Deep Reinforcement Learning

2022-23 Second Semester, M.Tech (AIML)

Session #1:
Introduction to the Course

Instructors :

1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Dr. V Chandra Sekhar (chandrasekhar.v@wilp.bits-pilani.ac.in)

What is Reinforcement Learning ?

- reward based learning / feedback based learning
- not a type of NN nor it is an alternative to NN. Rather it is an approach for learning
- Autonomous driving, gaming

Why Reinforcement Learning ?

- a goal-oriented learning based on interaction with environment

Course Objectives

Course Objectives:

1. Understand
   a. the conceptual, mathematical foundations of deep reinforcement learning
   b. various classic & state of the art Deep Reinforcement Learning algorithms
2. Implement and Evaluate the deep reinforcement learning solutions to various problems like planning, control and decision making in various domains
3. Provide conceptual, mathematical and practical exposure on DRL
   a. to understand the recent developments in deep reinforcement learning and
   b. to enable modelling new problems as DRL problems.

Learning Outcomes

1. understand the fundamental concepts of reinforcement learning (RL), algorithms and apply them for solving problems including control, decision-making, and planning.
2. Implement DRL algorithms, handle challenges in training due to stability and convergence
3. evaluate the performance of DRL algorithms, including metrics such as sample efficiency, robustness and generalization.
4. understand the challenges and opportunities of applying DRL to real-world problems & model real life problems

Course Operation

- **Instructors**

  Prof. S.P.Vimal

  Dr. V Chandra Sekhar

- **Textbooks**
  1. Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, Second Ed. , MIT Press
  2. Foundations of Deep Reinforcement Learning: Theory and Practice in Python (Addison-Wesley Data & Analytics Series) 1st Edition by Laura Graesser and Wah Loon Keng

Course Operation

- **Evaluation**

  Two Quizzes for 5% each; **Best of two will be taken for 5%** (in final grading);

  Whatever be the points set for quizzes, the score will be scaled to 5%

  NO MAKEUP, for whatever be the reason. Ensure to attend at least one of the quizzes.

  Two Assignments - Tensorflow/ Pytorch / OpenAI Gym Toolkit → 25 %

  Assignment 1: Partially Numerical + Implementation of Classic Algorithms - 10%

  Assignment 2: Deep Learning based RL - 15%

  Mid-Term Exam - 30% [Only to be written in A4 pages, scanned and uploaded]

  Comprehensive Exam - 40% [Only to be written in A4 pages, scanned and uploaded]

- **Webinars/Tutorials**

  4 tutorials : 2 before mid-sem & 2 after mid-sem

- Teaching Assistants will be introduced to you in the next class

Course Operation

•Schedule of Schedule of Quizzes

Sunday, January 14, 2024	7:00:00 PM	Monday, January 15, 2024	7:00:00 PM
Sunday, March 10, 2024	7:00:00 PM	Monday, March 11, 2024	7:00:00 PM

•Schedule of Assignments

Assignment - #1: 02 Jan, 2024 7:00 PM 18 Jan, 2024 11:59 PM
Assignment - #2: 01, Mar 2024 7:00 PM 15, Mar 2024 11:59 PM

•Schedule of Webinars

21-Dec-23; 10-Jan-24; 22-Feb-24; 13-Mar-24

Course Operation

- How to reach us ? (for any question on lab aspects, availability of slides on portal, quiz availability , assignment operations)

1.Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2.Dr. V Chandra Sekhar (chandrasekhar.v@wilp.bits-pilani.ac.in)

- **Plagiarism [Important]**

All submissions for graded components must be the result of your original effort. It is strictly prohibited to copy and paste verbatim from any sources, whether online or from your peers. The use of unauthorized sources or materials, as well as collusion or unauthorized collaboration to gain an unfair advantage, is also strictly prohibited. Please note that we will not distinguish between the person sharing their resources and the one receiving them for plagiarism, and the consequences will apply to both parties equally.
In cases where suspicious circumstances arise, such as identical verbatim answers or a significant overlap of unreasonable similarities in a set of submissions, will be investigated, and severe punishments will be imposed on all those found guilty of plagiarism.

Reinforcement Learning

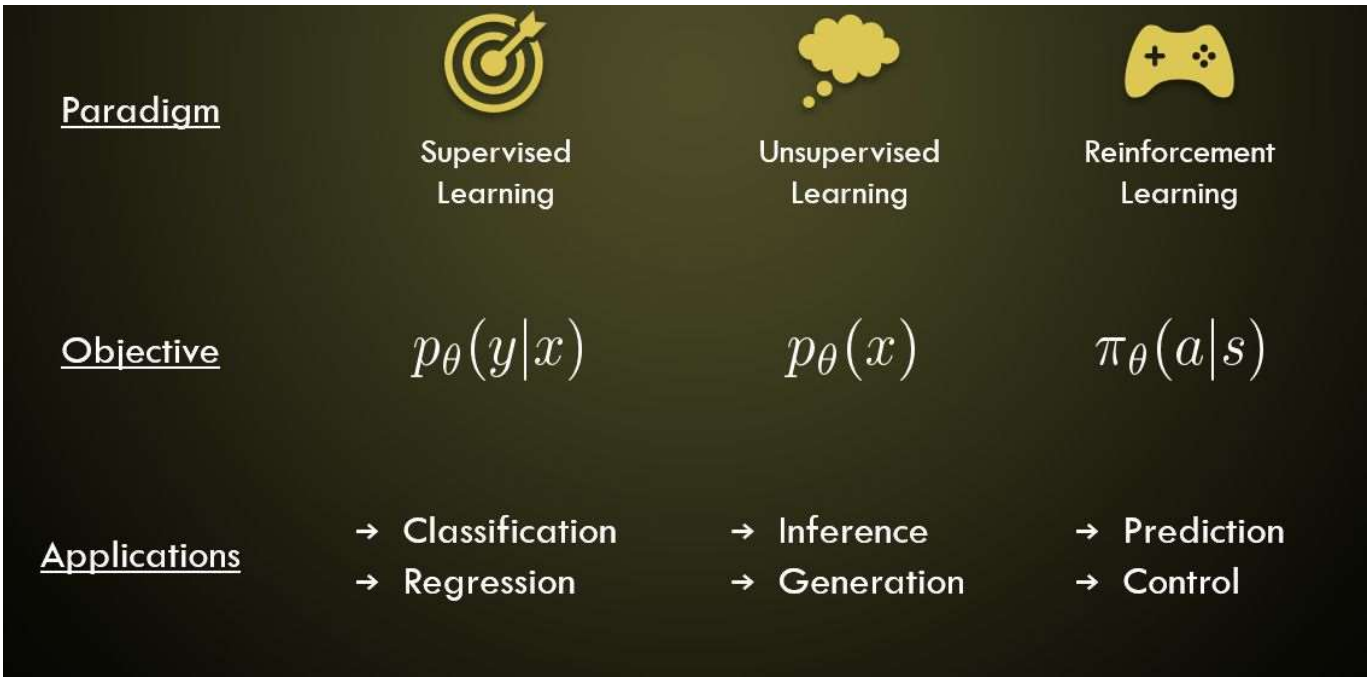
Reinforcement learning (RL) is based on rewarding desired behaviors or punishing undesired ones. Instead of one input producing one output, the algorithm produces a variety of outputs and is trained to select the right one based on certain variables – Gartner

When to use RL?

RL can be used in large environments in the following situations:

1.A model of the environment is known, but an analytic solution is not available;
2.Only a simulation model of the environment is given (the subject of simulation-based optimization)
3.The only way to collect information about the environment is to interact with it.

(Deep) Reinforcement Learning



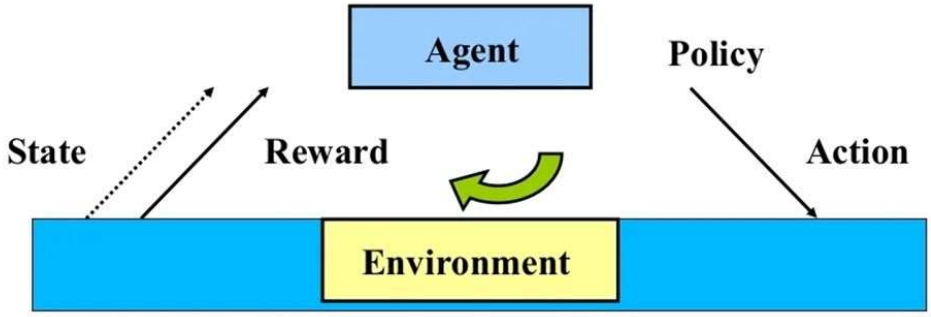
Types of Learning

Criteria	Supervised ML	Unsupervised ML	Reinforcement ML
Definition	Learns by using labelled data	Trained using unlabelled data without any guidance.	Works on interacting with the environment
Type of data	Labelled data	Unlabelled data	No – predefined data
Type of problems	Regression and classification	Association and Clustering	Exploitation or Exploration
Supervision	Extra supervision	No supervision	No supervision
Algorithms	Linear Regression, Logistic Regression, SVM, KNN etc.	K – Means, C – Means, Apriori	Q – Learning, SARSA
Aim	Calculate outcomes	Discover underlying patterns	Learn a series of action
Application	Risk Evaluation, Forecast Sales	Recommendation System, Anomaly Detection	Self Driving Cars, Gaming, Healthcare

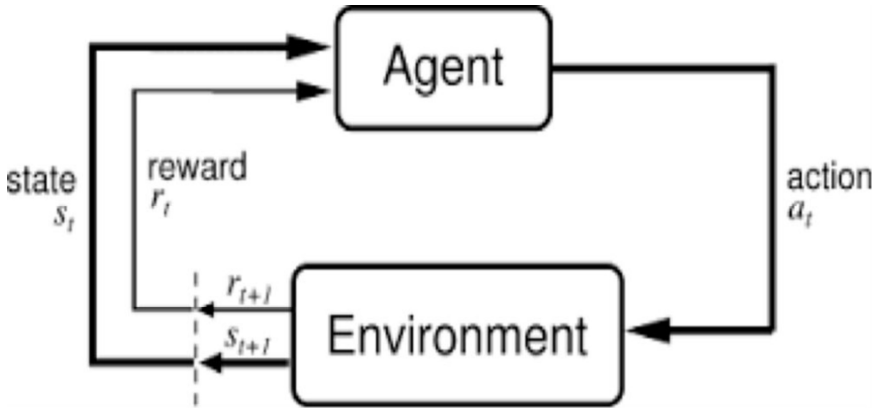
Characteristics of RL

- No supervision, only a real value or reward signal
- Decision making is sequential
- Time plays a major role in reinforcement problems
- Feedback isn't prompt but delayed

Elements of Reinforcement Learning



Elements of Reinforcement Learning



Beyond the agent and the environment, one can identify four main sub-elements of a reinforcement learning system: *a policy*, a *reward* , a *value* function, and, optionally, a *model* of the environment.

Elements of Reinforcement Learning

•**Agent**

- an **entity** that tries to learn the best way to perform a specific task.
- In our example, the child is the agent who learns to ride a bicycle.

•**Action (A)** -

- **what the agent does** at each time step.
- In the example of a child learning to walk, the action would be “walking”.
- A is the set of all possible moves.
- In video games, the list might include running right or left, jumping high or low, crouching or standing still.

Elements of Reinforcement Learning

•**State (S)**

- **current situation** of the agent.
- After doing performing an action, the agent can move to different states.
- In the example of a child learning to walk, the child can take the action of taking a step and move to the next state (position).

•**Rewards (R)**

- feedback that is given to the agent based on the action of the agent.
- If the action of the agent is good and can lead to winning or a positive side then a positive reward is given and vice versa.

Elements of Reinforcement Learning

- **Environment**
  - outside world of an agent or physical world in which the agent operates.
- **Discount factor**
  - The **discount factor** is multiplied by future rewards as discovered by the agent in order to dampen these rewards' effect on the agent's choice of action. Why? It is designed to make future rewards worth less than immediate rewards.

Often expressed with the lower-case Greek letter gamma: γ. If γ is .8, and there's a reward of 10 points after 3 time steps, the present value of that reward is $0.8^3 \times 10$.

Elements of Reinforcement Learning

Formal Definition - ***Reinforcement learning*** *(**RL**) is an area of machine learning concerned with how intelligent **agents** ought to take **actions** in an **environment** in order to maximize the notion of cumulative **reward**.*

End of Session #1

Elements of Reinforcement Learning

•***Goal of RL*** - maximize the total amount of rewards or cumulative rewards that are received by taking actions in given states.

•Notations –

a set of states as ***S***,

a set of actions as ***A***,

a set of rewards as ***R***.

At each time step ***t*** = 0, 1, 2, …, some representation of the environment's state $S_t \in S$ is received by the agent. According to this state, the agent selects an action $A_t \in A$ which gives us the state-action pair (S_t, A_t). In the next time step ***t+1***, the transition of the environment happens and the new state $S_{t+1} \in S$ is achieved. At this time step ***t+1***, a reward $R_{t+1} \in R$ is received by the agent for the action A_t taken from state S_t.

Elements of Reinforcement Learning

•Maximize cumulative rewards, Expected Return G_t

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$
$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

•Discount factor γ is introduced here which forces the agent to focus on immediate rewards instead of future rewards. The value of γ remains between 0 and 1.

Elements of Reinforcement Learning

•Policy (ΤΤ)

- Policy in RL decides which action will the agent take in the current state.
- It tells the *probability that an agent will select a specific action from a specific state*.
- Policy is a function that maps a given state to probabilities of selecting each possible action from the given state.

•If at time ***t,*** an agent follows policy ***π***, then ***π(a|s)*** becomes the probability that the action at time step ***t*** is $a_{t=a}$ if the state at time step ***t*** is $S_{t=s}$ .The meaning of this is, the probability that an agent will take an action ***a*** in state ***s*** is ***π(a|s)*** at time ***t*** with policy **ΤΤ**.

Elements of Reinforcement Learning

•**Value Functions**

- a simple measure of how good it is for an agent to be in a given state, or how good it is for the agent to perform a given action in a given state.

•Two types

- state- value function
- action-value function

Elements of Reinforcement Learning

•**State-value function**

- The *state-value function* for policy **π** denoted as $\mathbf{v}_{\boldsymbol{\pi}}$ determines the *goodness of any given state for an agent who is following policy π*.
- This function gives us the value which is the expected return starting from state ***s*** at time step ***t*** and following policy π afterward.

$$v_\pi(s) = E_\pi[G_t \mid S_t = s]$$
$$= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right].$$

Elements of Reinforcement Learning

•**Action value function**

- determines the goodness of the action taken by the agent from a given state for policy π.
- This function gives the value which is the expected return starting from state ***s*** at time step ***t***, with action ***a***, and following policy π afterward.
- The output of this function is also called as *Q-value* where q stands for Quality. Note that in the *state-value function,* we did not consider the action taken by the agent.

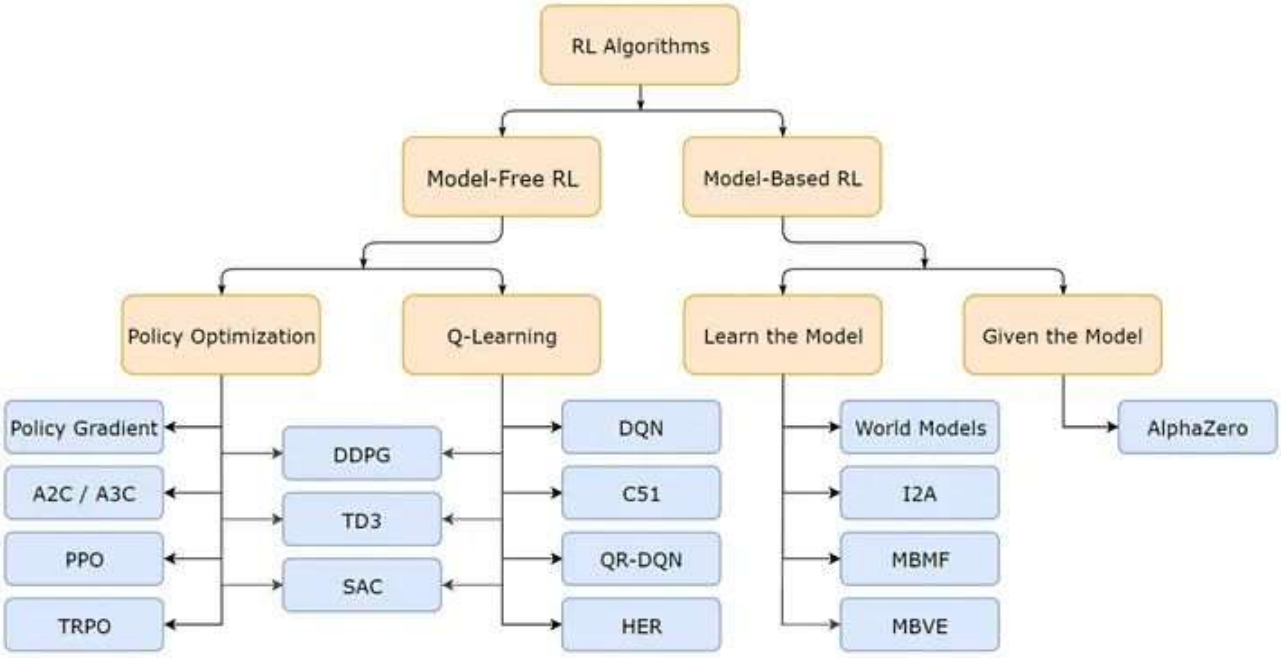
$$q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a]$$
$$= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right].$$

Elements of Reinforcement Learning

•**Model of the environment**

- mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave.
- For example, given a state and action, the model might predict the resultant next state and next reward.
- Models are used for planning

(Deep) Reinforcement Learning



From OpenAI

Advantages of Reinforcement Learning

- solve very complex problems that cannot be solved by conventional techniques
- achieve long-term results
- model can correct the errors that occurred during the training process.
- In the absence of a training dataset, it is bound to learn from its experience
- can be useful when the only way to collect information about the environment is to interact with it
- Reinforcement learning algorithms maintain a ***balance between exploration and exploitation***. Exploration is the process of trying different things to see if they are better than what has been tried before. Exploitation is the process of trying the things that have worked best in the past. Other learning algorithms do not perform this balance

An example scenario - Tic-Tac-Toe

Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally

Assumptions

- playing against an imperfect player, one whose play is sometimes incorrect and allows you to win

Aim

How might we construct a player that will find the imperfections in its opponent's play and learn to maximize its chances of winning?

Reinforcement Learning for Tic-Tac-Toe

•We set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state.

•We treat this estimate as the state's value, and the whole table is the learned value function.

•State A has higher value than state B, or is considered better than state B, if the current estimate of the probability of our winning from A is higher than it is from B.

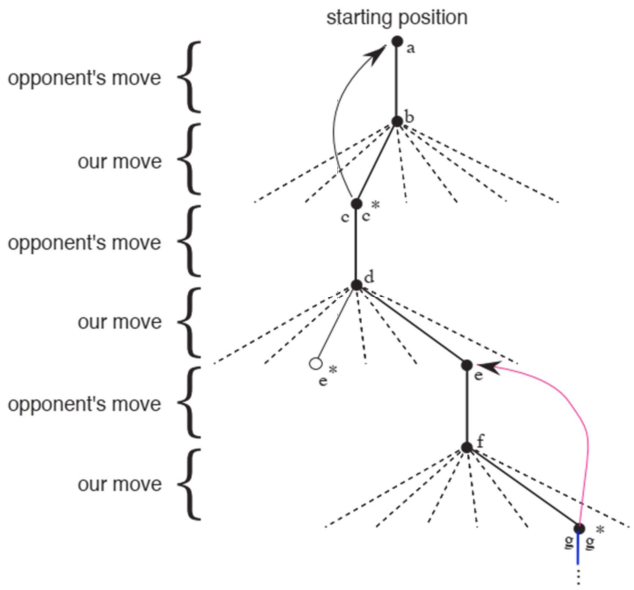
Reinforcement Learning for Tic-Tac-Toe

- Assuming we always play X's, three X = probability is 1, three O = probability =0 . Initial = 0.5
- Play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves and look up their current values in the table.
- Most of the time we move greedily, selecting the move that leads to the state with greatest value, i.e, with the highest estimated probability of winning.
- Occasionally, however, we select randomly from among the other moves instead. These are called **exploratory moves** because they cause us to experience states that we might otherwise never see.

Reinforcement Learning for Tic-Tac-Toe

•The solid lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make.

•Our second move was an exploratory move, meaning that it was taken even though another sibling move, the one leading to e*, was ranked higher.



Reinforcement Learning for Tic-Tac-Toe

- Once a game is started, our agent computes all possible actions it can take in the current state and the new states which would result from each action.
- The values of these states are collected from a ***state_value vector***, which contains values for all possible states in the game.
- The agent can then choose the action which leads to the state with the highest value(exploitation), or chooses a random action(exploration), depending on the value of epsilon.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

Deep Reinforcement Learning
2022-23 Second Semester, M.Tech (AIML)

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Session #2-3:
Multi-armed Bandits

Instructors :
1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Dr. V Chandra Sekhar (chandrasekhar.v@wilp.bits-pilani.ac.in)

Agenda for the class

- Recap
- k-armed Bandit Problem & its significance
- Action-Value Methods
  - Sample Average Method & Incremental Implementation
- Non-stationary Problem
- Initial Values & Action Selection
- Gradient Bandit Algorithms [Class #3]
- Associative Search [Class #3]

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Tic-Tac-Toc

X	O	O
O	X	X
		X

Tic-Tac-Toc

States	Initial Values
$\begin{bmatrix} X & & \\ & & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & X \end{bmatrix}$	1.0
$\begin{bmatrix} X & & O \\ X & & O \\ & X & O \end{bmatrix}$	0
...	...

Learning Task: Play as many times against the opponent and learn the values

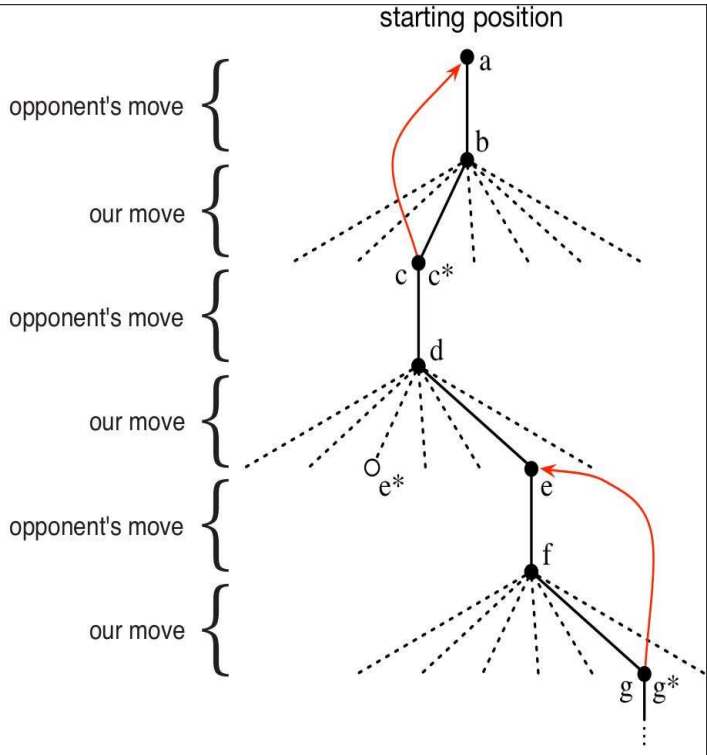
X	O	O
O	X	X
		X

Set up a table of states initial values

Tic-Tac-Toc (prev. class)

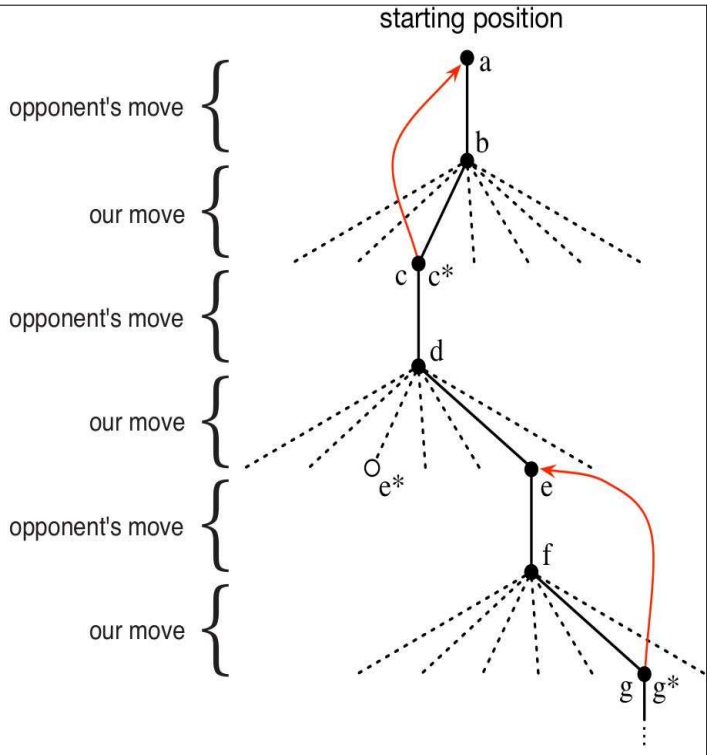
States	Initial Values
$\begin{bmatrix} X & & \\ & & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & X \end{bmatrix}$	1.0
$\begin{bmatrix} X & & O \\ X & & O \\ & X & O \end{bmatrix}$	0
...	...

