if we perform action "a" in state "s":

$$s_{t+1} = \delta(s_t, a)$$

Q-FUNCTION (Q FOR QUALITY)

- One approach to Reinforcement Learning (RL) is to estimate V*(s).

  Bellman Equation: $V^*(s) \leftarrow \max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$

- However, this approach requires you to know $r(s,a)$ and $\delta(s,a)$, which is unrealistic in many real problems.
  - E.g., what is the reward if a robot is exploring Mars and decides to take a right turn?
- Fortunately, we can circumvent this problem by exploring and experiencing (i.e., learning) how the world reacts to our actions.
- We want a function that directly learns good state-action pairs, i.e. what action should I take in this state. We call this $Q(s,a)$, quality function of (s,a).

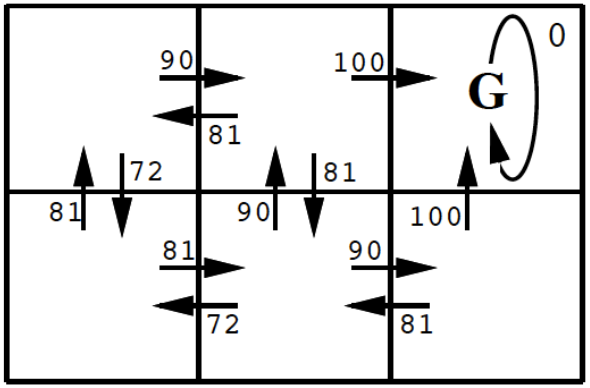
$$\pi^*(s) = \arg\max_a Q(s,a)$$

- Given $Q(s,a)$, it is now trivial to execute the optimal policy, *without knowing* $r(s,a)$ and $\delta(s,a)$. We have:

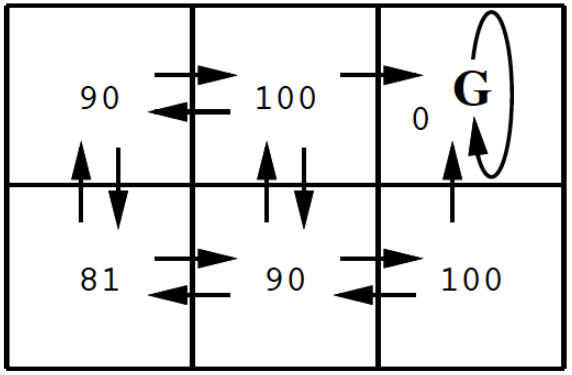
$$V^*(s) = \max_a Q(s,a)$$

EXAMPLE #2: GOAL FINDING USING Q(·,·)

Find your way to the goal.



Learn: $Q(s,a)$ values



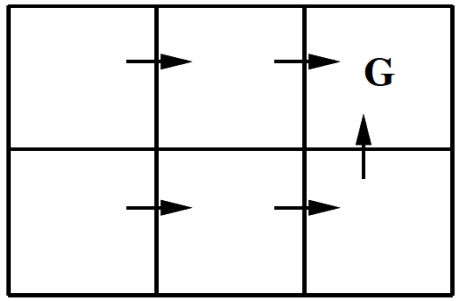
Compute: $V^*(s)$ values

So, how to learn Q(s,a)?

Check that

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

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



Find: One optimal policy

Q-LEARNING

- So, how to learn the quality function $Q(s,a)$?
  Recall that

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

  which still depends on $r(s,a)$ and $\delta(s,a)$.
- However, imagine the robot is exploring its environment, trying new actions as it goes. At every step it receives some reward "r", and it observes the environment change into a new state s' for action a.
- The question becomes "How can we use these observations, (s, a, s',r) to learn a model?"

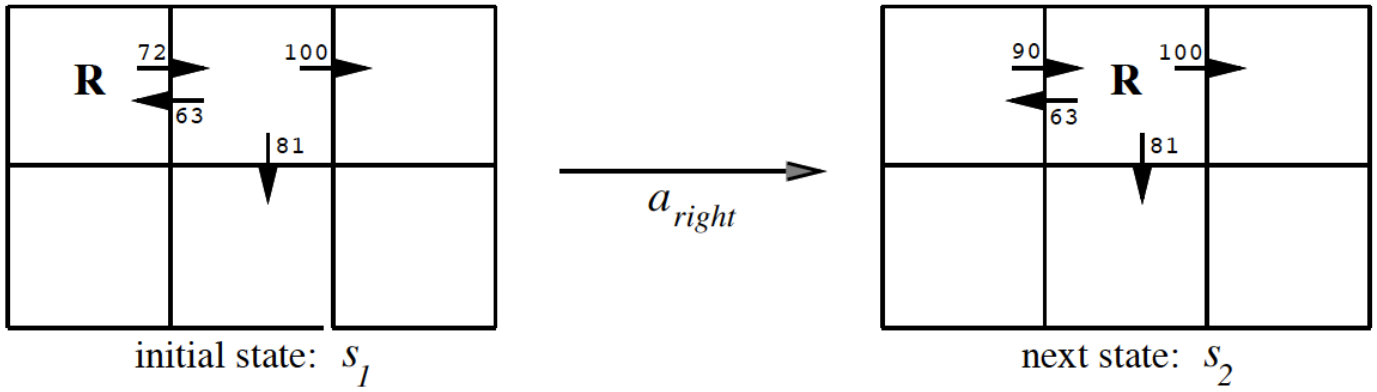
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \qquad s' = s_{t+1}$$