S_t - state before greedy move
S_{t+1} - state after greedy move



$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

Tic-Tac-Toc (prev. class)



Temporal Difference Learning Rule

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

α- Step Size Parameter

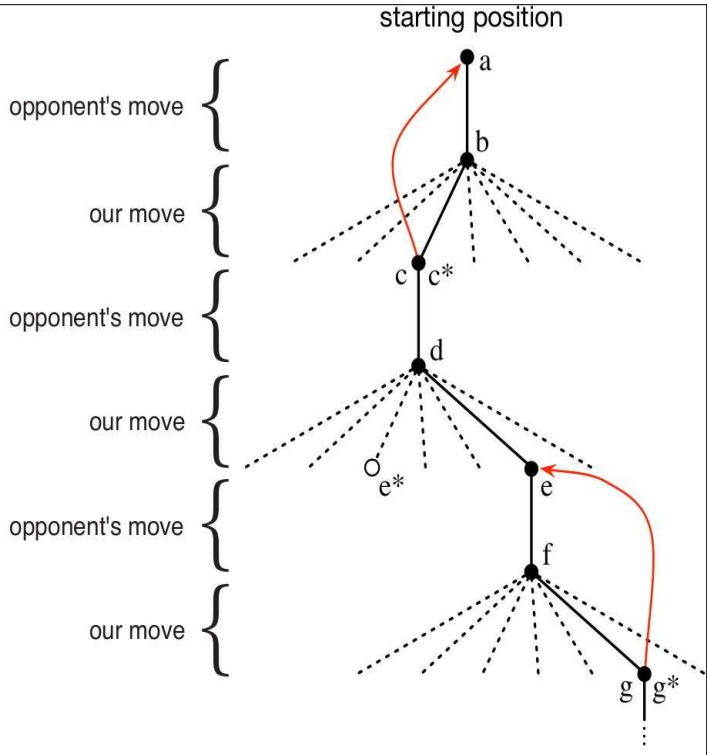
Tic-Tac-Toc (prev. class)

Questions:

(1) What happens if α is gradually made to 0 over many games with the opponent?

(2) What happens if α is gradually reduced over many games, but never made 0?

(3) What happens if α is kept constant throughout its life time?



Temporal Difference Learning Rule

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

α- Step Size Parameter

Tic-Tac-Toc (prev. class)

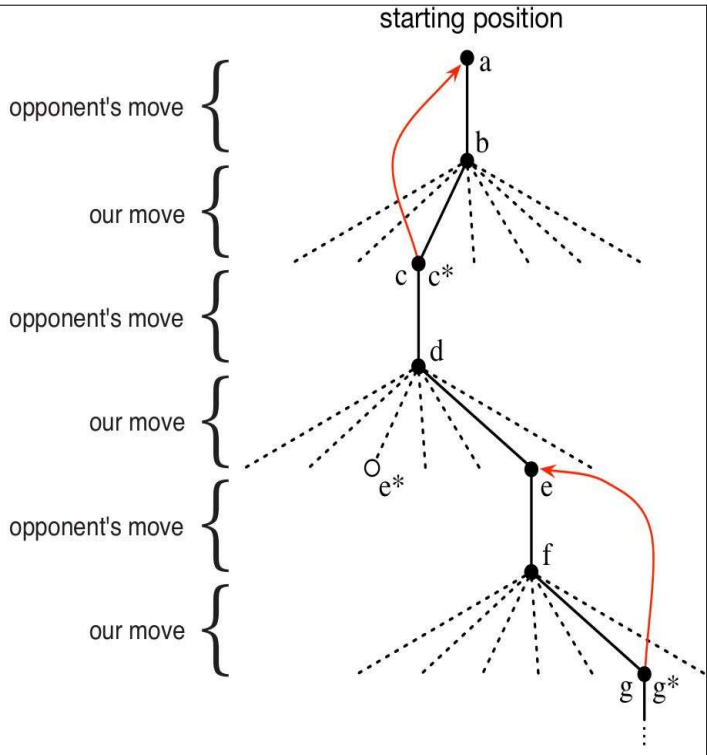
Key Takeaways:

(1) Learning while interacting with the environment (opponent).

(2) We have a clear goal

(3) Our policy is to make moves that maximizes our chances of reaching goal

- Use the values of states most of the time (exploration) and explore rest of the time.



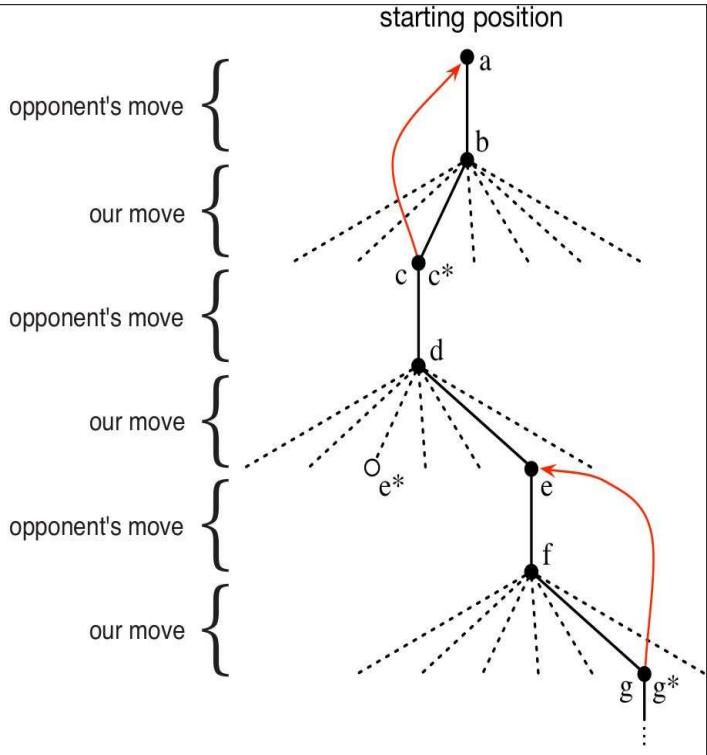
Temporal Difference Learning Rule

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

α- Step Size Parameter

Tic-Tac-Toc (prev. class)

Reading Assigned:
Identify how this reinforcement learning solution is different from solutions using minimax algorithm and genetic algorithms.
Post your answers in the discussion forum;



Temporal Difference Learning Rule

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

α- Step Size Parameter

K-armed Bandit Problem

K-armed Bandit Problem

Problem

- You are faced repeatedly with a choice among k different options, or actions
- After each choice of actions you receive a numerical reward

  Reward is chosen from a stationary probability distribution that depends on the selected action
- **Objective** : to *maximize the expected total reward over some time period*

K-armed Bandit Problem

Problem

- You are faced repeatedly with a choice among k different options, or actions
- After each choice of actions you receive a numerical reward

  Reward is chosen from a stationary probability distribution that depends on the selected action
- **Objective** : to *maximize the expected total reward over some time period*

K-armed Bandit Problem

Problem

- You are faced repeatedly with a choice among k different options, or actions
- After each choice of actions you receive a numerical reward

  Reward is chosen from a stationary probability distribution that depends on the selected action
- **Objective** : to *maximize the expected total reward over some time period*

Strategy:
- Identify the best lever(s)
- Keep pulling the identified ones

Questions:
- How do we define the *best one*s?
- What are the best levers?

K-armed Bandit Problem

Problem

- You are faced repeatedly with a choice among k different options, or actions
- After each choice of actions you receive a numerical reward

  Reward is chosen from a stationary probability distribution that depends on the selected action
- **Objective** : to *maximize the expected total reward over some time period*

Strategy:
- Identify the best lever(s)
- Keep pulling the identified ones

Questions:
- How do we define the *best one*s?
- What are the best levers?

K-armed Bandit Problem

Problem

- You are faced repeatedly with a choice among k different options, or actions
- After each choice of actions you receive a numerical reward

  Reward is chosen from a stationary probability distribution that depends on the selected action
- **Objective** : to *maximize the expected total reward over some time period*

$\mathbb{E}[a] = -\$0.5$

$\mathbb{E}[b] = -\$0.2$

$\mathbb{E}[c] = \$0.1$

$\mathbb{E}[d] = \$0.11$

- ***Expected Mean Reward*** for each action selected
  → call it ***Value*** of the action

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$$

K-armed Bandit Problem

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$$

- A_t - action selected on time step t
- $Q_t(a)$ - estimated value of action a at time step t
- $q_*(a)$ - value of an arbitrary action a

Note: If you knew the value of each action, then it would be trivial to solve the k -armed bandit problem: you would always select the action with highest value :-)

K-armed Bandit Problem

$\mathbb{E}[a] = -\$0.5$

$\mathbb{E}[b] = -\$0.2$

$\mathbb{E}[c] = \$0.1$

$\mathbb{E}[d] = \$0.11$

K-armed Bandit Problem

-1, -1, 5

$\hat{\mathbb{E}}[a] = 1$

-0.2, -0.2

$\hat{\mathbb{E}}[b] = -0.2$

-0.5, -0.5, -0.5

$\hat{\mathbb{E}}[c] = -0.5$

-2, -2

$\hat{\mathbb{E}}[d] = -2$

Keep pulling the levers; update the estimate of action values;

K-armed Bandit Problem

-1, -1, 5

$\widehat{\mathbb{E}}[a]=1$

-0.2, -0.2

$\widehat{\mathbb{E}}[b]=-0.2$

-0.5, -0.5, -0.5

$\widehat{\mathbb{E}}[c]=-0.5$

-2, -2

$\widehat{\mathbb{E}}[d]=-2$

K-armed Bandit Problem

1. How to maintain the estimate of expected rewards for each action?

   Average the rewards actually received !!!

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t}$$

$$= \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

1. How to use the estimate in selecting the right action?

   Greedy Action Selection $\quad A_t \doteq \arg\max_a Q_t(a)$

K-armed Bandit Problem

2. How to use the estimate in selecting the right action?

   Greedy Action Selection

$$A_t \doteq \arg\max_a Q_t(a)$$

Actions which are inferior by the value estimate upto time t, could be indeed better than the greedy action at t !!!

3. Exploration vs. Exploitation?

   ε-Greedy Action Selection / near-greedy action selection

   Behave greedily most of the time; Once in a while, with small probability **ε** select randomly from among all the actions with equal probability, independently of the action-value estimates.

K-armed Bandit Problem

{-1, -1, 5}
$N_{11}(a)=3$
$q_*(a)=-\$0.5$
$Q_{11}(a)=1$

{-0.2, -0.2}
$N_{11}(b)=2$
$q_*(b)=-\$0.2$
$Q_{11}(b)=-0.2$

{-0.5, -0.5, -0.5}
$N_{11}(c)=3$
$q_*(c)=\$0.1$
$Q_{11}(c)=-0.5$

{-2, -2}
$N_{11}(d)=2$
$q_*(d)=\$1$
$Q_{11}(d)=-2$

K-armed Bandit Problem

Greedy Action

{-1, -1, 5}

$N_{11}(a)=3$

$q_*(a)=-\$0.5$

$Q_{11}(a)=1$

{-0.2, -0.2}

$N_{11}(b)=2$

$q_*(b)=-\$0.2$

$Q_{11}(b)=-0.2$

{-0.5, -0.5, -0.5}

$N_{11}(c)=3$

$q_*(c)=\$0.1$

$Q_{11}(c)=-0.5$

{-2, -2}

$N_{11}(d)=2$

q*(d)=$1

$Q_{11}(d)=-2$

K-armed Bandit Problem

Action to Explore

{-1, -1, 5}

$N_{11}(a)=3$

$q_*(a)=-\$0.5$

$Q_{11}(a)=1$

{-0.2, -0.2}

$N_{11}(b)=2$

$q_*(b)=-\$0.2$

$Q_{11}(b)=-0.2$

{-0.5, -0.5, -0.5}

$N_{11}(c)=3$

$q_*(c)=\$0.1$

$Q_{11}(c)=-0.5$

{-2, -2}

$N_{11}(d)=2$

q*(d)=$1

$Q_{11}(d)=-2$

K-armed Bandit Problem

ε-Greedy Action Selection / near-greedy action selection

```
epsilon = 0.05 // small value to control exploration
def get_action():
    if random.random() > epsilon:
        return argmaxa(Q(a))
    else:
        return random.choice(A)
```

- **In the limit as the number of steps increases, every action will be sampled by ε-greedy action selection an infinite number of times. This ensures that all the Qt (a) converge to q_* (a).**
- **Easy to implement / optimize for epsilon / yields good results**

Ex-1: In ε-greedy action selection, for the case of two actions and ε = 0.5, what is the probability that *the greedy action* is selected?

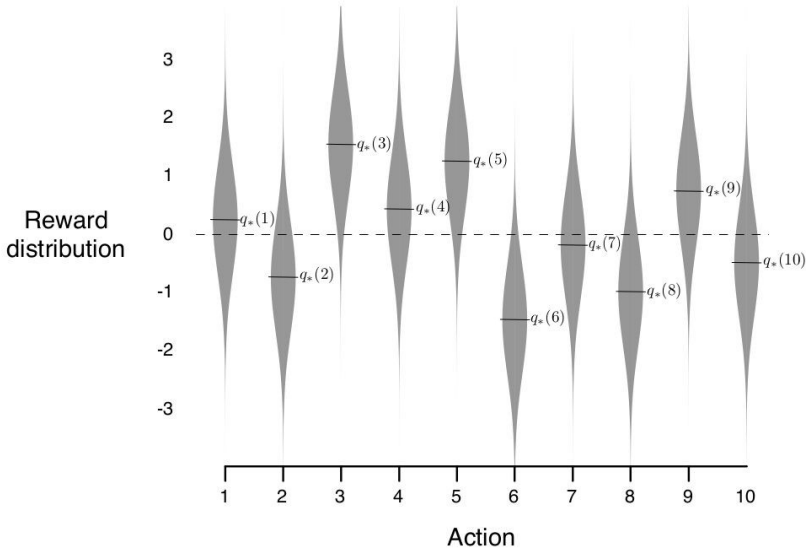
Ex-1: In ε-greedy action selection, for the case of two actions and ε = 0.5, what is the probability that *the greedy action* is selected?

p (greedy action)
= p (greedy action AND greedy selection) + p (greedy action AND random selection)
= p (greedy action | greedy selection) p (greedy selection)
+ p (greedy action | random selection) p (random selection)
= p (greedy action | greedy selection) (1-ε) + p (greedy action | random selection) (ε)
= p (greedy action | greedy selection) (0.5) + p (greedy action | random selection) (0.5)
= (1) (0.5) + (0.5) (0.5)
= 0.5 + 0.25
= 0.75

10-armed Testbed

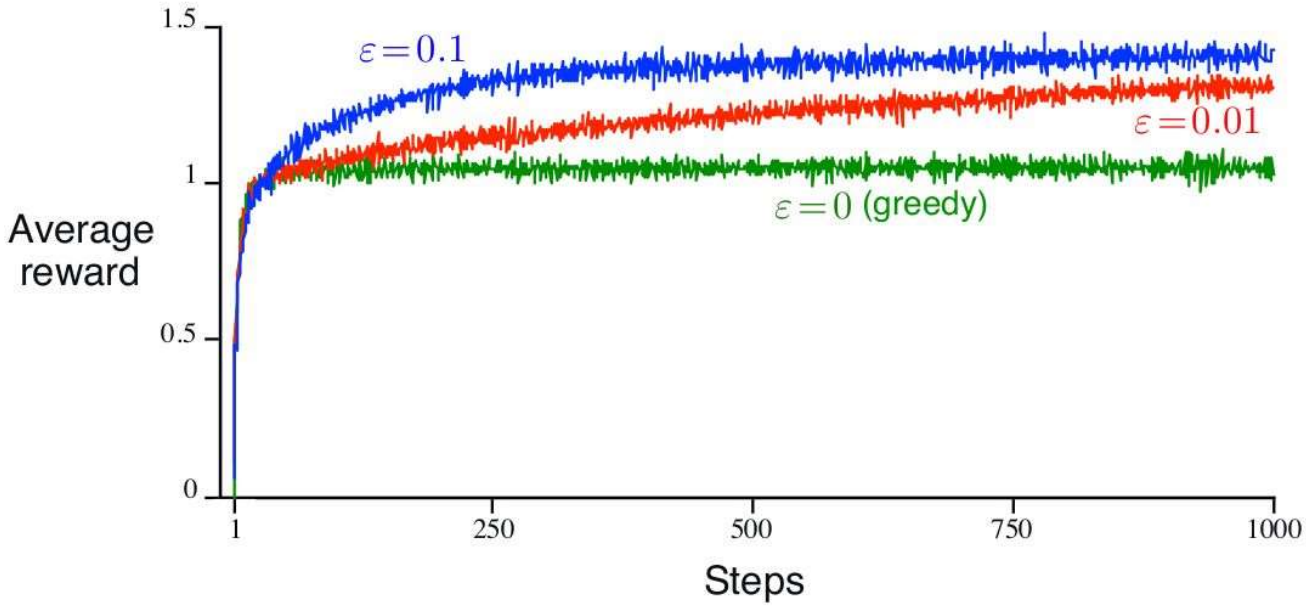
Example:

- A set of 2000 randomly generated k -armed bandit problems with k = 10
- Action values were selected according to a normal (Gaussian) distribution with mean 0 and variance 1.
- While selecting action A_t at time step t, the actual reward, R_t , was selected from a normal distribution with mean $q_*(A_t)$ and variance 1
- One Run : Apply a method for 1000 time steps to one of the bandit problems
- Perform 2000 runs, each run with a different bandit problem, to get an algorithms average behavior

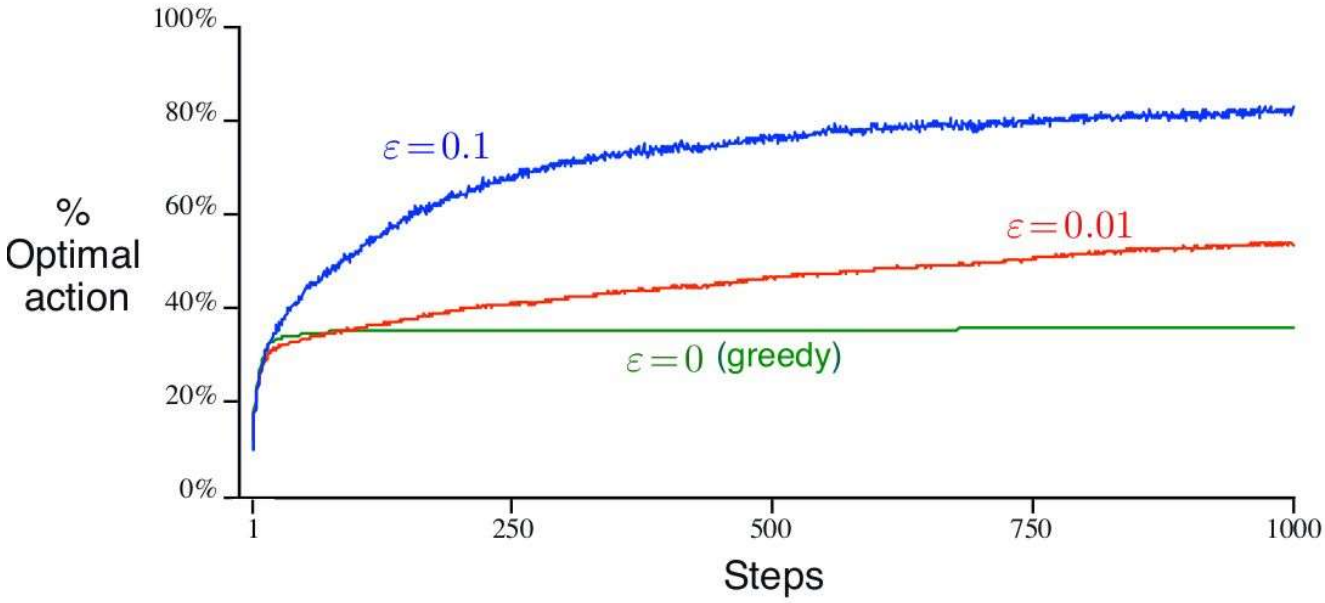


An example bandit problem from the 10-armed testbed

Average performance of ε-greedy action-value methods on the 10-armed testbed



Average performance of ε-greedy action-value methods on the 10-armed testbed



Discussion on Exploration vs. Exploitation

1) What if the reward variance is
   a. larger, say 10 instead of 1?
   b. zero ? [deterministic]

2) What if the bandit task is non-stationary? [that is, the true values of the actions changed over time]

Ex-2:

Consider a k -armed bandit problem with k = 4 actions, denoted 1, 2, 3, and 4.

Consider applying to this problem a bandit algorithm using ε-greedy action selection, sample-average action-value estimates, and initial estimates of Q1 (a) = 0, for all a.

Suppose the initial sequence of actions and rewards is A1 = 1, R1 = 1, A2 = 2, R2 = 1, A3 = 2, R3 = 2, A4 = 2, R4 = 2, A5 = 3, R5 = 0.

On some of these time steps the ε case may have occurred, causing an action to be selected at random.

On which time steps did this definitely occur? On which time steps could this possibly have occurred?

Incremental Implementation

- Efficient approach to compute the estimate of action-value;

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}.$$

- Given Qn and the nth reward, Rn , the new average of all n rewards can be computed as follows

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big],
\end{aligned}$$

Incremental Implementation

Note:

- StepSize decreases with each update
- We use α or $\alpha_t(a)$ to denote step size (constant / varies with each step)

Discussion:

Const vs. Variable step size?

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big],
\end{aligned}$$

$$NewEstimate \leftarrow OldEstimate \; + \; StepSize \Big[Target - OldEstimate\Big]$$

Bandit Algorithm with Incremental Update/ ε-greedy selection

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Non-stationary Problem

- Most RL problems are non-stationary !
- Give more weight to recent rewards than to long-past rewards !!!

$$Q_{n+1} \doteq Q_n + \alpha\left[R_n - Q_n\right]$$

Non-stationary Problem

- Most RL problems are non-stationary !
- Give more weight to recent rewards than to long-past rewards !!!

$$Q_{n+1} \doteq Q_n + \alpha\Big[R_n - Q_n\Big]$$

Exponential recency-weighted average

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha\Big[R_n - Q_n\Big] \\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2\alpha R_{n-2} + \\
&\qquad \cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i.
\end{aligned}$$

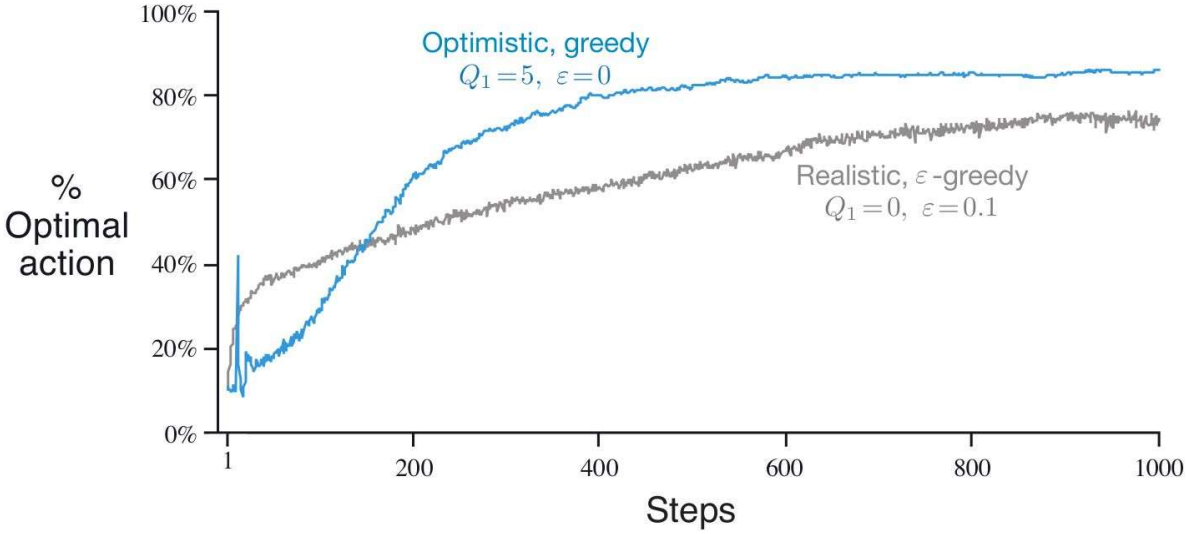
Optimistic Initial Values

- All the above discussed methods are ***biased*** by their initial estimates
- For sample average method the bias disappears once all actions have been selected at least once
- For methods with constant α, the bias is permanent, though decreasing over time
- Initial action values can also be used as a simple way of encouraging exploration.
- In 10 armed testbed, set initial estimate to +5 rather than 0.

  This can encourage action-value methods to explore.

  Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being disappointed with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge.

Optimistic Initial Values



Caution:
Optimistic Initial Values can only be considered as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration.

Question:
Explain how in the non-stationary scenario the optimistic initial values will fail (to explore adequately).

The effect of optimistic initial action-value estimates on the 10-armed testbed.
Both methods used a constant step-size parameter, $\alpha = 0.1$

Upper-Confidence-Bound Action Selection

- **ε**-greedy action selection forces the non-greedy actions to be tried,
  Indiscriminately, with no preference for those that are nearly greedy or particularly uncertain
- It would be better to select among the non-greedy actions according to their potential for actually being optimal
  Take into account both how close their estimates are to being maximal and the uncertainties in those estimates.

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

Upper-Confidence-Bound Action Selection

- Each time a is selected the uncertainty is presumably reduced
- Each time an action other than a is selected, t increases but $N_t(a)$ does not; because t appears in the numerator, the uncertainty estimate increases.
- Actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time

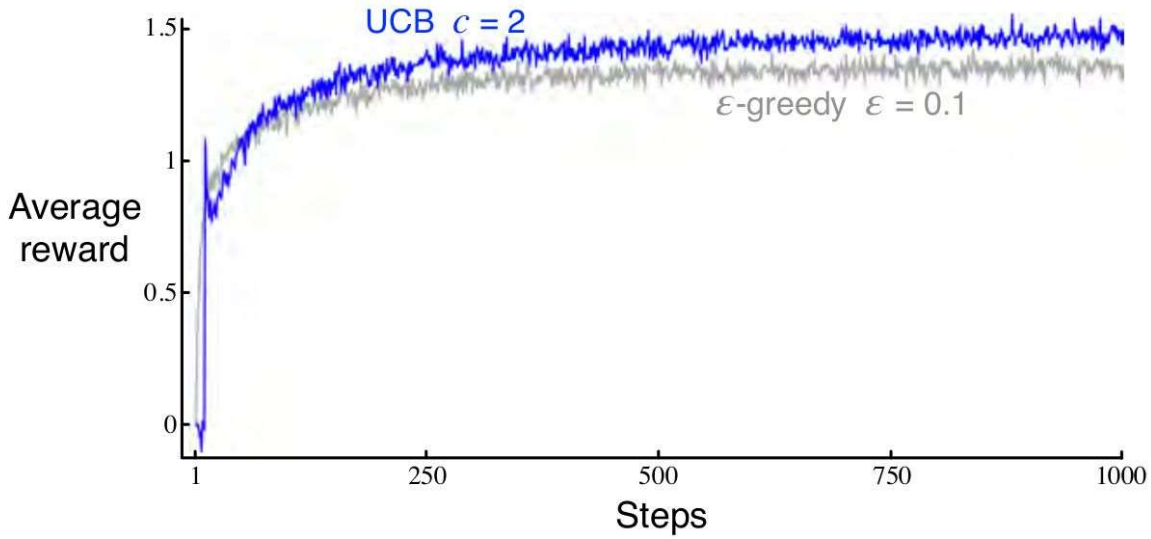
Action Value at time t for a

Confidence Level

Measure of Uncertainty

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

Upper-Confidence-Bound Action Selection



UCB often performs well, as shown here, but is more difficult than "-greedy to extend beyond bandits to the more general reinforcement learning settings

Policy-based algorithms

- Forget about action-value (Q) estimates, we don't really care about them
- We care about what actions to chose

Let's assign a preference to each action and tweak its value

- Define $H_t(a)$ as a numerical preference value associated with action a
- Which action is selected?
  - $A_t = \underset{a}{\text{argmax}}[H_t(a)]$
    - Hardmax results in no exploration -- deterministic action selection

Softmax!

Softmax function

- **Input**: vector of preferences
- **Output**: vector of probabilities forming a valid distribution
- $\Pr\{a_t = a\} = \frac{e^{H_t(a)}}{\sum_{a' \in A} e^{H_t(a')}} = \Pr(a)$
- $H_t \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{bmatrix} 6 \\ 9 \\ 2 \end{bmatrix}$
- $\text{softmax} \begin{pmatrix} 6 \\ 9 \\ 2 \end{pmatrix} = \begin{bmatrix} 0.047 \\ 0.952 \\ 0.00087 \end{bmatrix}$
- That is, with $\Pr(0.95)$ choose a_2, $\Pr(0.05)$ choose a_1, and $< \Pr(0.01)$ choose a_3

Softmax function

- Exploration – checked!
- Softmax provides another important attribute – **a differentiable policy**
- Say that we learn that action a_1 results in good relative performance
- Hardmax: $A_t = \underset{a}{\text{argmax}}[H_t(a_1), H_t(a_2), H_t(a_3)]$
  - Change $H(a_1)$ such that $\Pr(a_1)$ is increased, $\frac{\partial \Pr(a_1)}{\partial H(a_1)} = NA$
- Softmax: $\Pr(a_1) = \frac{e^{H_t(a)}}{\sum_{a' \in A} e^{H_t(a')}}$
  - Change $H(a_1)$ such that $\Pr(a_1)$ is increased -> update towards $\frac{\partial \Pr(a_1)}{\partial H(a_1)}$

Gradient ascend

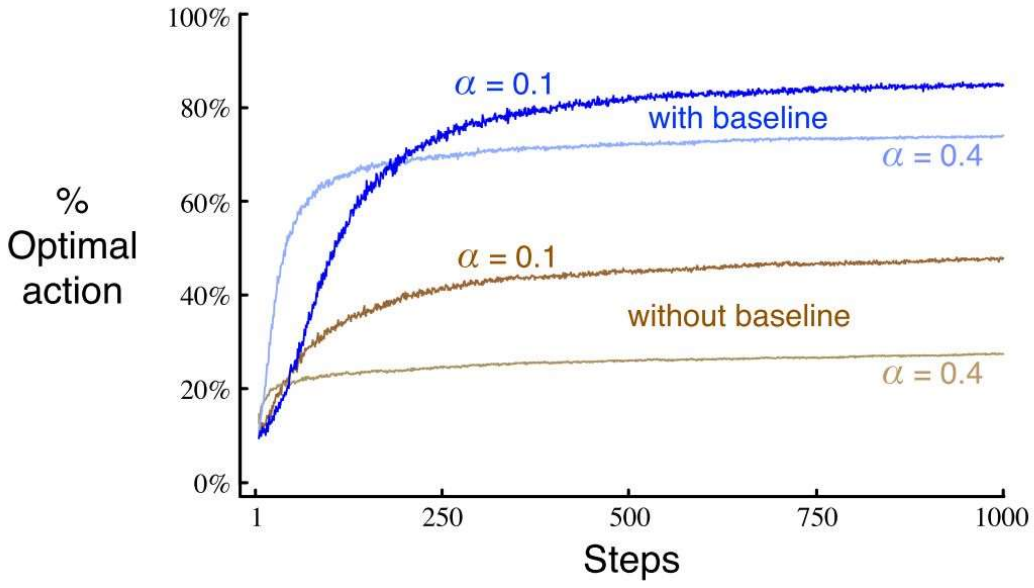
- $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \overline{R_t})\frac{\partial \Pr(A_t)}{\partial H(A_t)}$?
- $\forall a \neq A_t, H_{t+1}(a) = H_t(a) + \alpha(R_t - \overline{R_t})\frac{\partial \Pr(A_t)}{\partial H(a)}$
- Update the preferences based on observed reward and a baseline reward ($\overline{R_t}$)
- If the observed reward is larger than the baseline:
  - Increase the preference of the chosen action, A_t
  - Decrease the preference of all other actions, $\forall a \neq A_t$
- Else do the opposite

Update Rule

On each step, after selecting action A t and receiving the reward Rt ,
Update the action preferences :

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \overline{R_t})(1 - \Pr(A_t))$$
$$\forall a \neq A_t, H_{t+1}(a) = H_t(a) - \alpha(R_t - \overline{R_t})\Pr(a)$$

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha\big(R_t - \bar{R}_t\big)\big(1 - \pi_t(A_t)\big), \qquad \text{and}$$
$$H_{t+1}(a) \doteq H_t(a) - \alpha\big(R_t - \bar{R}_t\big)\pi_t(a), \qquad \text{for all } a \neq A_t$$

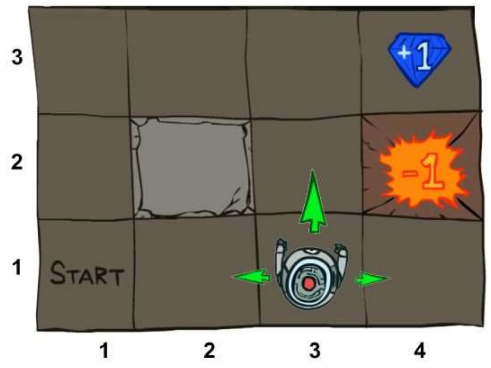


What did we learn?

- **Problem**: choose the action that results in highest expected reward
- **Assumptions**: 1. actions' expected reward is unknown, 2. we are confronted with the same problem over and over, 3. we are able to observe an action's outcome once chosen
- **Approach**: learn the actions' expected reward through exploration (value based) or learn a policy directly (policy based), exploit learnt knowledge to choose best action
- **Methods:** 1. greedy + initializing estimates optimistically, 2. epsilon-greedy, 3. Upper-Confidence-Bounds, 4. gradient ascend + soft-max

A different scenario

- Associative vs. Non-associative tasks ?
- Policy: A mapping from situations to the actions that are best in those situations
- (discuss) How do we extend the solution for non-associative task to an associative task?
  - **Approach**: Extend the solutions to non-stationary task to non-associative tasks
    - Works, if the true action values changes slowly
  - What if the context switching between the situations are made explicit?
    - How?
    - Need Special approaches !!!



Required Readings

1. Chapter-2 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto
2. A Survey on Practical Applications of Multi-Armed and Contextual Bandits, Djallel Bouneffouf , Irina Rish [https://arxiv.org/pdf/1904.10040.pdf]

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you !

Deep Reinforcement Learning
2022-23 Second Semester, M.Tech (AIML)

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Session #4:
Markov Decision Processes

Instructors :
1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Prof. Sangeetha Viswanathan (sangeetha.viswanathan@pilani.bits-pilani.ac.in)

Agenda for the class

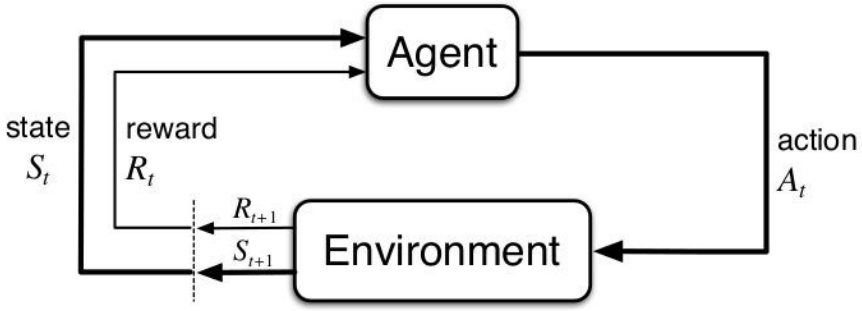
- Agent-Environment Interface (Sequential Decision Problem)
- MDP
  - Defining MDP,
  - Rewards,
  - Returns, Policy & Value Function,
  - Optimal Policy and Value Functions
- Approaches to solve MDP

Announcement !!!
We have our Teaching Assistants now !!!
1. Partha Pratim Saha {parthapratim@wilp.bits-pilani.ac.in}
2. P. Anusha {anusha.p@wilp.bits-pilani.ac.in}

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Agent-Environment Interface

- Agent - Learner & the decision maker
- Environment - Everything outside the agent
- Interaction:
  - Agent performs an action
  - Environment responds by
    - presenting a new situation (change in state)
    - presents numerical reward
- Objective (of the interaction):
  - Maximize the return (cumulative rewards) over time



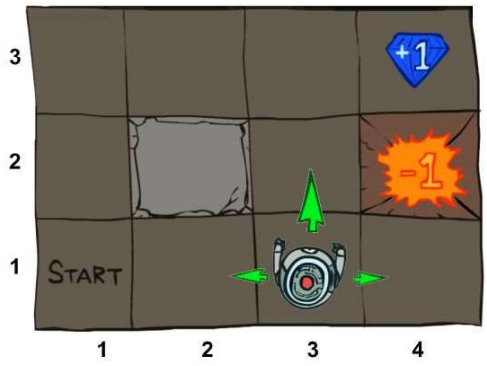
Note:

- Interaction occurs in discrete time steps

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

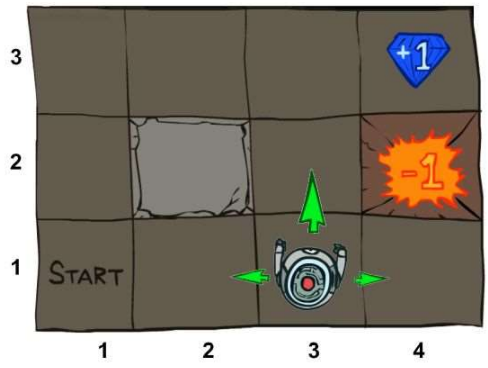
Grid World Example

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - -0.1 per step (battery loss)
  - +1 if arriving at (4,3) ; -1 for arriving at (4,2) ;1 for arriving at (2,2)
- Goal: maximize accumulated rewards



Markov Decision Processes

- An MDP is defined by
  - A set of states
  - A set of actions
  - State-transition probabilities
    - Probability of arriving to after performing at
    - Also called the model dynamics
  - A reward function
    - The utility gained from arriving to after performing at
    - Sometimes just or even
  - A start state
  - Maybe a terminal state



Markov Decision Processes

Model Dynamics

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

State-transition probabilities

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a)$$

Expected rewards for state–action–next-state triples

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)}$$

Markov Decision Processes - Discussion

- *MDP framework is abstract and flexible*
  - Time steps need not refer to fixed intervals of real time
  - The actions can be
    - at low-level controls or high-level decisions
    - totally mental or computational
  - States can take a wide variety of forms
    - Determined by *low-level sensations* or *high-level and abstract* (ex. symbolic descriptions of objects in a room)
- *The agent–environment boundary represents the limit of the agent's absolute control*, not of its knowledge.
  - *The boundary can be located at different places for different purposes*

Markov Decision Processes - Discussion

- *MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction.*
- It proposes that *whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment:*
  - *one signal to represent the choices made by the agent (the actions)*
  - *one signal to represent the basis on which the choices are made (the states),*
  - *and one signal to define the agent's goal (the rewards).*

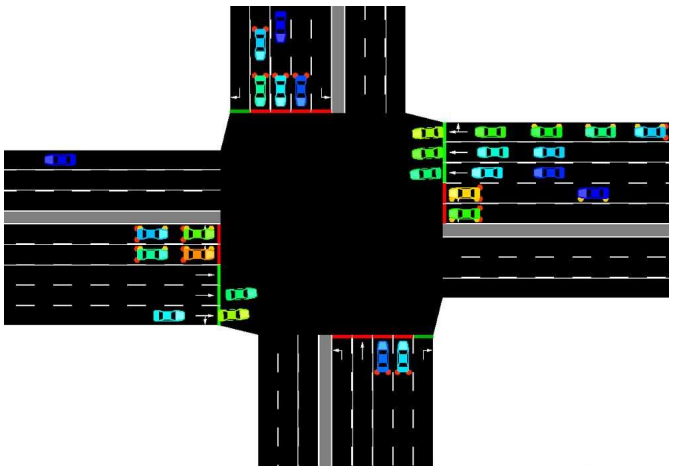
MDP Formalization : Video Games

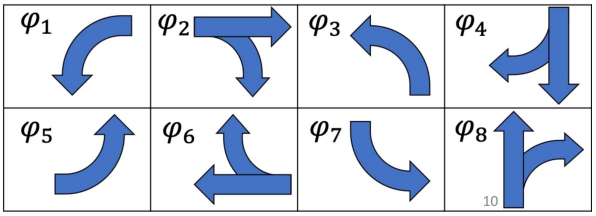
- *State:*
  - *raw pixels*
- *Actions:*
  - *game controls*
- *Reward:*
  - *change in score*
- *State-transition probabilities:*
  - *defined by stochasticity in game evolution*

Ref: Playing Atari with deep reinforcement learning", Mnih et al., 2013

MDP Formalization : Traffic Signal Control

- *State:*
  - *Current signal assignment (green, yellow, and red assignment for each phase)*
  - *For each lane: number of approaching vehicles, accumulated waiting time, number of stopped vehicles, and average speed of approaching vehicles*
- *Actions:*
  - *signal assignment*
- *Reward:*
  - *Reduction in traffic delay*
- *State-transition probabilities:*
  - *defined by stochasticity in approaching demand*





Ref: "Learning an Interpretable Traffic Signal Control Policy", Ault et al., 2020

MDP Formalization : Recycling Robot (Detailed Ex.)

- *Robot has*
  - *sensors for detecting cans*
  - *arm and gripper that can pick the cans and place in an onboard bin;*
- *Runs on a rechargeable battery*
- *Its control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper*
- *Task for the RL Agent: Make high-level decisions about how to search for cans based on the current charge level of the battery*

MDP Formalization : Recycling Robot (Detailed Ex.)

- *State:*
  - *Assume that only two charge levels can be distinguished*
  - *S = {high, low}*
- *Actions:*
  - *A(high) = {search, wait}*
  - *A(low) = {search, wait, recharge}*
- *Reward:*
  - *Zero most of the time, except when securing a can*
  - *Cans are secured by searching and waiting, but* $r_{search} > r_{wait}$
- *State-transition probabilities:*
  - *[Next Slide]*

MDP Formalization : Recycling Robot (Detailed Ex.)

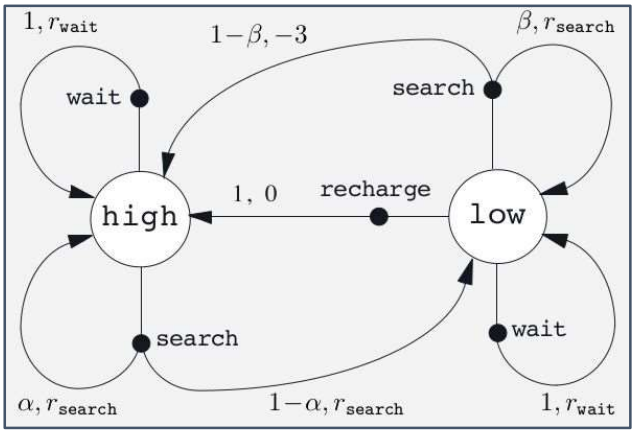
- *State-transition probabilities (contd…):*

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1-\alpha$	$r_{\texttt{search}}$
low	search	high	$1-\beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-

MDP Formalization : Recycling Robot (Detailed Ex.)

- *State-transition probabilities (contd…):*

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1-\alpha$	$r_{\texttt{search}}$
low	search	high	$1-\beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-



Note on Goals & Rewards

- *Reward Hypothesis:*

  > *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

- *The rewards we set up truly indicate what we want accomplished,*
  - *not the place to impart prior knowledge on how we want it to do*
- *Ex: Chess Playing Agent*
  - *If the agent is rewarded for taking opponents pieces, the agent might fall for the opponent's trap.*
- *Ex: Vacuum Cleaner Agent*
  - *If the agent is rewarded for each unit of dirt it sucks, it can repeatedly deposit and suck the dirt for larger reward*

Returns & Episodes

- *Goal is to maximize the expected return*
- *Return (Gt) is defined as some specific function of the reward sequence*
- *Episodic tasks vs. Continuing tasks*
- *When there is a notion of final time step, say T, return can be*

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

  - *Applicable when agent-environment interaction breaks into episodes*
  - *Ex: Playing Game, Trips through maze etc. [called episodic tasks]*

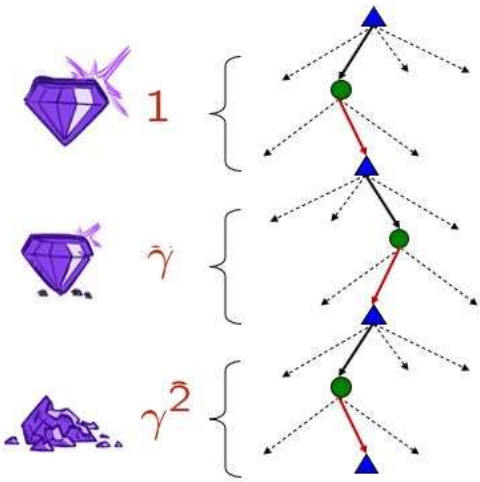
Returns & Episodes

- *Generally T = ∞*
  - *What if the agent receive a reward of +1 for each timestep?*
  - *Discounted Return:*

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Note: γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*

  - *Discount rate determines the present value of future rewards*



Returns & Episodes

- *What if γ is 0?*
- *What if γ is 1?*
- *Computing discounted rewards incrementally*

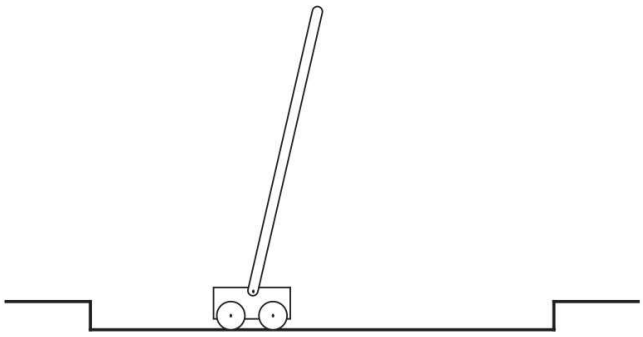
$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

- *Sum of an infinite number of terms, it is still finite if the reward is nonzero and constant and if $\gamma < 1$.*
- *Ex: reward is +1 constant*

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}.$$

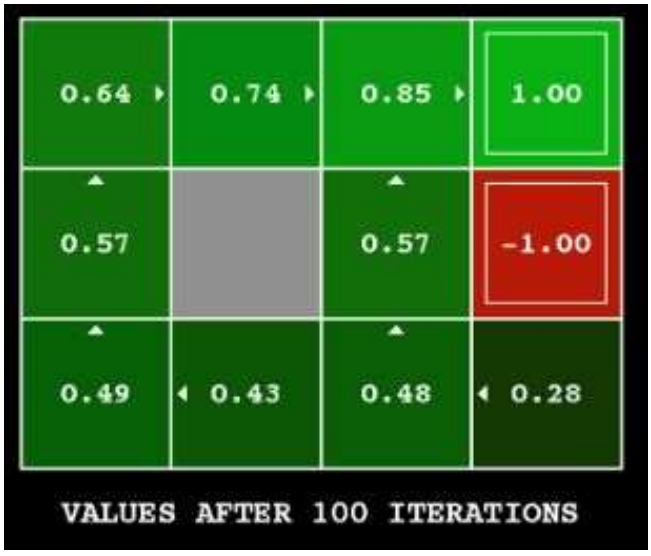
Returns & Episodes

- ➔ *Objective: To apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over*
- ➔ *Discuss:*
  - ➔ *Consider the task as episodic, that is try/maintain balance until failure. What could be the reward function?*
  - ➔ *Repeat prev. assuming task is continuous.*



Policy

- A mapping from states to probabilities of selecting each possible action.
  - $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$
- The purpose of learning is to improve the agent's policy with its experience



Defining Value Functions

State-value function for policy π

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

Action-value function for policy π

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Defining Value Functions

State Value function in terms of Action-value function for policy π

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a)$$

Action Value function in terms of State value function for policy π

$$q_\pi(s,a) = \sum_{s \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma v_\pi(s')\right]$$

Bellman Equation for V_π

May skip to the next slide !

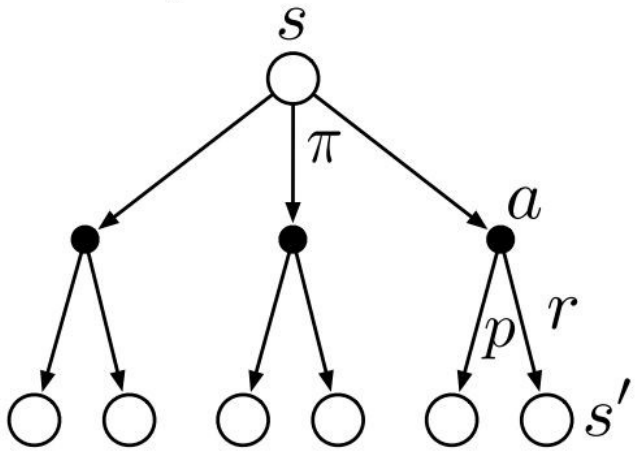
- Dynamic programming equation associated with discrete-time optimization problems
  - Expressing V_π recursively i.e. relating $V_\pi(s)$ to $V_\pi(s')$ for all $s' \in$ succ(s)

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}
\end{aligned}
$$

Bellman Equation for V_π

$$
v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big]
$$

Value of the start state must equal
(1) the (discounted) value of the expected next state,
plus
(1) the reward expected along the way

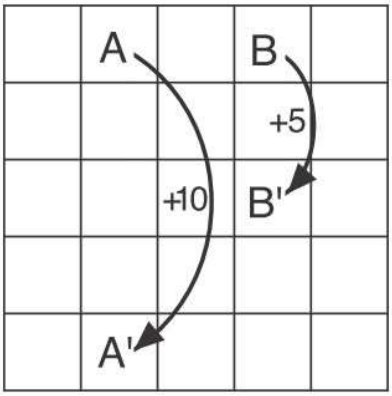


Backup Diagram

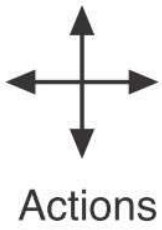
Understanding $V_\pi(s)$ with Gridworld

Reward:

- -1 if an action takes agent off the grid
- Exceptional reward from A and B for all actions taking agent to A' and B' resp.
- 0, everywhere else



Exceptional reward dynamics



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

State-value function for the equiprobable random policy with $\gamma = 0.9$

Understanding $V_\pi(s)$ with Gridworld

$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big]$$

Verify $V_\pi(s)$ using Bellman equation for this state with $\gamma = 0.9$, and equiprobable random policy

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Understanding $V_\pi(s)$ with Gridworld

$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \\ &= \sum_a 0.25 \cdot \left[0 + 0.9 \cdot (2.3 + 0.4 - 0.4 + 0.7)\right] \\ &= 0.25 \cdot [0.9 \cdot 3.0] = 0.675 \approx 0.7 \end{aligned}$$

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Ex-1

Recollect the reward function used for Gridworld as below:

- -1 if an action takes agent off the grid
- Exceptional reward from A and B for all actions taking agent to A' and B' resp.
- 0, everywhere else

Let us add a constant c (say 10) to the rewards of all the actions. Will it change anything?

Optimal Policies and Optimal Value Functions

- $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$
- There is always at least one policy that is better than or equal to all other policies → optimal policy (denoted as π_*)
  - There could be more than one optimal policy !!!