Q-LEARNING

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \qquad s' = s_{t+1}$$

- This equation continually estimates Q at state *s* consistent with an estimate of Q at state *s*', i.e., one step in the future, resulting a temporal difference (TD) learning.
- Note that *s*' is closer to goal, and hence more "reliable", but still an estimate itself.
- Updating estimates based on other estimates is called *bootstrapping*.
- We do an update after each state-action pair, i.e., we are learning online!
- We are learning useful things about explored state-action pairs. These are typically most useful because they are likely to be encountered again.
- Under suitable conditions, these updates can actually be proved to converge to the real answer.

AN EXAMPLE OF Q-LEARNING



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \max\{63, 81, 100\}$$

$$\leftarrow 90$$

Q-learning propagates Q-estimates 1-step backwards

EXPLORATION / EXPLOITATION

- It is very important that the agent does not simply follow the current policy when learning Q (off-policy learning). The reason is that you may get stuck in a suboptimal solution, i.e. there may be other solutions out there that you have never seen.

- Hence, it is good to try new things so now and then, e.g. via a control parameter *T*:
  - If *T* is large, lots of exploring, and
  - if *T* is small, follow current policy (for exploitation).
- One can decrease *T* over time so that for

$$P(a \mid s) \propto e^{\hat{Q}(s,a)/T}$$

Q-learning does not follow current policy initially (exploration) but follow it as time goes by (exploitation).

IMPROVEMENTS

- A major issue of Q-Learning is its scalability. As we need to compute Q(*s,a*) for every state-action pair, the search space can be exponentially huge. Imagine the state of current game pixels (i.e. the pixel combinations of the game scene), it is computationally infeasible to compute for the entire state space.

- One possible solution: Use a function approximator to estimate Q(*s,a*), e.g. a neural network!

- If a deep neural network is employed, **Deep Q-Learning** is resulted.

CASE STUDY

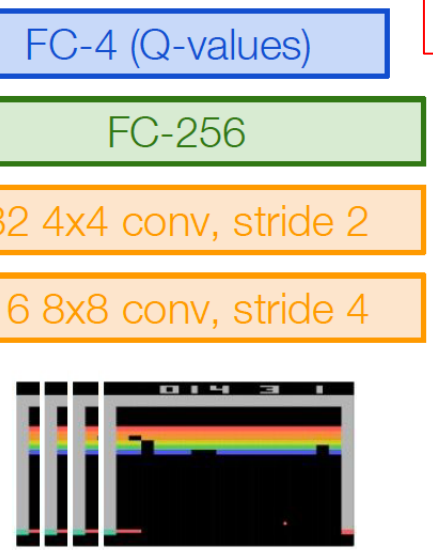
Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

Q-NETWORK ARCHITECTURE

[MNIH ET AL. NIPS WORKSHOP 2013; NATURE 2015]

$Q(s, a; \theta)$: neural network with weights θ



Last fc-layer has 4-d output (for 4 actions), i.e. $Q(s_t,a_1)$, $Q(s_t,a_2)$, $Q(s_t,a_3)$, $Q(s_t,a_4)$.

Ordinary CNN-like layers

Input state

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

TRAINING OF Q-NETWORK

- Learning from batches of consecutive samples is difficult:
  - Samples are correlated => inefficient learning!
  - Imbalanced samples (e.g. we always move right rather than left in breakout games) => leading to bad feedback loops

- Different strategies have been proposed to address the problems.

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Training the Q-network: Loss function (from before)

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

Still the loss function and gradient descent!

CONCLUDING REMARKS

- Reinforcement learning addresses a very broad and relevant question:

  How can we learn to survive in our environment?

- We have looked at Q-learning, which simply learns from experience.
  - No model of the world (i.e., no understanding of the world) is needed.

- We made simplifying assumptions: e.g. the future is independent of the past, given the present. This is the *Markov* assumption. The model is called a *Markov Decision Process (MDP).*

- We assumed deterministic dynamics, reward function, but the world is really stochastic.

- There are many extensions to speed up learning, like deep Q-learning.

- There have been many successful real world applications.

ACKNOWLEDGEMENT

Slides from following sources:

- Fei-Fei Li, Justin Johnson and Serena Yeung, Stanford U
- T. Mitchell, Machine Learning, Chapter 13.
- Max Welling, University of Amsterdam

Images from Google search of Internet