Optimal state-value function $v_*(s) \doteq \max_\pi v_\pi(s)$

Optimal action-value function $q_*(s,a) \doteq \max_\pi q_\pi(s,a)$

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

Optimal Policies and Optimal Value Functions

Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

Bellman optimality equation for V_*

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big].
\end{aligned}$$

Optimal Policies and Optimal Value Functions

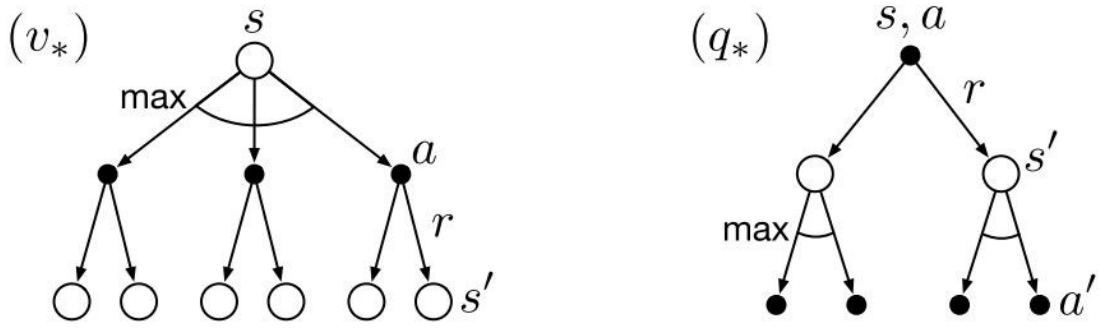
Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

Bellman optimality equation for q_*

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right].
\end{aligned}
$$

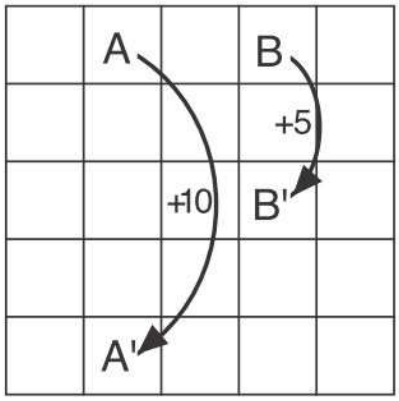
Optimal Policies and Optimal Value Functions

Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*



Backup diagrams for v* and q*

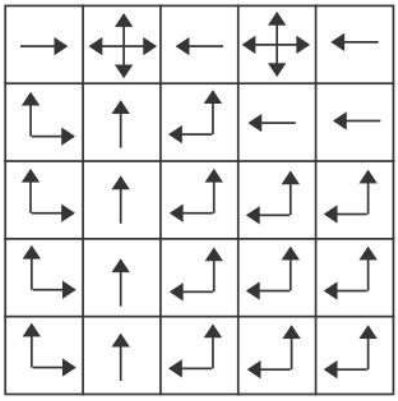
Optimal solutions to the gridworld example



Gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

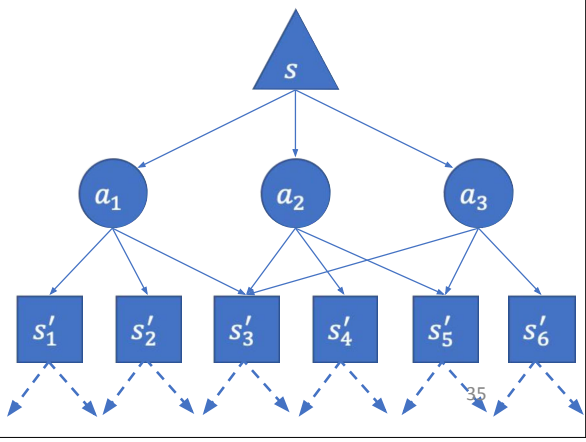
Backup diagrams for v* and q*

Break for 5 mins

MDP - Objective

A set of states $s \in \mathcal{S}$
A set of actions $a \in \mathcal{A}$
State-transition probabilities $P(s'|s,a)$
A reward function $R(s,a,s')$

- Compute a policy: what action to take at each state
  - $\pi: S \rightarrow A$
- Compute the **optimal** policy: maximum expected reward, π^*
- $\pi^*(s) = ?$
- $= \underset{a}{\text{argmax}}[\sum_{s'} P(s'|s,a)R(s,a,s')]$
  - Must also optimize over the future (next steps)
- $= \underset{a}{\text{argmax}}[\sum_{s'} P(s'|s,a)\,(R(s,a,s') + \underbrace{\mathbb{E}_{\pi^*}[G|s']}_{v^*(s')})]$



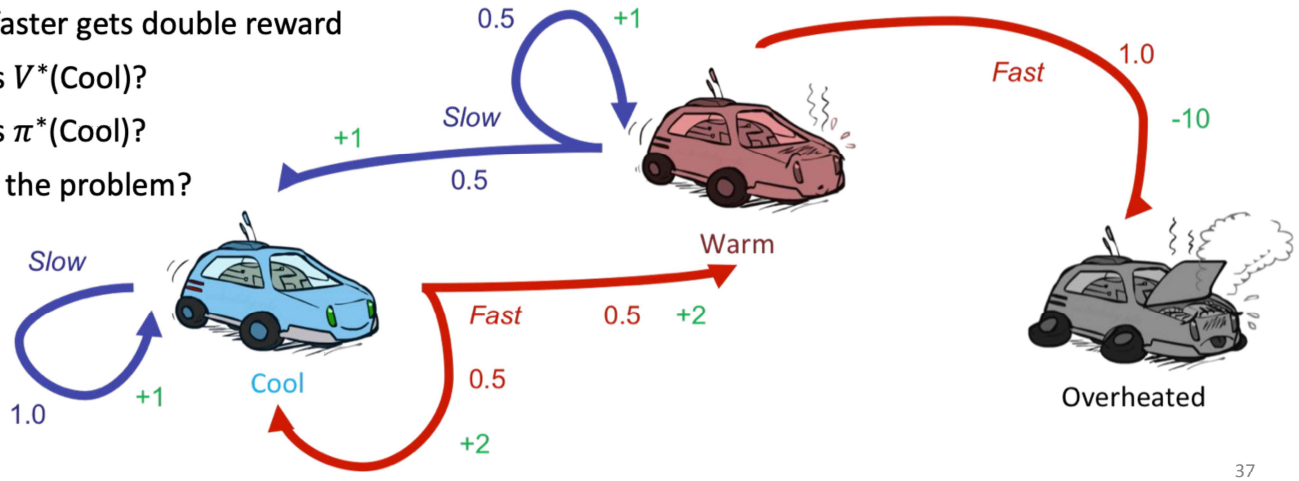
Notation

- π^* - a policy that yields the maximal expected sum of rewards
- G- observed sum of rewards, i.e., $\sum r_t$
- $v^*(s)$ - the expected sum of rewards from being at s then following π^*
  - $= \mathbb{E}_{\pi^*}[G|s]$

Race car example

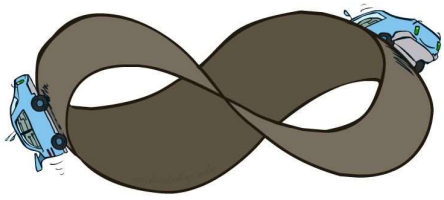
$$\max_a \left[\sum_{s'} P(s'|s,a) \left(R(s,a,s') + v^*(s') \right) \right]$$

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward
- What is V^*(Cool)?
- What is π^*(Cool)?
- What's the problem?



Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)
  - Discounting: use $0 < \gamma < 1$



$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

  - Smaller γ means smaller "horizon" – shorter term focus

Discount factor

- As the agent traverse the world it receives a sequence of rewards
- Which sequence has higher utility?
  - $\tau_1 = +1, +1, +1, +1, +1, +1 \ldots$
  - $\tau_2 = +2, +2, +2, +2, +2, +2 \ldots$

$$\sum_{t=0}^{\infty} 1 = \sum_{t=0}^{\infty} 2 = \infty$$

- Let's decay future rewards exponentially by a factor, $0 \leq \gamma < 1$

$$\sum_{t=0}^{\infty} \gamma^t r = \frac{r}{1-\gamma}$$ **Geometric series**

- Now τ_1 yields higher utility than τ_2

Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- Discount factor: values of rewards decay exponentially

1

Worth Now

γ

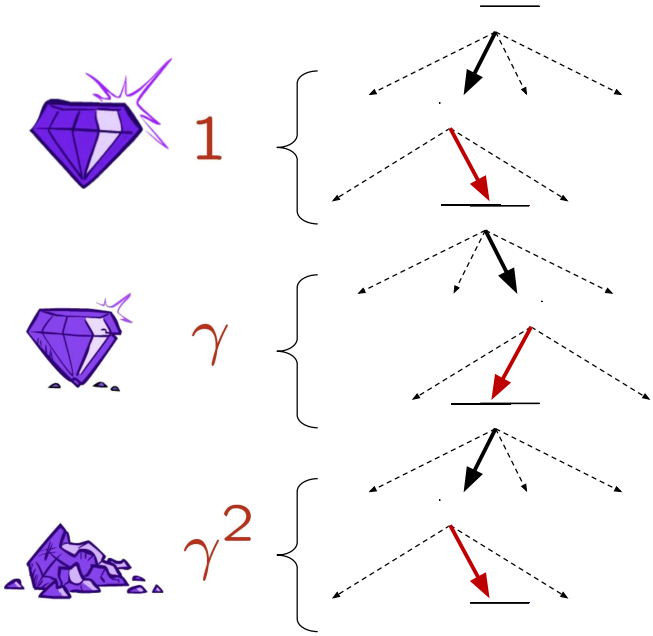
Worth Next Step

γ^2

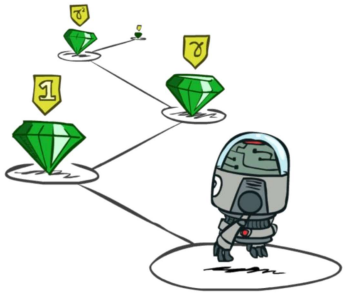
Worth In Two Steps

Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge
- Example: discount of 0.5
  - G(r=[1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - G([1,2,3]) < G([3,2,1])



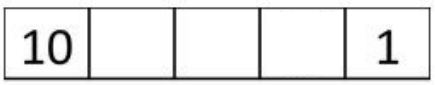
Quiz: Discounting



- Given grid world:

  - Actions: East, West, and Exit ('Exit' only available in terminal states: a, e)
  - Rewards are given only after an exit action
  - Transitions: deterministic
- Quiz 1: For γ = 1, what is the optimal policy?

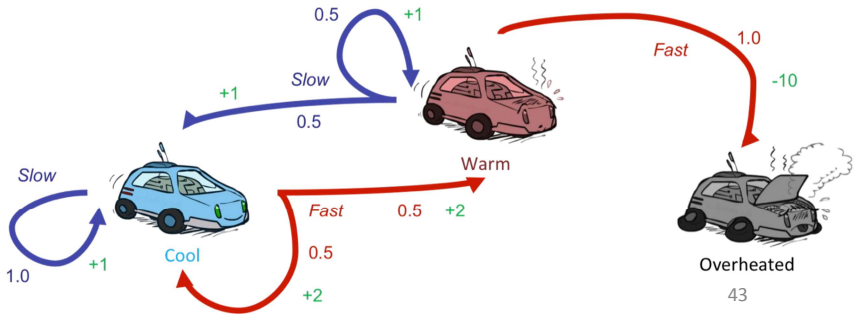
- Quiz 2: For γ = 0.1, what is the optimal policy?



- Quiz 3: For which γ are West and East equally good when in state d?

Race car example

- Consider a discount factor, $\gamma = 0.9$
- What is $v^*(Cool)$
- $= \max_a \left[r(s,a) + \sum_{s'} p(s'|s,a)\ \gamma\ v^*(s') \right]$
- $= \max[1 + 0.9 \cdot 1 v^*(Cool), 2 + 0.9 \cdot 0.5 v^*(Cool) + 0.9 \cdot 0.5 v^*(Warm)]$
  - Computing…
  - …Stack overflow
- Work in iterations



Value iteration

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

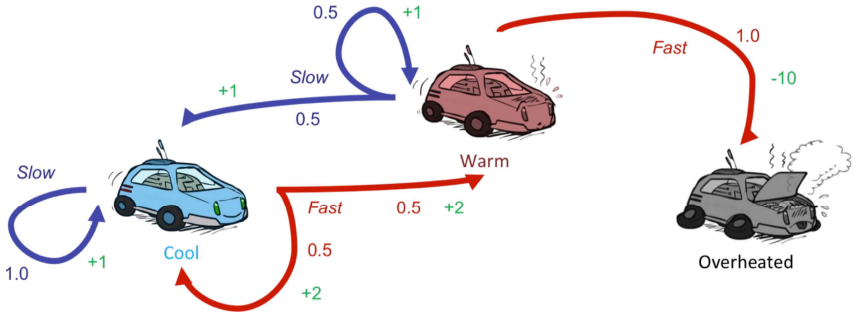
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Value Iteration

V_0	0	0	0
V_1	2	1	0
V_2	3.35	2.35	0

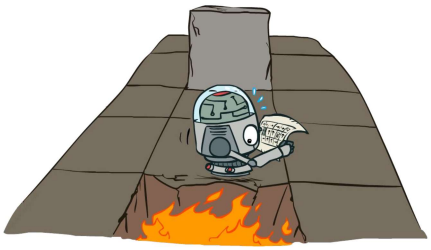
Check this computation on paper.



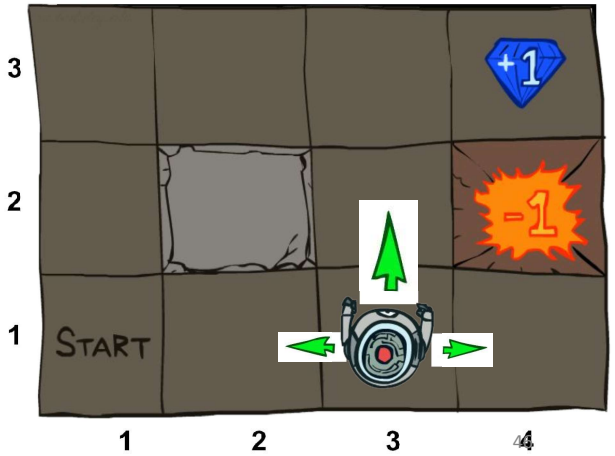
$\gamma = 0.9$

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right]$$

Example: Grid World

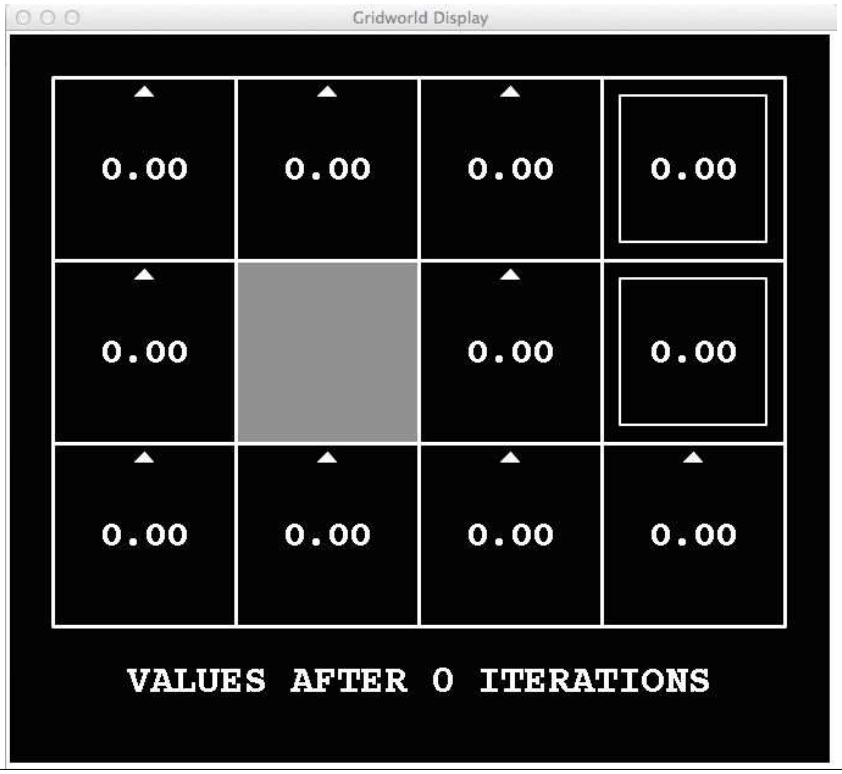


- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small negative reward each step (battery drain)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of (discounted) rewards



$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

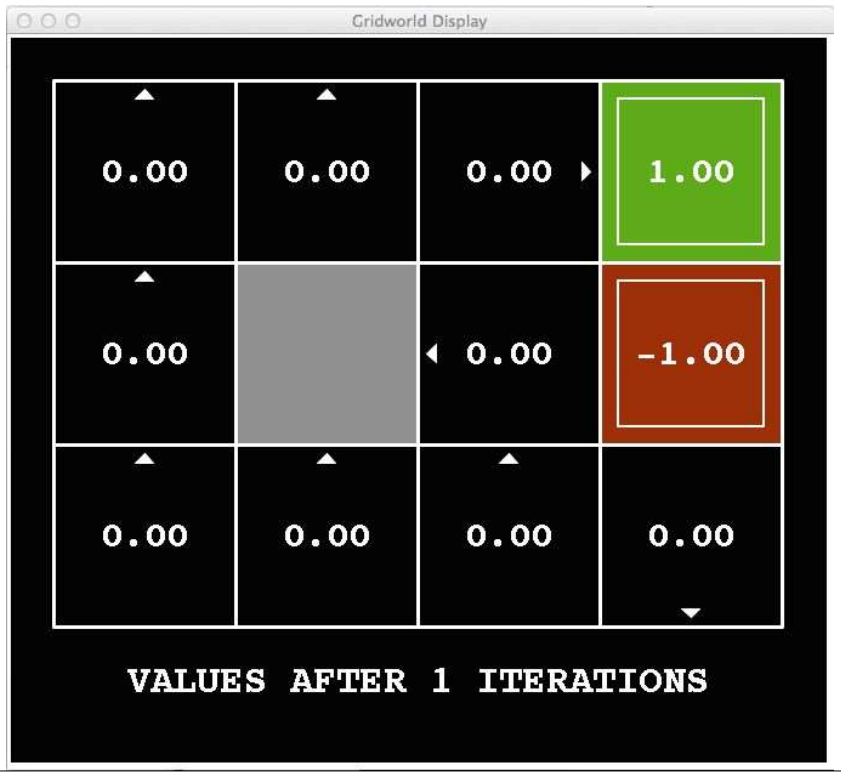
k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

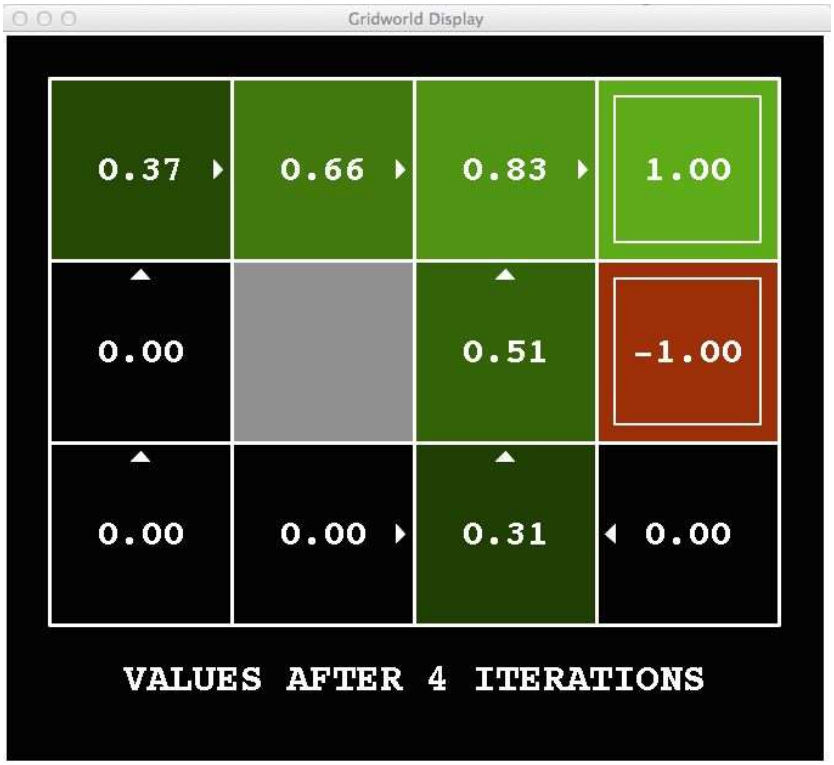
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

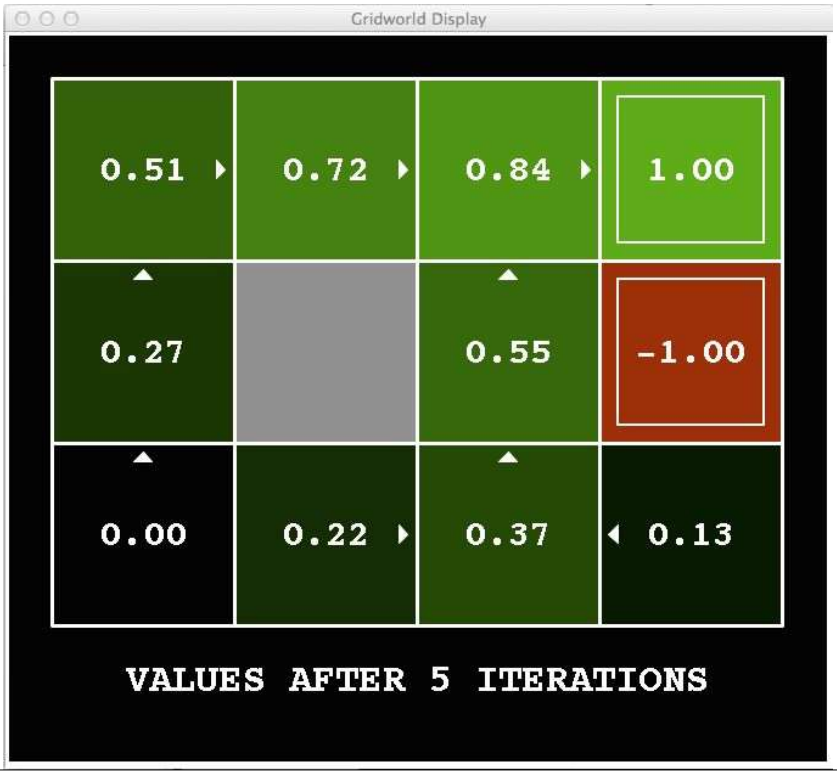
k=4



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

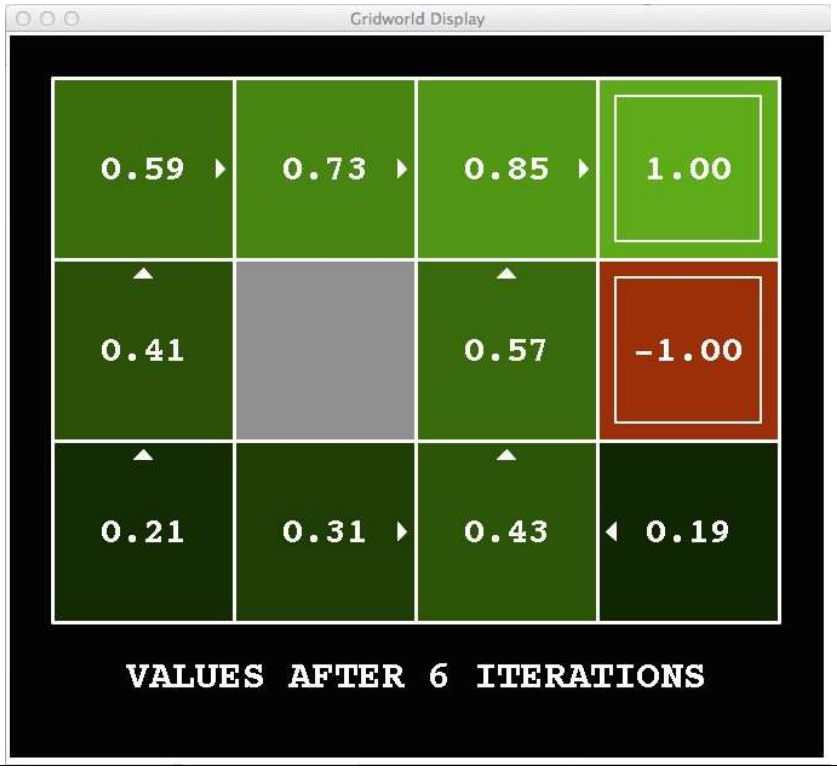
k=5



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

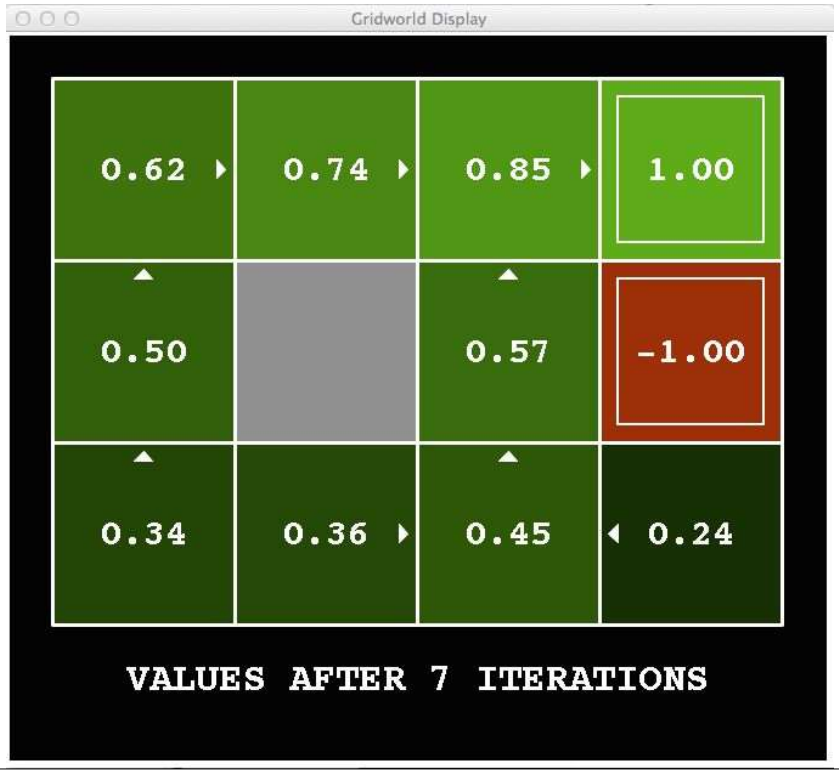
k=6



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

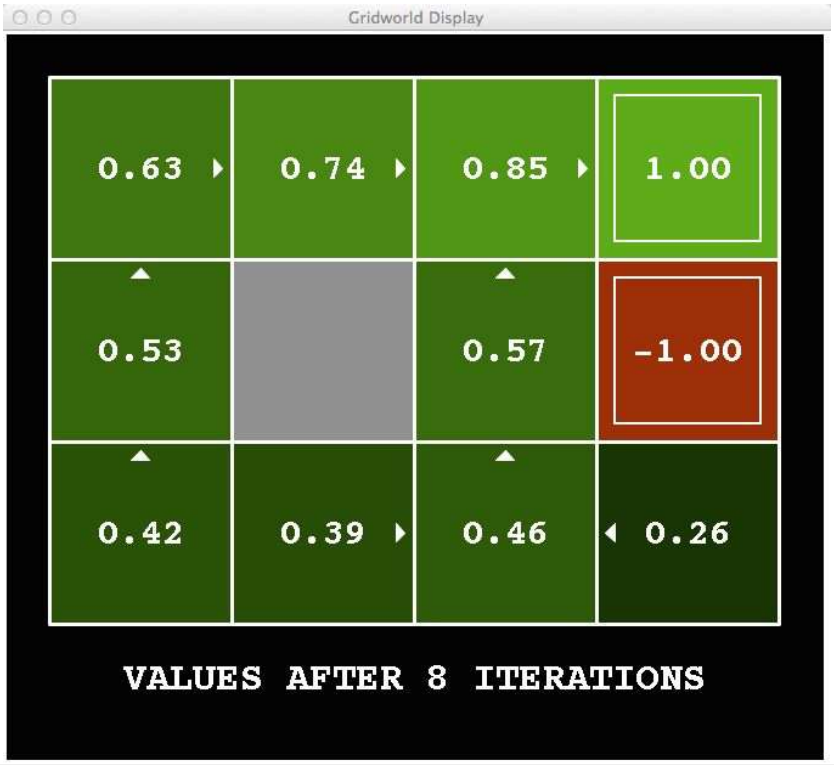
k=7



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

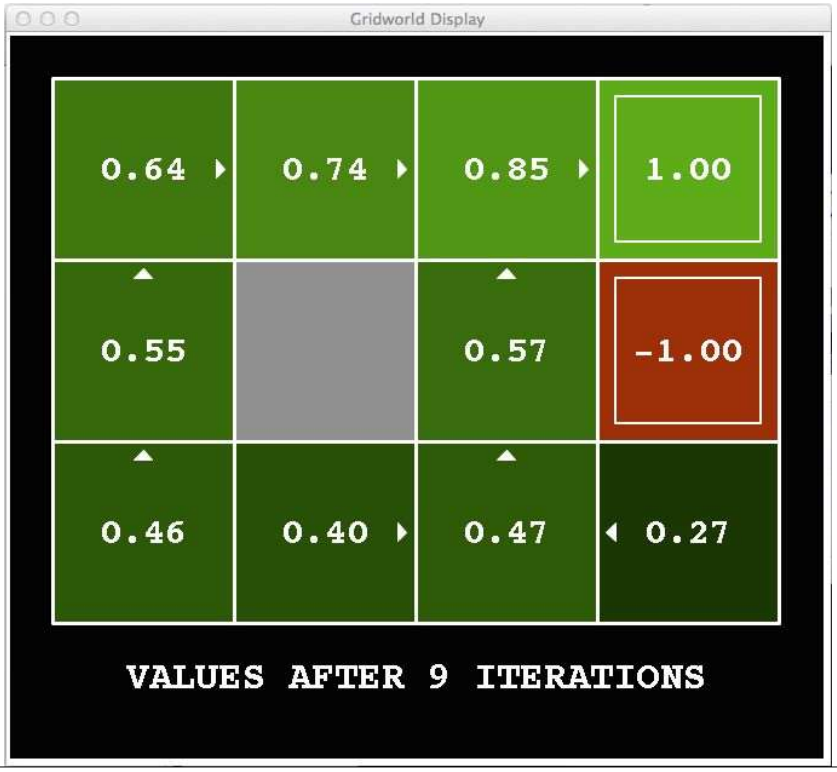
k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

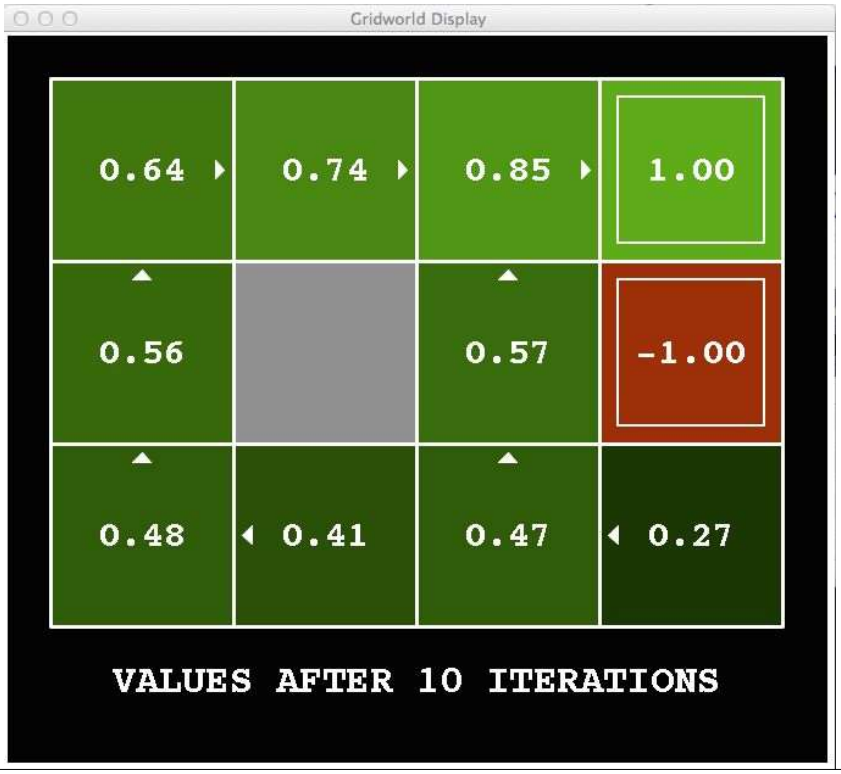
k=9



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

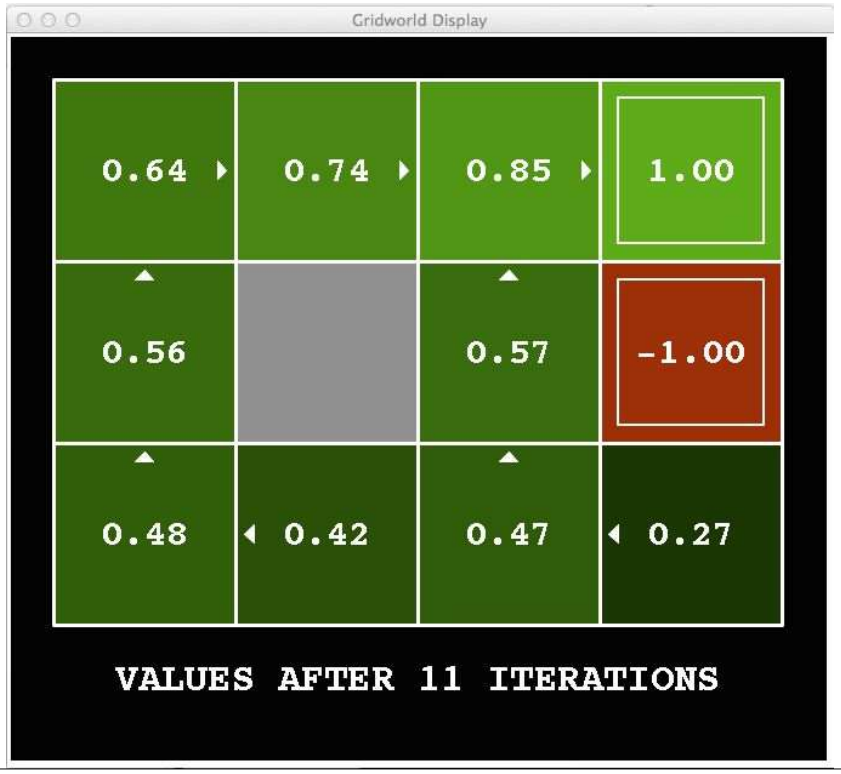
k=10



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

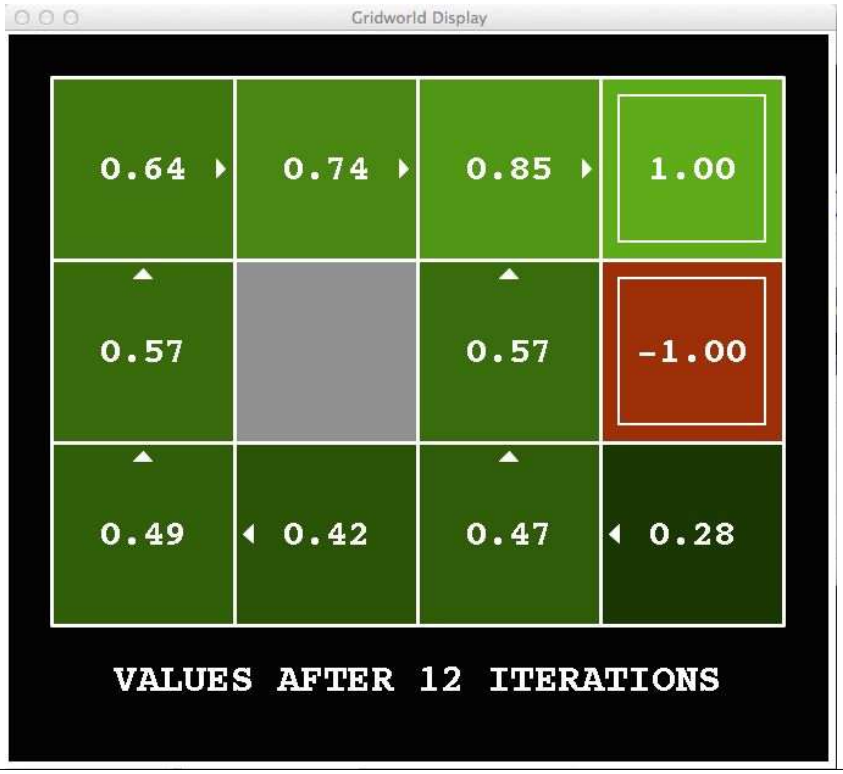
k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

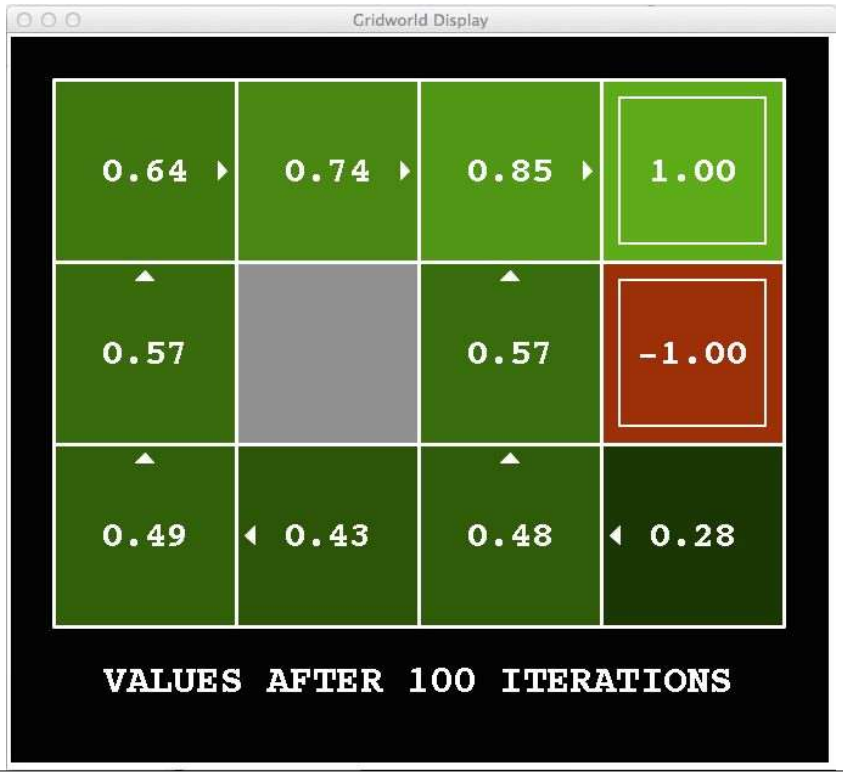
k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Problems with Value Iteration

- Value iteration repeats the Bellman updates:
- $V_{k+1}(s) \leftarrow \max_a \left[R(s,a) + \sum_{s'} P(s'|s,a)\ \gamma V_k(s')\right]$

$$= \max_a \left[\sum_{s'} P(s'|s,a)\ (R(s,a,s') + \gamma V_k(s'))\right]$$

- Issue 1: It's slow – $O(S^2A)$ per iteration
  - Do we really need to update every state at every iteration?
- Issue 2: A policy cannot be easily extracted
  - Policy extraction requires another $O(S^2A)$
- Issue 3: The policy often converges long before the values
  - Can we identify when the policy converged?
- Issue 4: requires knowing the model, $P(s'|s,a)$, and the reward function, $R(s,a)$
- Issue 5: requires discrete (finite) set of actions
- Issue 6: infeasible in large state spaces

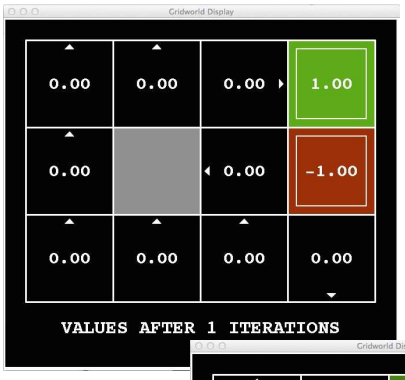
Solutions (briefly, more later…)

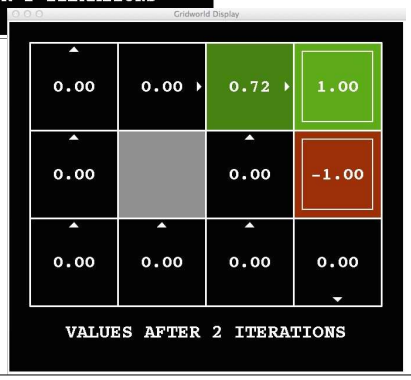
- Issue 1: It's slow – $O(S^2A)$ per iteration
  - Asynchronous value iteration
- Issue 2: A policy cannot be easily extracted
  - Learn q (action) values
- Issue 3: The policy often converges long before the values
  - Policy-based methods
- Issue 4: requires knowing the model and the reward function
  - Reinforcement learning
- Issue 5: requires discrete (finite) set of actions
  - Policy gradient methods
- Issue 6: infeasible for large (or continues) state spaces
  - Function approximators

Issue 1: It's slow – $O(S^2A)$ per iteration

- Asynchronous value iteration
- In value iteration, we update every state in each iteration
- Actually, *any* sequences of Bellman updates will converge if every state is visited infinitely often regardless of the visitation order
- Idea: prioritize states whose value we expect to change significantly

Asynchronous Value Iteration





- Which states should be prioritized for an update?

A single update per iteration

Algorithm 3 Prioritized Value Iteration

1: **repeat**
2: $s \leftarrow \arg\max_{\xi \in S} H(\xi)$
3: $V(s) \leftarrow \max_{a \in A} \{R(s,a) + \gamma \sum_{s' \in S} Pr(s'|s,a)V(s')\}$
4: **for all** $s' \in SDS(s)$ **do**
5: // recompute $H(s')$
6: **end for**
7: **until** convergence

$$SDS(s) = \{s' : \exists a, p(s|s',a) > 0\}$$

For the home assignment set:

$$H(s') = \left| V(s') - \max_{a} \left\{ R(s',a) + \gamma \sum_{s''} \Pr(s''|s',a)V(s'') \right\} \right|$$

Double the work?

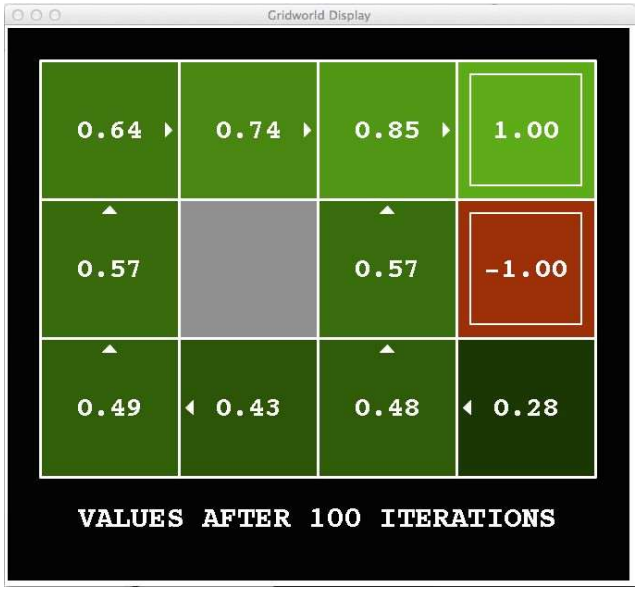
For the home assignment set:

$$H(s') = \left| V(s') - \max_{a} \left\{ R(s', a) + \gamma \sum_{s''} \Pr(s''|s', a) V(s'') \right\} \right|$$

- Computing priority is similar to updating the state value (W.R.T computational effort)
- Why do double work?
  - If we computed the priority, we can go ahead and update the value for free
- Notice that we don't need to update the priorities for the entire state space
- For many of the states the priority doesn't change following an updated value for a single state s
- Only states s' with $\sum_a p(s'|s,a) > 0$ require update

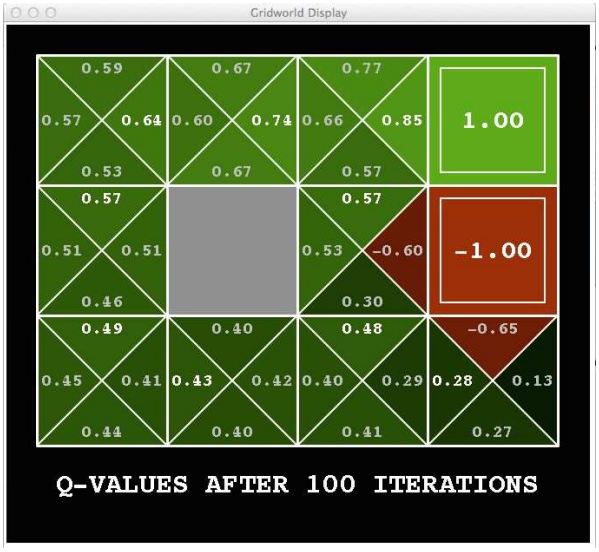
Issue 2: A policy cannot be easily extracted

- Given state values, what is the appropriate policy?
  - $\pi(s) \leftarrow \underset{a}{\text{argmax}}\left[R(s,a) + \sum_{s'} P(s'|s,a)\ \gamma V_k(s')\right]$
  - Requires another full value sweep: $O(S^2A)$
- Learn q (action) values instead
- $Q^*(s,a)$ - the expected sum of rewards from being at s, taking action a and then following π^*



Q-learning

- $Q^*(s, a)$ - the expected sum of rewards from being at s, taking action a and then following π^*
- $\pi^*(s) \leftarrow \underset{a}{\text{argmax}}[Q^*(s, a)]$
- Can we learn Q values with dynamic programming?
  - Yes, similar to value iteration



Q-learning as value iteration

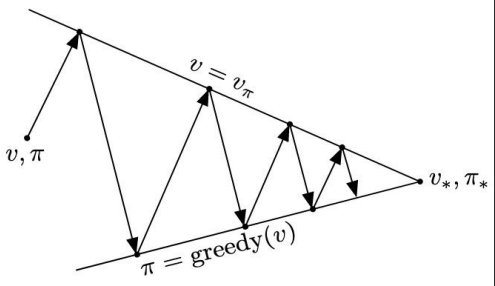
- $V^*(s) \coloneqq \max_a[\sum_{s'} P(s'|s, a)\,(R(s, a, s') + \gamma V^*(s'))]$
- $V^*(s) \coloneqq \max_a[Q^*(s, a)]$
- $Q^*(s, a) \coloneqq \sum_{s'} P(s'|s, a)\,(R(s, a, s') + \gamma V^*(s'))$
- $Q^*(s, a) = \sum_{s'} P(s'|s, a)\,(R(s, a, s') + \gamma \max_a[Q^*(s', a)])$
- Solve iteratively
  - $Q_{k+1}(s, a) = \sum_{s'} P(s'|s, a)\,(R(s, a, s') + \gamma \max_a[Q_k(s', a)])$
  - Can also use Asynchronous learning

Issue 3: The policy often converges long before the values

- Value iteration converges to the true utility value: $V_{k\to\infty} \to V^*$
- V^* implies the optimal policy: π^*
- Can we converge directly on π^*?
  - Improve the policy in iteration until reaching the optimal one

Policy Iteration

1. Compute V_π: calculate state value for some fixed policy (not necessarily the optimal values, $V_\pi \neq V^*$)
2. Update π: update policy using one-step look-ahead with the resulting (non optimal) values
3. Repeat until policy converges
   (optimal values and policy)



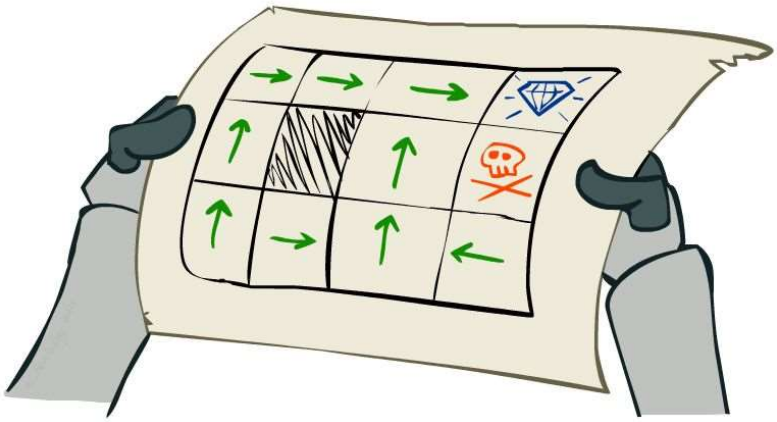
- Guaranteed converges to π^*
  - $\forall s, V_{k>0}(s) \leq V_{k+1}(s)$ i.e., π_i improves monotonically with i
  - A fixed point, $\forall s, V_k(s) = V_{k+1}(s)$, implies π^*

Policy Evaluation

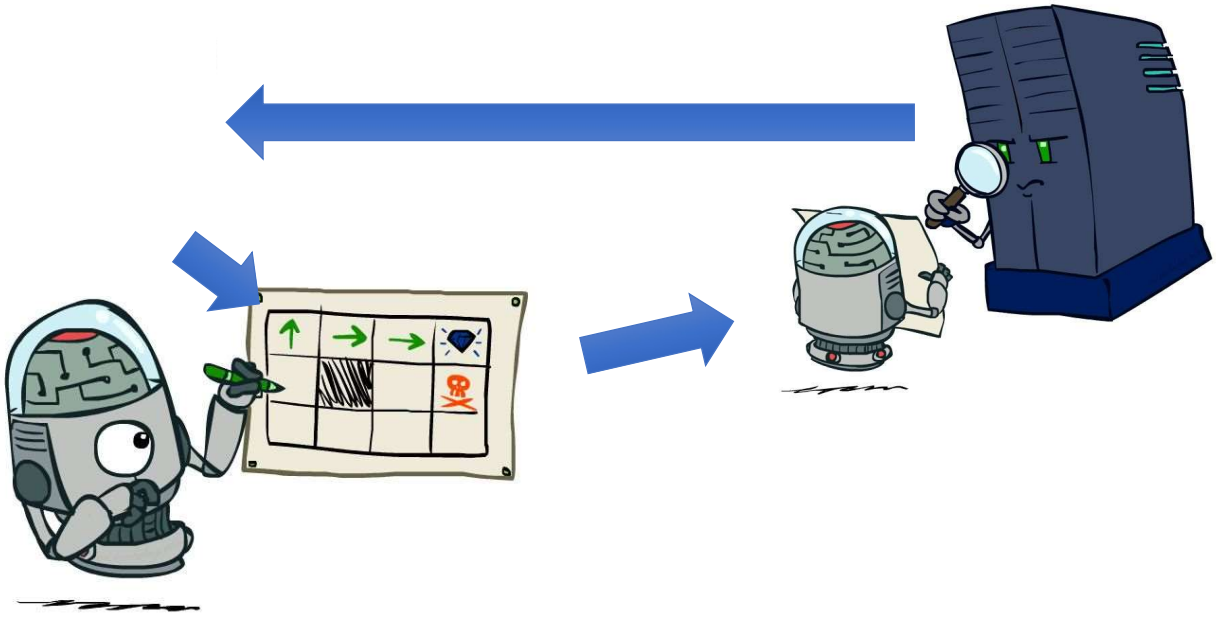
- Why is calculating V_π easier that calculating V^*?
  - Turns non-linear Bellman equations into linear equations
- $v^*(s) = \max_a [\sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma v^*(s'))]$
- $v_\pi(s) = \sum_{s'} P(s'|s,\pi(s))\,(R(s,\pi(s),s') + \gamma v^*(s'))$
- Solve a set of linear equations in $O(S^2)$
  - Solve with Numpy (numpy.linalg.solve)
  - Required for your home assignment
  - See: https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html#numpy.linalg.solve

Policy value as a Linear program

- $v_{11} = 0.8 \cdot (-0.1 + 0.95 \cdot v_{12}) + 0.1 \cdot (-0.1 + 0.95 \cdot v_{21}) + 0.1 \cdot (-0.1 + 0.95 \cdot v_{11})$
- $v_{12} = 0.8 \cdot (-0.1 + 0.95 \cdot v_{13}) + 0.2 \cdot (-0.1 + 0.95 \cdot v_{12})$
- ...
- $v_{42} = -1$
- $v_{43} = 1$



Policy iteration

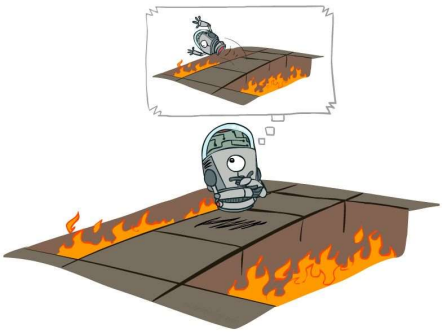


Comparison

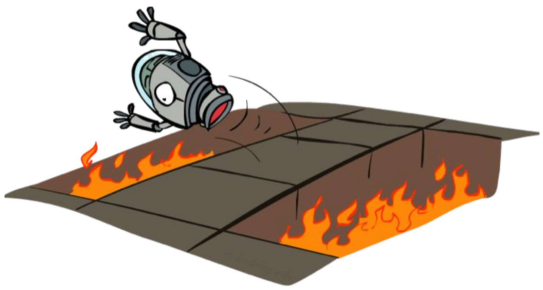
- Both value iteration and policy iteration compute the same thing (optimal state values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly define it
- In policy iteration:
  - We do several passes that update utilities with fixed policies (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)

Issue 4: requires knowing the model and the reward function

- We will explore online learning (reinforcement learning) approaches
- How can we learn the model and reward function from interactions?
- Do we even need to learn them? Can we learn V^*, Q^* without a model?
- Can we do without V^*, Q^* ? Can we run policy iteration without a model?



Offline optimization



Online Learning

Issue 5: requires discrete (finite) set of actions

- We will explore policy gradient approaches that are suitable for continuous actions, e.g., throttle and steering for a vehicle
- Can such approaches be relevant for discrete action spaces?
  - Yes! We can always define a continues action space as a distribution over the discreate actions (e.g., using the softmax function)
- Can we combine value-based approaches and policy gradient approaches and get the best of both?
  - Yes! Actor-critic methods

Issue 6: infeasible in large (or continues) state spaces

- Most real-life problems contain very large state spaces (practically infinite)
- It is infeasible to learn and store a value for every state
- Moreover, doing so is not useful as the chance of encountering a state more than once is very small
- We will learn to generalize our learning to apply to unseen states
- We will use value function approximators that can generalize the acquired knowledge and provide a value to any state (even if it was not previously seen)

Notation

- π^* - a policy that that yields the maximal expected sum of rewards
- $V^*(s)$ - the expected sum of rewards from being at s then following π^*
- $V_\pi(s)$ - the expected sum of rewards from being at s then following π
- $Q^*(s,a)$ - the expected sum of rewards from being at s, taking action a and then following π^*
- $Q_\pi(s,a)$ - the expected sum of rewards from being at s, taking action a and then following π
- G_t - observed sum of rewards following time t, i.e., $\sum_{k=t}^{T} r_k$

Required Readings

1. Chapter-3,4 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

Deep Reinforcement Learning
2022-23 Second Semester, M.Tech (AIML)

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Session #6-7:
Monte Carlo Methods

Instructors :
1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Prof. Sangeetha Viswanathan (sangeetha.viswanathan@pilani.bits-pilani.ac.in)

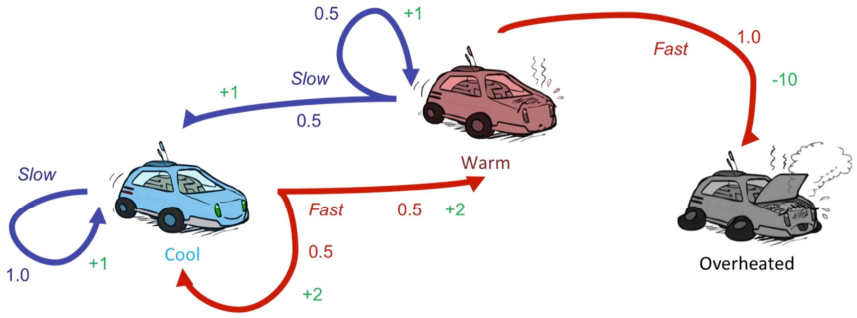
Agenda for the class

- Introduction
- On-Policy Monte Carlo Methods
- Off-Policy Monte Carlo Methods

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

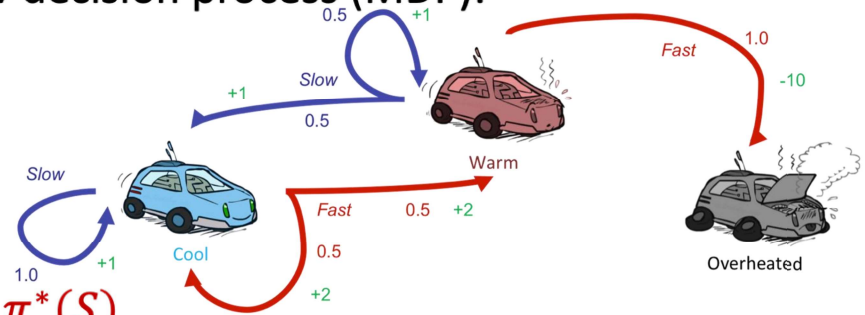
Introduction

- Recollect the problem
  - We need to learn a policy that takes us as far and as faster possible;



Introduction

- Still assume an underlying Markov decision process (MDP):
  - A set of states s ∈ S
  - A set of actions A
  - A model $P(s'|s,a)$
  - A reward function $R(s,a,s')$
  - A discount factor γ
  - Still looking for the best policy $\pi^*(S)$

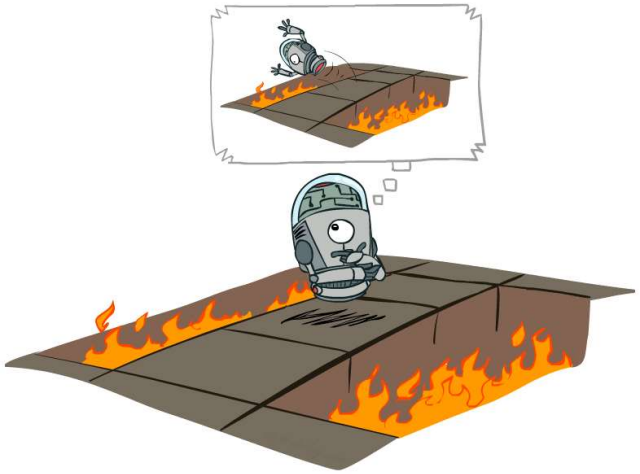


Introduction

- Still assume an underlying Markov decision process (MDP):
  - A set of states s ∈ S
  - A set of actions A
  - A model $P(s'|s,a)$
  - A reward function $R(s,a,s')$
  - A discount factor γ
  - Still looking for the best policy $\pi^*(S)$

- New twist: don't know the model and the reward function
  - That is, we don't know the actions' outcome
  - Must interact with the environment to learn

(Aside) Offline vs. Online (RL)



Offline Optimization

Online Learning

Monte Carlo Methods

- Monte Carlo methods are a broad class of computational algorithms that *rely on repeated random sampling to obtain numerical results*
- The underlying concept is to obtain unbiased samples from a complex/unknown distribution through a random process
- They are often used in physical and mathematical problems and are most useful when it is difficult or impossible to compute a solution analytically
  - Weather prediction
  - Computational biology
  - Computer graphics
  - Finance and business
  - Sport game prediction

First-visit Monte-Carlo Policy Evaluation [estimate $V\pi(s)$]

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

(a) Generate an episode using π

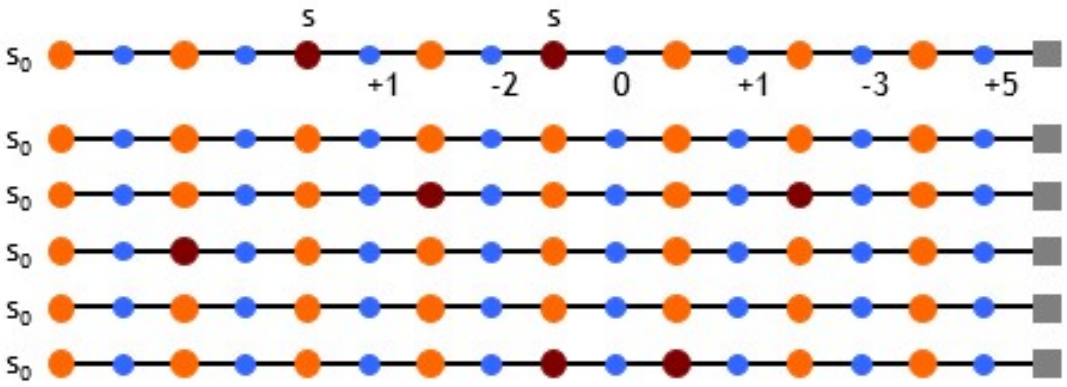
(b) For each state s appearing in the episode:

$R \leftarrow$ return following the first occurrence of s

Append R to $Returns(s)$

$V(s) \leftarrow \text{average}(Returns(s))$

Ex-1:First-visit Monte-Carlo Policy Evaluation [estimate Vπ(s)]

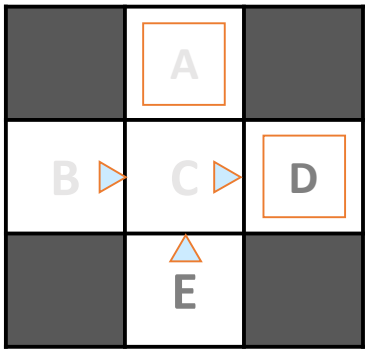


$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

Acknowledgements : This example is taken from the tutorial by Peter Bodík, RAD Lab, UC Berkeley

Ex-2: First-visit Monte-Carlo Policy Evaluation [estimate Vπ(s)]

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, , +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, , +10

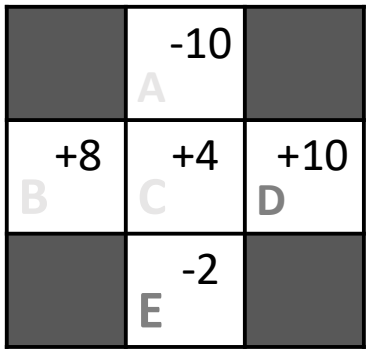
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, , +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, , -10

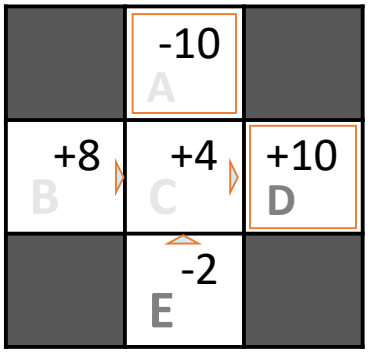
Output Values



Problems with MC Evaluation

- What's good about direct evaluation?
  - It's easy to understand
  - It doesn't require any knowledge of the underlying model
  - It converges to the true expected values
- What bad about it?
  - It wastes information about transition probabilities
  - Each state must be learned separately
  - So, it takes a long time to learn

Output Values



Think: If B and E both go to C with the same probability, how can their values be different?

Must explore!

- Hard policy (insufficient): $\pi(s) = a\,, \quad \pi: S \to \mathcal{A}$
- Soft policy: $\pi(a|s) = [0,1], \quad \pi: S \times \mathcal{A} \to p$
  - At the beginning $\forall a,\ \pi(a|s) > 0$ to allow exploration
  - Gradually shift towards a deterministic policy
- For instance: select a random action with probability ε
  - $\forall a \neq A^*, \pi(s,a) = \frac{\varepsilon}{|\mathcal{A}(s)|}$
  - Else select the greedy action: $\pi(s,A^*) = 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$

ε-greedy MC control

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

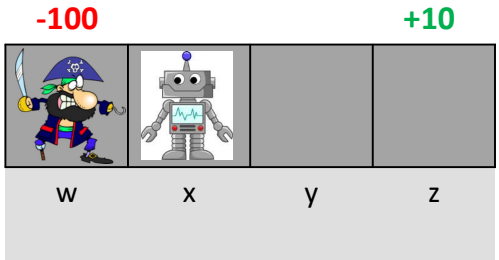
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

5	4,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-	-,-	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

On-policy first-visit MC control (for ε-soft policies),

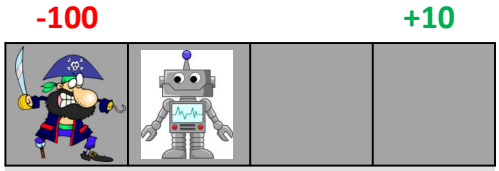
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

5	4,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-	-,-	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \leftarrow, 0, w, exit, -100$

On-policy first-visit MC control (for ε-soft policies),

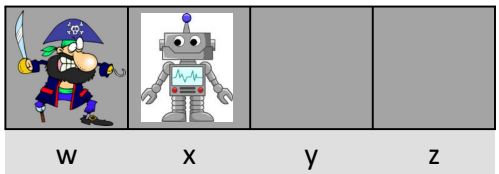
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s, a appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  (c) For each s in the episode:
    $A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
    For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

5	4,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,0	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \leftarrow, 0, w, exit, -100$

On-policy first-visit MC control (for ε-soft policies),

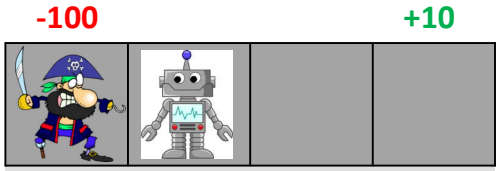
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s, a appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  (c) For each s in the episode:
    $A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
    For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,0	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \leftarrow, 0, w, exit, -100$

On-policy first-visit MC control (for ε-soft policies),

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

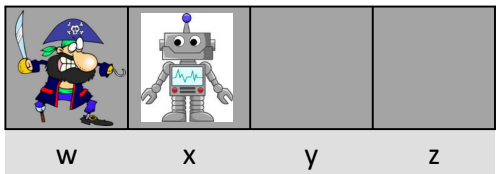
Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

17

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,0	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \leftarrow, 0, w, exit, -100$

- $A^* = [\rightarrow, exit]$

On-policy first-visit MC control (for ε-soft policies),

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

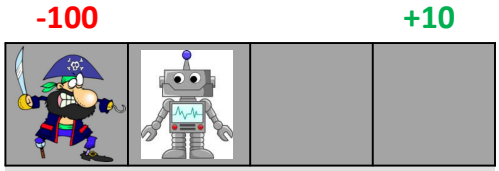
Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

18

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,0	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \leftarrow, 0, w, exit, -100$
- $A^* = [\rightarrow, exit]$

On-policy first-visit MC control (for ε-soft policies),

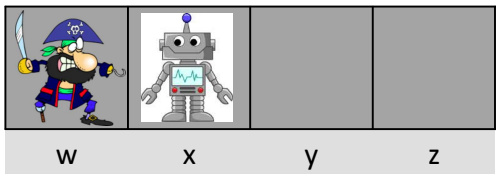
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,3	2,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,0	-,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \rightarrow, 0, y, \leftarrow, 0, x, \leftarrow, 0, exit, -100$

On-policy first-visit MC control (for ε-soft policies),

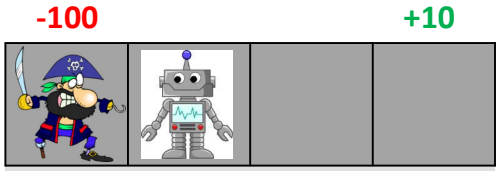
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,-72.9	-81,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,-72.9	-81,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \rightarrow, 0, y, \leftarrow, 0, x, \leftarrow, 0, exit, -100$

On-policy first-visit MC control (for ε-soft policies),

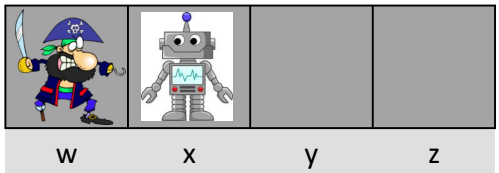
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,-72.9	-81,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,-72.9	-81,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \rightarrow, 0, y, \leftarrow, 0, x, \leftarrow, 0, exit, -100$

- $A^* = [\rightarrow, \rightarrow, exit]$

On-policy first-visit MC control (for ε-soft policies),

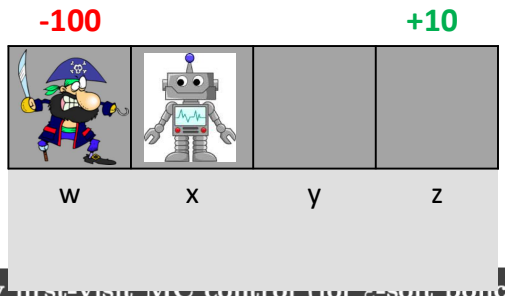
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC control - example

$\gamma = 0.9$



- $Q =$

-100	-90,-72.9	-81,1	0
w	x	y	z

- $\overline{Returns} =$

-100	-90,-72.9	-81,-	-
w	x	y	z

- $\pi(a|s) = (1-\varepsilon) \cdot$

exit			exit
w	x	y	z

  - $\varepsilon \cdot$ Random

- $\tau = x, \rightarrow, 0, y, \leftarrow, 0, x, \leftarrow, 0, exit, -100$
- $A^* = [\rightarrow, \rightarrow, exit]$

On-policy first-visit MC control (for ε-soft policies),

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ the return that follows the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$ (with ti
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

Quick Recap !
On-policy vs. Off-policy Learning

Required Readings

1. Chapter-3,4 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

Deep Reinforcement Learning

2022-23 Second Semester, M.Tech (AIML)

Session #9:
Temporal Difference Learning

S.P.Vimal | Dept. of CSIS, WILP, BITS-Pilani | vimalsp@wilp.bits-pilani.ac.in

Agenda for the class

- Temporal Difference Learning
  - TD(0)
  - SARSA
  - Q-Learning

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Solving MDPs so far

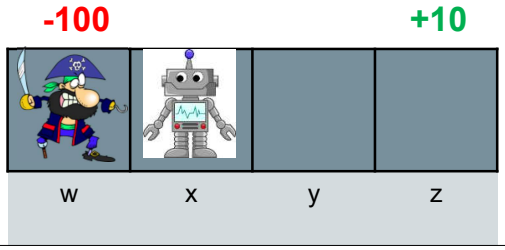
Dynamic programming
- ☑ Off policy
- ☑ local learning, propagating values from neighbors (Bootstrapping)
- ✗ Model based

Monte-Carlo
- ✗ On-policy (though important sampling can be used)
- ✗ Requires a full episode to train on
- ☑ Model free, online learning

- $Q(z, exit) = 10$
- $Q(y, \rightarrow) = 0 + \gamma \max_a Q(z, a)$
- $Q(x, \rightarrow) = 0 + \gamma \max_a Q(y, a)$

$$q^*(s,a) = \sum_{s'} p(s'|s,a)\,(r(s,a,s') + \gamma \max_a [q^*(s',a)])$$

$\gamma = 0.9$



- Episode = $\{x, y, z, exit\}$
- $Q(z, exit) = 10$
- $Q(y, \rightarrow) = 9$
- $Q(x, \rightarrow) = 8.1$

Fuse DP and MC

Dynamic programming
- ☑ Off policy
- ☑ local learning, propagating values from neighbors (Bootstrapping)
- ✗ Model based

Monte-Carlo
- ✗ On-policy (though important sampling can be used)
- ✗ Requires a full episode to train on
- ☑ Model free, online learning

TD Learning
- ☑ Off policy
- ☑ local learning, propagating values from neighbors (Bootsraping)
- ☑ Model free, online learning

Temporal difference learning

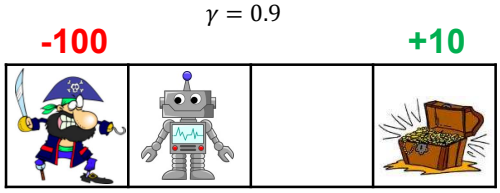
-
- $Q(s,a) = Q(s,a) + \alpha\left(R_{t+1} + \gamma \max_{a'}[Q(s',a')] - Q(s,a)\right)$
  - $V(s) = V(s) + \alpha\big(R_{t+1} + \gamma V(s') - V(s)\big)$
- Update estimate based on other estimates
- Model free
- Online, incremental learning
- Guaranteed to converge to the true value!
  - Some conditions on the step size, α (see slide #19 in 2Multi_armed_bandits.pptx)
- Usually converges faster than MC methods

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$

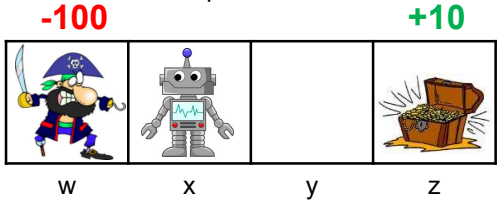


SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



$Q =$

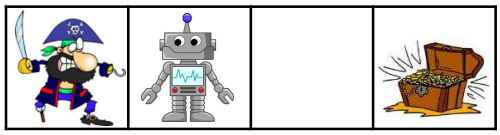
w	x	y	z
0	0,0	0,0	0

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	0	w	null
S	A	R	S'	A'

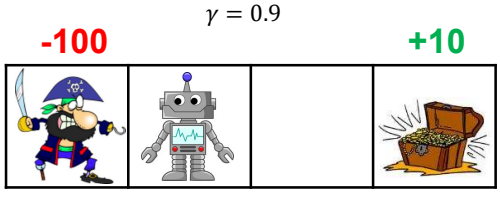
$$Q = \begin{array}{|c|c|c|c|}\hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

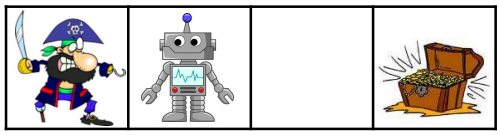
$$Q = \begin{array}{|c|c|c|c|}\hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

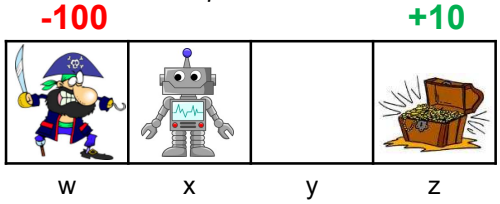
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline w & x & y & z \\ \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



w	exit	0	w	exi
S	A	R	S'	A'

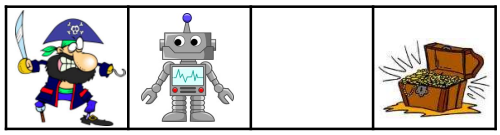
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline w & x & y & z \\ \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \cancel{\gamma Q(S',A')} - Q(S,A)\big]$ → 0
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



w	*e*	−100	*ter*	*exit*
S	A	R	S'	A'

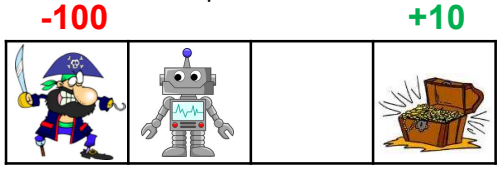
$$Q = \begin{array}{|c|c|c|c|}\hline \textbf{0} & \textbf{0,0} & \textbf{0,0} & \textbf{0} \\ \hline \textbf{w} & \textbf{x} & \textbf{y} & \textbf{z} \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



w	*exit*	−100	*ter*	·
S	A	R	S'	A'

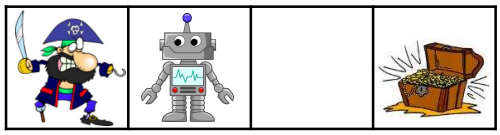
$$Q = \begin{array}{|c|c|c|c|}\hline \textbf{-100} & \textbf{0,0} & \textbf{0,0} & \textbf{0} \\ \hline \textbf{w} & \textbf{x} & \textbf{y} & \textbf{z} \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	−100	ter	·
S	A	R	S'	A'

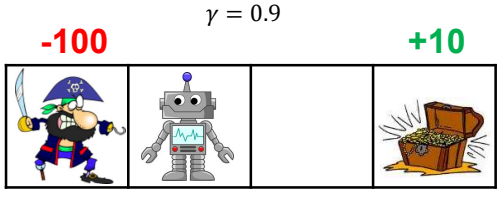
$$Q = \begin{array}{|c|c|c|c|} \hline \textbf{-100} & \textbf{0,0} & \textbf{0,0} & \textbf{0} \\ \hline \textbf{w} & \textbf{x} & \textbf{y} & \textbf{z} \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

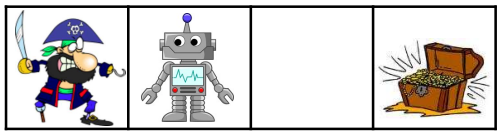
$$Q = \begin{array}{|c|c|c|c|} \hline \textbf{-100} & \textbf{0,0} & \textbf{0,0} & \textbf{0} \\ \hline \textbf{w} & \textbf{x} & \textbf{y} & \textbf{z} \\ \hline \end{array}$$

SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

$$Q = \begin{array}{|c|c|c|c|} \hline \text{-100} & \text{-90,0} & \text{0,0} & \text{0} \\ \hline \end{array}$$
w x y z

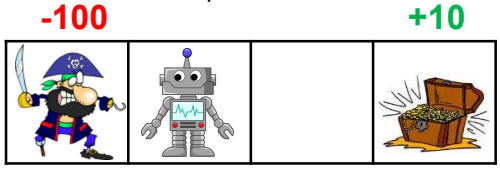
SARSA: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

And so on…

$\gamma = 0.9$



w	exit	0	w	exit
S	A	R	S'	A'

$$Q = \begin{array}{|c|c|c|c|} \hline \text{-100} & \text{-90,0} & \text{0,0} & \text{0} \\ \hline \end{array}$$
w x y z

Q-learning: Off-policy TD Control

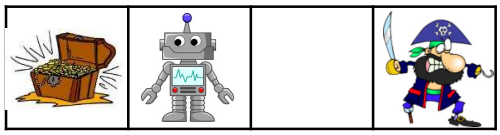
- Use the original TD update rule
- $Q(s,a) = Q(s,a) + \alpha\left(R_{t+1} + \gamma \max_{a'}[Q(s',a')] - Q(s,a)\right)$
- Approximates the state-action value for the optimal policy, i.e., q^*
  - Assuming that every state-action pair is visited infinitely often
- Follows from the proof of convergence for the Bellman function
  - See slides MDPs+DP

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	null	null	null
S	A	R	S'

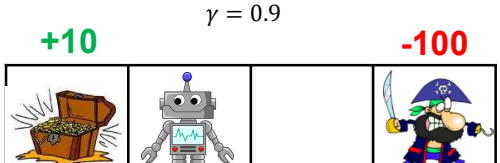
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

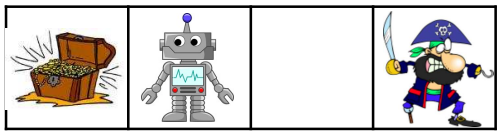
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

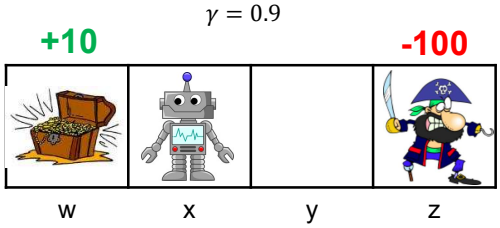
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



y	→	0	y
S	A	R	S'

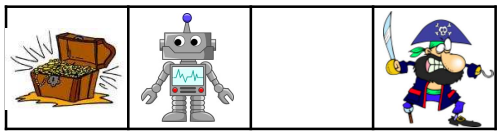
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

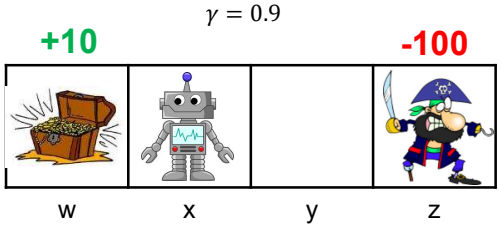
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



z	→	0	z
S	A	R	S'

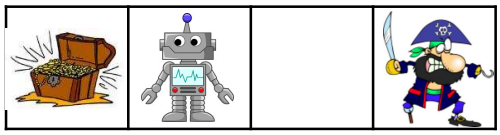
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



z	exit	−100	·
S	A	R	S'

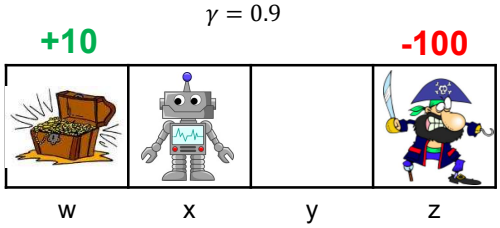
$$Q = \begin{array}{|c|c|c|c|}\hline 0 & 0{,}0 & 0{,}0 & 0 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \hline \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



z	exit	−100	·
S	A	R	S'

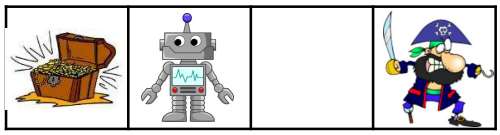
$$Q = \begin{array}{|c|c|c|c|}\hline 0 & 0{,}0 & 0{,}0 & -100 \\ \hline \text{w} & \text{x} & \text{y} & \text{z} \\ \hline \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



S	A	R	S'
x	e	-100	$\cdot$

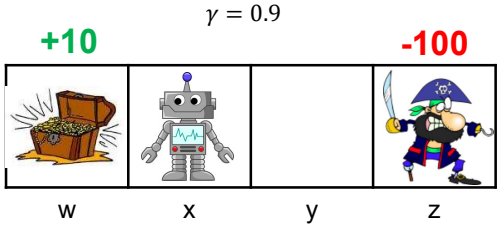
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & -100 \\ \hline w & x & y & z \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



S	A	R	S'
x	$\rightarrow$	0	y

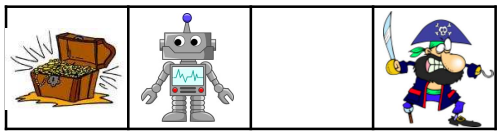
$$Q = \begin{array}{|c|c|c|c|} \hline 0 & 0{,}0 & 0{,}0 & -100 \\ \hline w & x & y & z \\ \end{array}$$

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

$Q =$

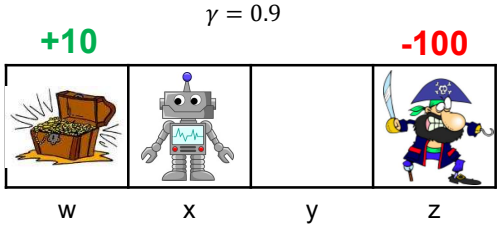
0	0,0	0,0	-100
w	x	y	z

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

$Q =$

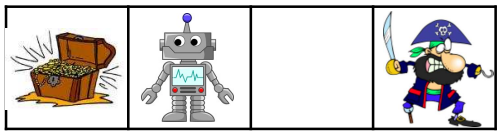
0	0,0	0,-90	-100
w	x	y	z

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	→	0	z
S	A	R	S'

$Q =$

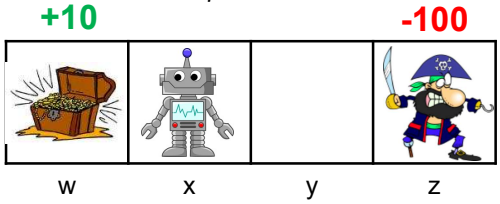
0	0,0	0,-90	-100
w	x	y	z

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

$Q =$

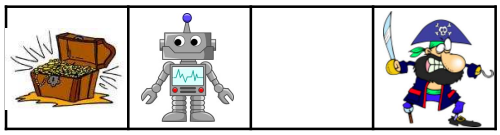
0	0,0	0,-90	-100
w	x	y	z

Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

$Q =$

0	0,0	0,-90	-100
w	x	y	z

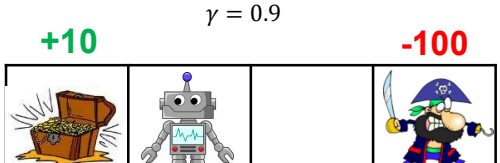
Q-learning: Off-policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

And so on…

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

$Q =$

0	0,0	0,-90	-100
w	x	y	z

Required Readings

1. Chapter-6 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Deep Reinforcement Learning

2022-23 Second Semester, M.Tech (AIML)

Session #10-11-12:
On Policy Prediction with Approximation

Instructors :

1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Prof. Sangeetha Viswanathan (sangeetha.viswanathan@pilani.bits-pilani.ac.in)

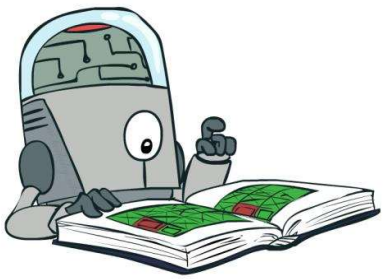
Agenda for the classes

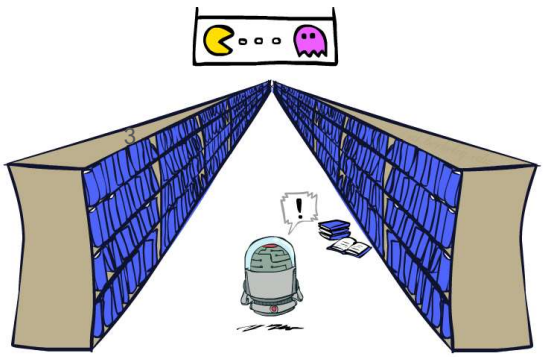
- ➔ Introduction
- ➔ Value Function Approximation
- ➔ Stochastic Gradient, Semi-Gradient Methods
- ➔ Role of Deep Learning for Function Approximation;
- ➔ Feature Construction Methods

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Generalizing Across States

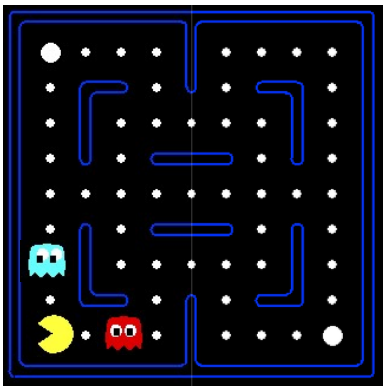
- Tabular Learning keeps a table of all state values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all during training
  - Too many states to hold a value table in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning



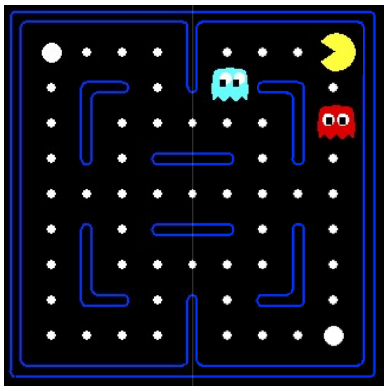


Example: Pacman

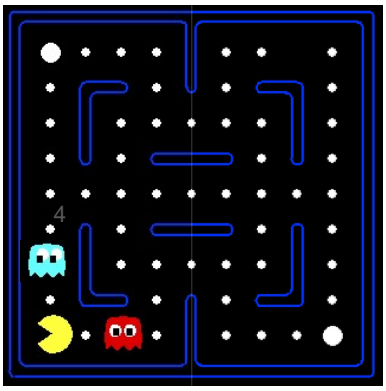
Let's say we discover through experience that this state is bad:



In naïve tabular-learning, we know nothing about this state:

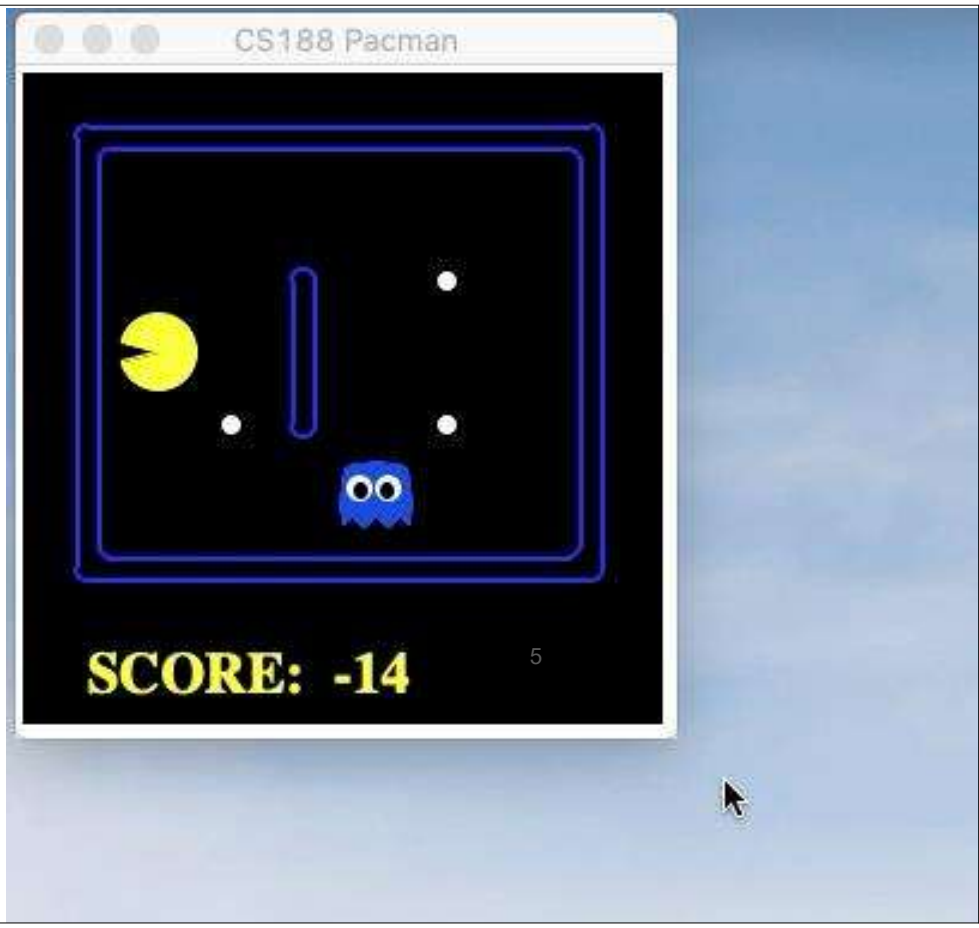


Or even this one!

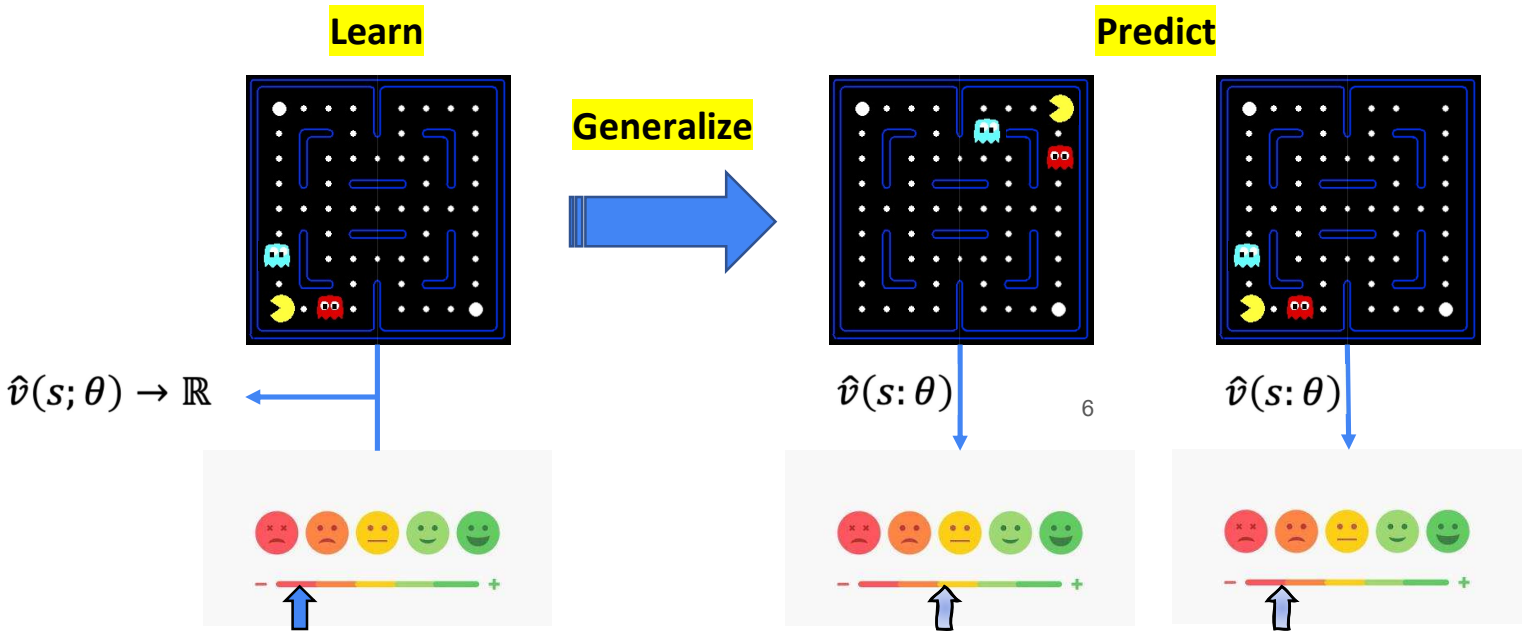


Demo

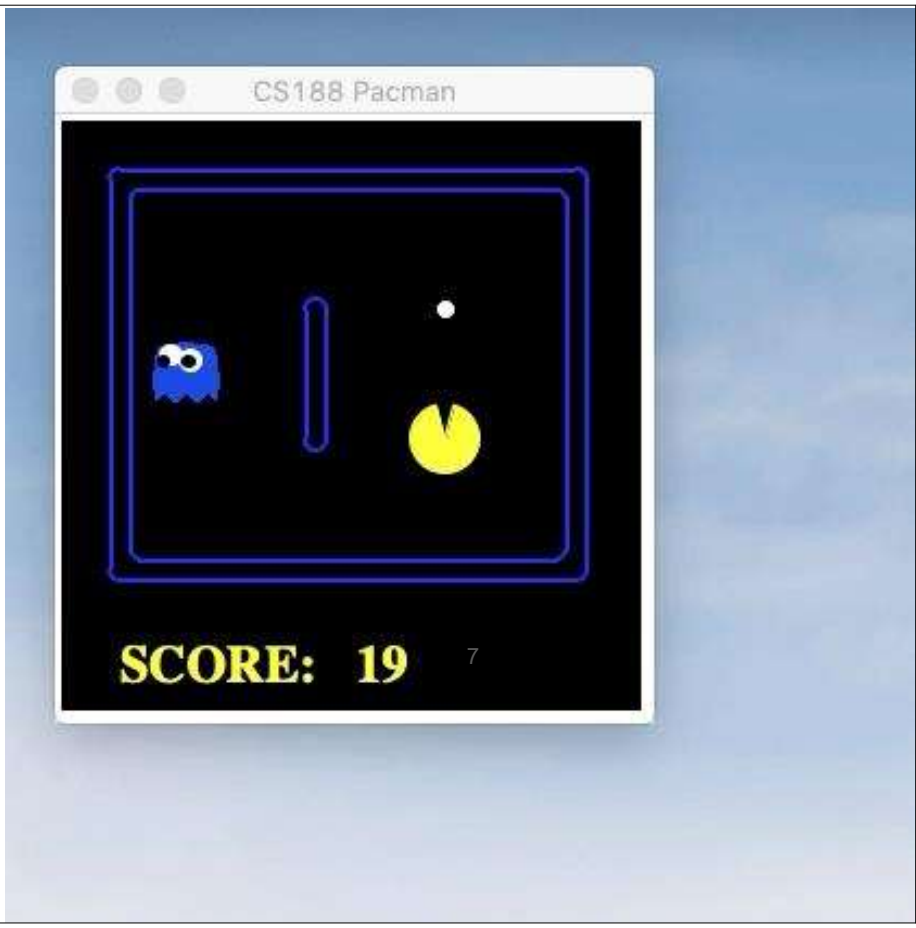
- Naïve Q-learning
- After 50 training episodes



Learn an approximation function

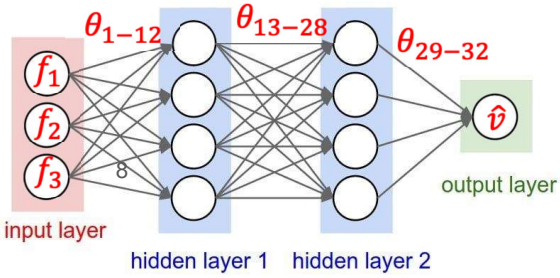


- Q-learning with function approximator



Parameterized function approximator

- Assume that each state is vector of features $(f_1, f_2, \ldots, f_n)$, e.g.,
  - Packman location, Ghost1 location , Ghost2 location, food location
  - Or even screen pixels
- A parametrized value approximator $\hat{v}(s;\theta)$ might look like this:
  - $= \sum_i \theta_i f_i$ or maybe like this $= \prod_i f_i^{\theta_i}$ or even this
- Assume we know the true value for a set of states:
  - $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
  - How can we update θ to reflect this information?

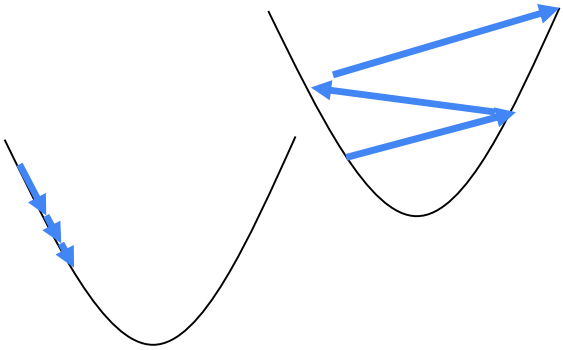


Gradient Descent

- Given: $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
- We want to set θ such that $\forall s, \hat{v}(s;\theta) = v(s)$
  - Not possible in the general case, why?
  - Instead we'll try to minimize the errors: loss $= \sum_s |v(s) - \hat{v}(s;\theta)|$
  - Partial derivative of the loss with respect to θ_i = how to change θ_i such that loss will increase the most
  - Go the other way -> decrease loss
  - Ooops! Absolute value is not differentiable -> can't compute gradients
  - Simple fix: loss $= \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$ = squared loss function

Gradient Decent

- loss $= \frac{1}{2}\sum_s [v(s) - \hat{v}(s;\theta)]^2$
- For each i
  - Push θ_i towards a direction that minimizes loss
  - $\theta_i = \theta_i - \frac{\partial loss}{\partial \theta_i}$
- More generally $\theta = \theta - \alpha \nabla loss$
  - $\nabla loss = \left(\frac{\partial loss}{\partial \theta_1}, \frac{\partial loss}{\partial \theta_2}, \ldots, \frac{\partial loss}{\partial \theta_n}\right)$
  - α is the learning rate, requires tuning per domain, too large learning diverges to small results in slow learning or even premature convergence



Stochastic Gradient Descent

SGD for Monte Carlo estimation

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$

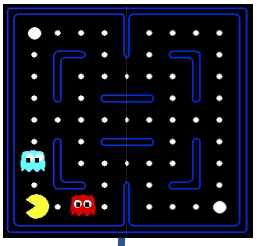
Initialize value-function weights $\mathbf{w}$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  For $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

w are the tunable parameters of the value approximation function

- Guaranteed to converge to a local optimum because G_t is an unbiased estimate of $v_\pi(S_t)$

Example

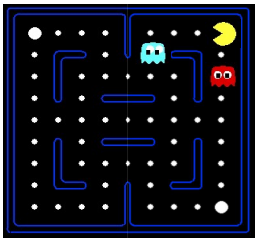
$f(S) = [2,2,1]$



10

- $S = \{f_1(S), f_2(S), f_3(S)\}$
  - $f_{1,2}$=distance to ghost 1,2, f_3=distance to food
- $\hat{v}(s) = \sum_i \theta_i f_i(s)$
  - init: $\theta = [0,0,0]$
- $\theta = \theta + \alpha\big(G_t - \hat{v}(s;\theta)\big)\nabla\hat{v}(s;\theta)$
- $\theta = [0,0,0] + 0.1(10 - [0,0,0] \cdot [2,2,1])[2,2,1]$
  - $\theta = [2,2,1]$
- $\hat{v}(S') = f(S') \cdot \theta = [2,4,1] \cdot [2,2,1] = 13$

$f(S') = [2,4,1]$



Learning approximation with bootstrapping

- Can we update the value approximation function at every step?
- Yes, define SGD as a function of the TD error
  - Tabular TD learning: $\hat{v}(s_t) = \hat{v}(s_t) + \alpha\big(r_t + \gamma\hat{v}(s_{t+1}) - \hat{v}(s_t)\big)$
  - Approximation TD learning: $\theta = \theta + \alpha\big(r_t + \gamma\hat{v}(s_{t+1};\theta) - \hat{v}(s_t;\theta)\big)\nabla\hat{v}(s_t;\theta)$
- Known as Semi-gradient methods
- **NOT** guaranteed to converge to a local optimum because $\hat{v}(s_{t+1};\theta)$ is a biased estimate of $v_\pi(s_{t+1})$
- Semi-gradient (bootstrapping) methods do not converge as robustly as (full) gradient methods

Semi-gradient methods

- They do converge reliably in important cases such as the linear approximation case
- They offer important advantages that make them often clearly preferred
- They typically enable significantly faster learning, as we have seen in Chapters 6 and 7
- They enable learning to be continual and online, without waiting for the end of an episode
- This enables them to be used on continuing problems and provides computational advantages

Semi-gradient TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

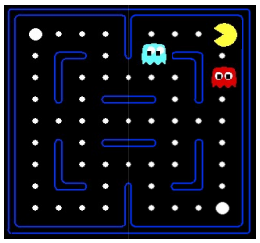
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S' is terminal

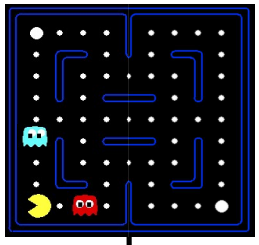
What's the difference from the tabular case?

Example

$f(U) = [2,4,1]$

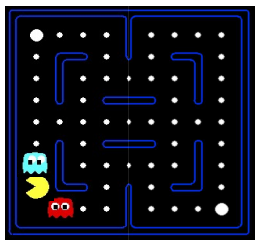


$f(S) = [2,3,1]$



$R = +10$

$f(S') = [1,2,1]$



- $S = \{f_1(S), f_2(S), f_3(S)\}$
  - $f_{1,2}$=distance to ghost 1,2, f_3=distance to food
- $\hat{v}(s) = \sum_i \theta_i f_i(s)$
  - init: $\theta = [0,0,0]$
- $\theta = \theta + \alpha\big(R + \gamma\hat{v}(S';\theta) - \hat{v}(S;\theta)\big)\nabla\hat{v}(S;\theta)$
- $\theta = [0,0,0] + 0.1(10 + [1,2,1] \cdot [0,0,0] - [2,3,1] \cdot [0,0,0])[2,3,1]$
  - $\theta = [2,3,1]$
- $\hat{v}(U) = f(U) \cdot \theta = [2,4,1] \cdot [2,3,1] = 17$

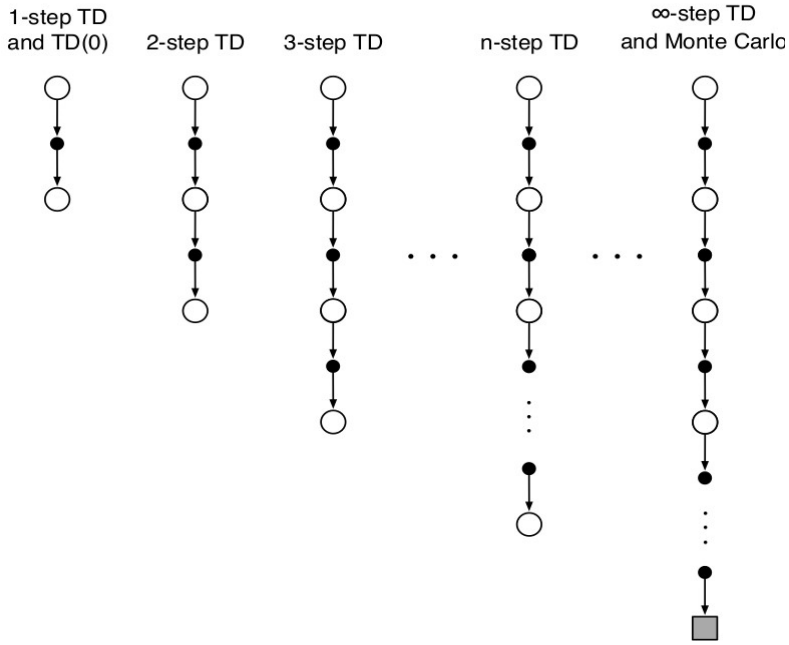
[Review] n-step TD Prediction

One-step return:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

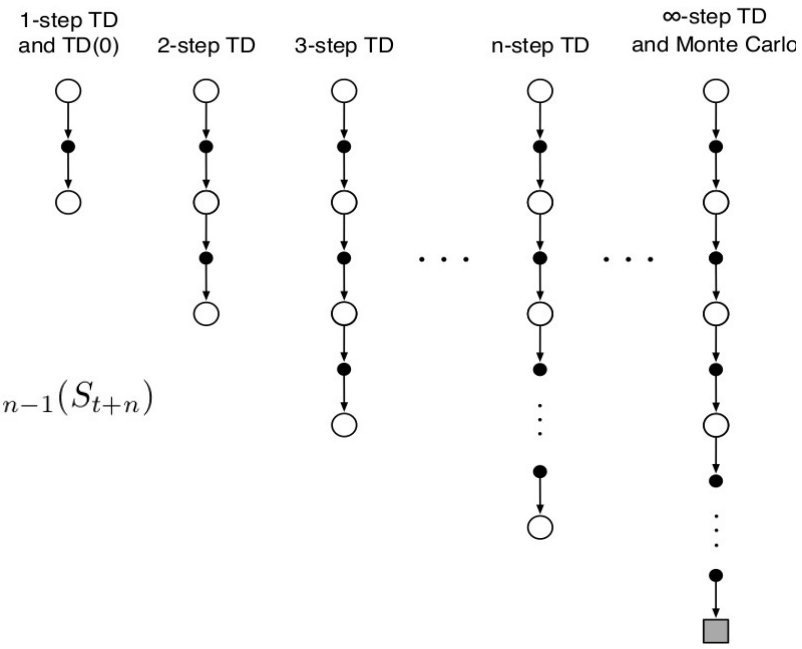
Two-step return:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$



Ref Section 7.1 of TB

[Review] n-step TD Prediction



One-step return:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

Two-step return:

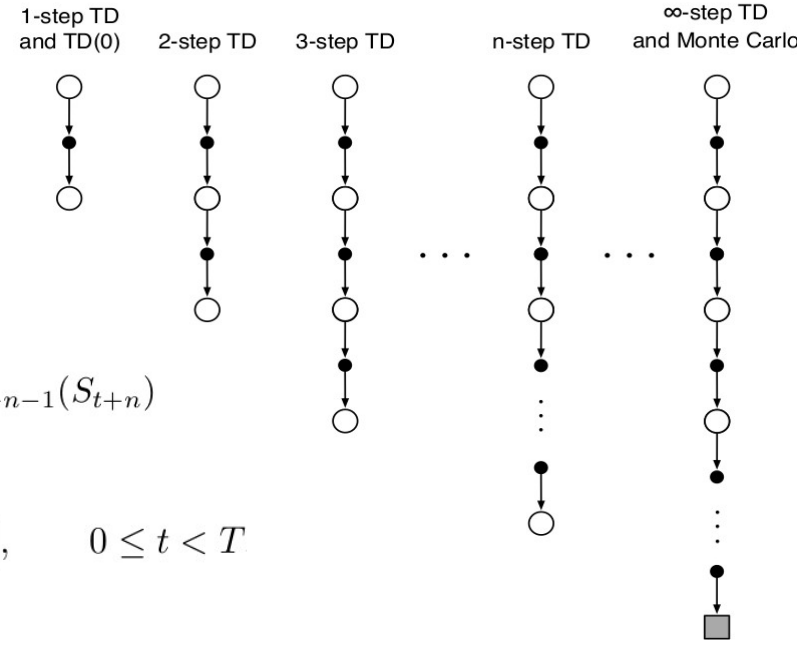
$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

Ref Section 7.1 of TB

[Review] n-step TD Prediction



One-step return:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

Two-step return:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

State-value learning algorithm for using n-step returns:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\big[G_{t:t+n} - V_{t+n-1}(S_t)\big], \qquad 0 \le t < T$$

Ref Section 7.1 of TB

[Review] n-step TD Prediction

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

n-step return

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (S_t and R_t) can take their index mod n

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau, \mathbf{w})]\,\nabla \hat{v}(S_\tau, \mathbf{w})$
  Until $\tau = T - 1$

- Again, only a simple modification over the tabular setting

n-step return

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

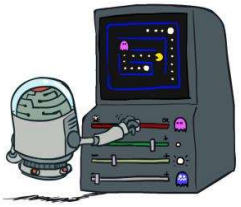
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (S_t and R_t) can take their index mod n

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take an action according to $\pi(\cdot|S_t)$
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then $T \leftarrow t + 1$
    $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

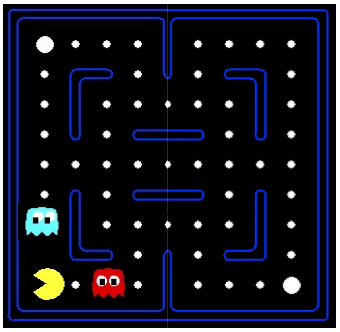
29

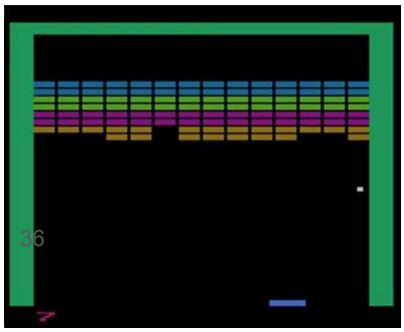
- Again, only a simple modification over the tabular setting
- Weight update instead of tabular entry update

Feature selection



- Assume a linear function approximator $\hat{v}(f(s);\theta) = f(s) \cdot \theta$
- What relevant features should represent states?



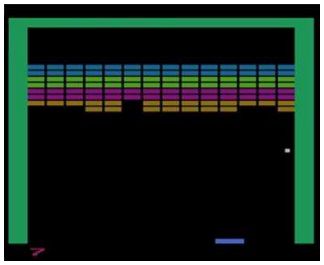


36

Features are domain depended
requiring expert knowledge

Automatic features extraction

- Consider a game state that is given as a bit map
- Raw data of type $pixel(7,3) = [0,0,0]$ (black)
- Desired features = {ball location, ball speed, ball direction, pan location…}
- How can we translate pixels to the relevant features?

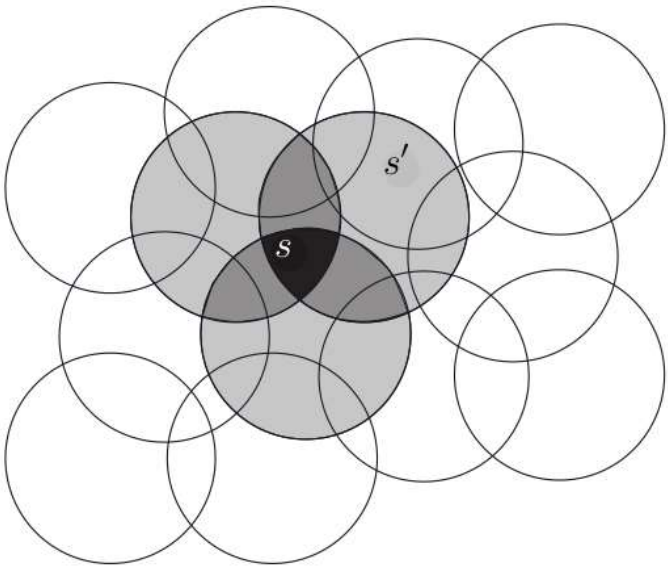


Automatic features extraction for linear approximator

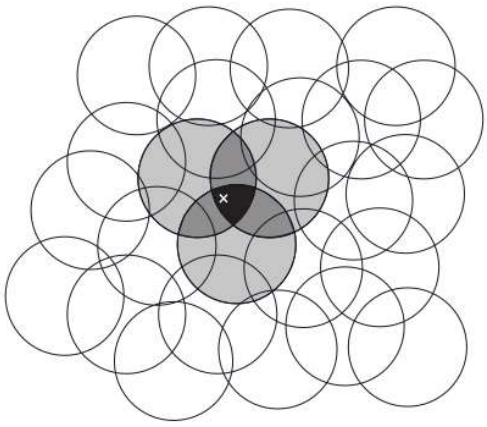
- **Polynomials**: $f_i(s) = \prod_{j=1}^{k} x_j^{c_{i,j}}$
  - where each $c_{i,j}$ is an integer in the set $\{0,1, \dots, n\}$ for an integer $n \geq 0$
  - These features makeup the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features
- **Fourier Basis**: $f_i(s) = \cos(\pi X^\top c^i)$
  - Where $c^i = (c_1^i, \dots, c_k^i)^\top$, with $c_j^i \in \{0, \dots, n\}$ for $j = \{1, \dots, k\}$ and $i = \{0, \dots, (n+1)^k\}$
  - This defines a feature for each of the $(n+1)^k$ possible integer vectors c^i
  - The inner product $X^\top c^i$ has the effect of assigning an integer in $\{0, \dots, n\}$ to each dimension of X
  - This integer determines the feature's frequency along that dimension
  - The features can be shifted and scaled to suit the bounded state space of a particular application

Automatic features extraction for linear approximator - Coarse Coding

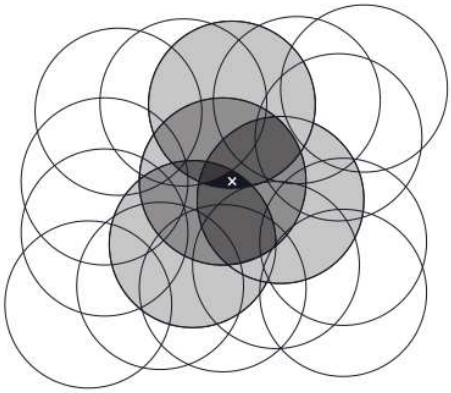
- Natural representation of the state set is continuous
- In 2-d, features corresponding to circles in state space
- Coding of a state:
  - If the state is inside a circle, then the corresponding feature has the value 1
  - otherwise the feature is 0
- Corresponding to each circle is a single weight (a component of w) that is learned
  - Training a state affects the weights of all the intersecting circles.



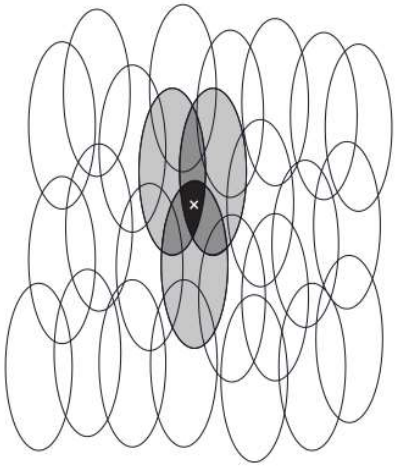
Automatic features extraction for linear approximator - Coarse Coding



Narrow generalization

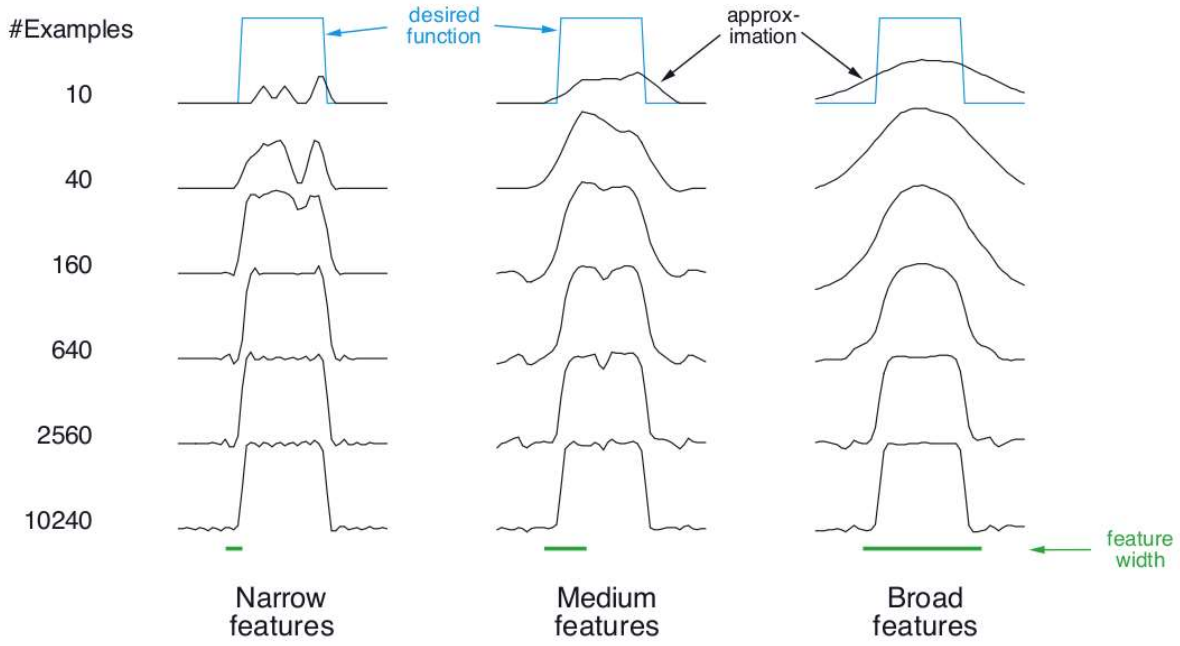


Broad generalization

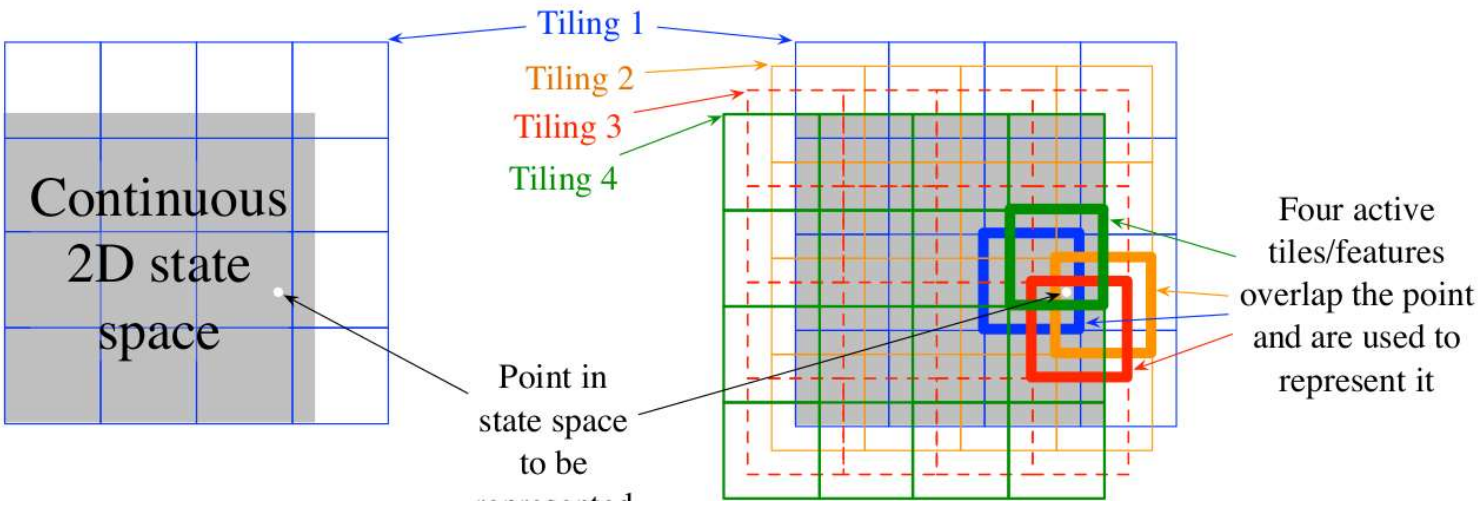


Asymmetric generalization

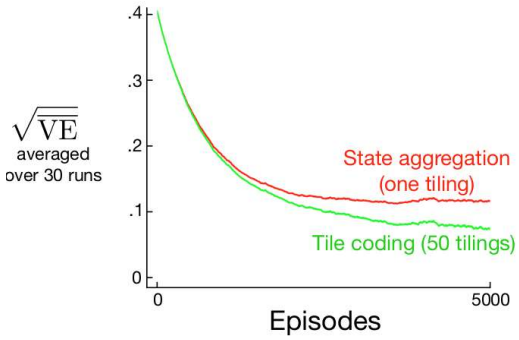
Automatic features extraction for linear approximator - Coarse Coding

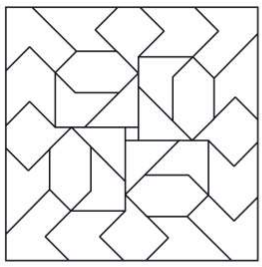


Automatic features extraction for linear approximator - Tile Coding

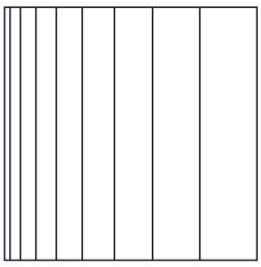


Automatic features extraction for linear approximator - Tile Coding

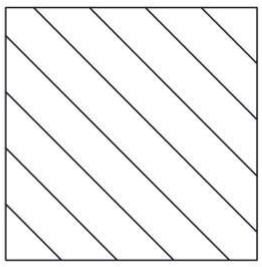




Irregular

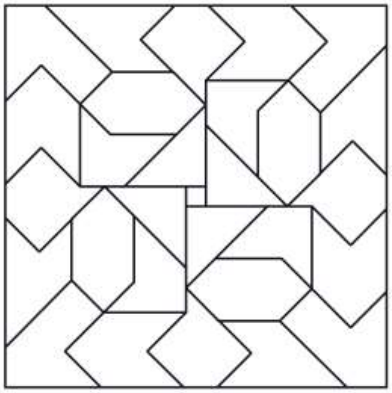


Log stripes

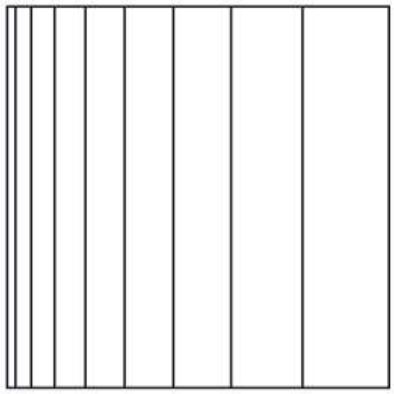


Diagonal stripes

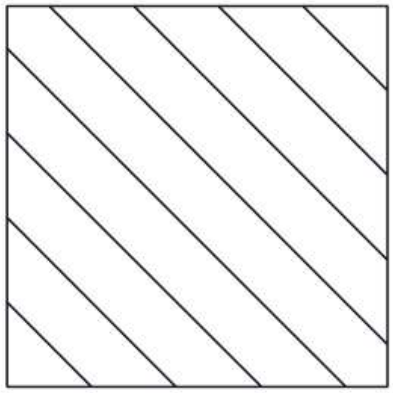
Automatic features extraction for linear approximator - Tile Coding



Irregular



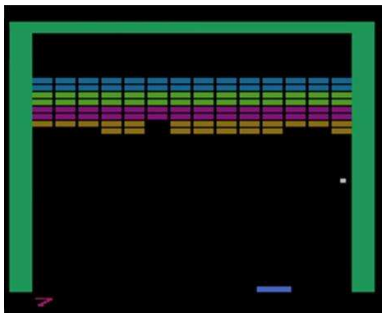
Log stripes



Diagonal stripes

Automatic features extraction for linear approximator

- Other approaches include: Coarse Coding, Tile Coding, Radial Basis Functions (See chapter 9.5 in textbook)
- Each of these approaches defines a set of features, some useful yet most are not
  - E.g., is there a polynomial/Fourier function that translates pixels to pan location?
  - Probably but it's a needle in a (combinatorial) haystack
- Can we do better (generically)
  - Yes, using deep neural networks…



What did we learn?

- Reinforcement learning must generalize on observed experience if it is to be applicable to real world domains
- We can use parameterized function approximation to represent our knowledge about the domain state/action values
- Use stochastic gradient descend to update the tunable parameters such that the observed (TD, rollout) error is reduced
- When using a linear approximator, the Least squares TD method provides the most sample efficient approximation

Part-2 DQN

Mnih et al. 2015

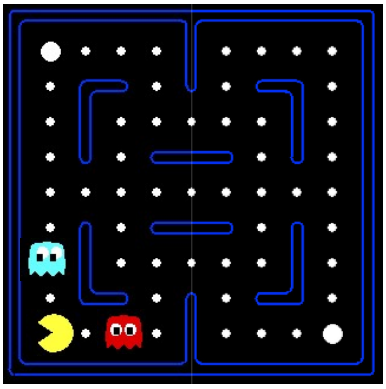
- First deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning
- The model is a convolutional neural network, trained with a variant of Q-learning
- Input is raw pixels and output is an action-value function estimating future rewards
- Surpassed a human expert on various Atari video games

The age of deep learning

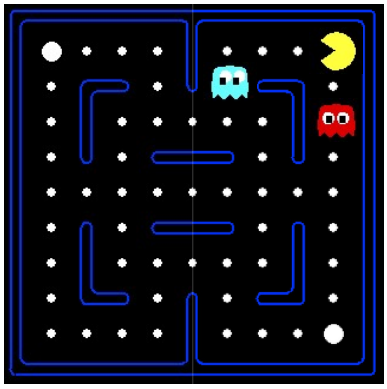
- Previous models relied on hand-crafted features combined with linear value functions
  - The performance of such systems heavily relies on the quality of the feature representation
- Advances in deep learning have made it possible to automatically extract high-level features from raw sensory data

Example: Pacman

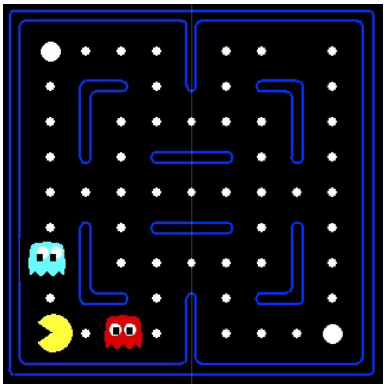
Let's say we discover through experience that this state is bad:



In naïve q-learning, we know nothing about this state:

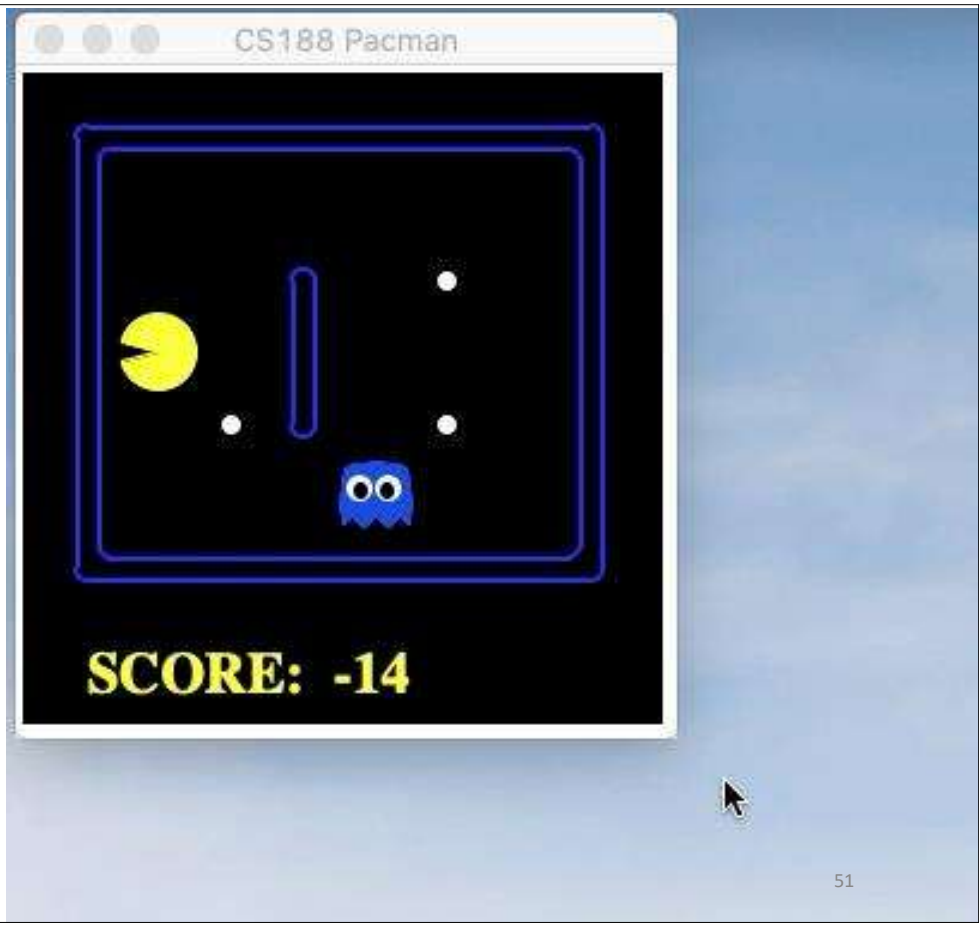


Or even this one!



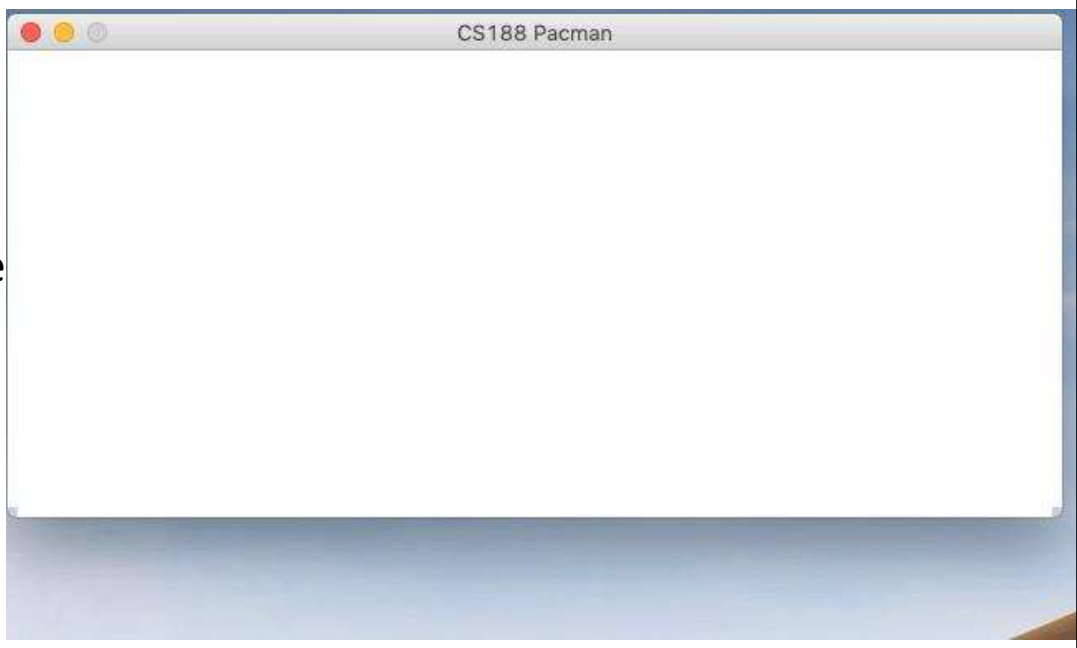
We must generalize our knowledge!

- Naïve Q-learning
- After 50 training episodes



- Generalize Q-learning with function approximator

- Generalizing knowledge results in efficient learning
- E.g., learn to avoid the ghosts



Generalizing with Deep learning

- **Supervised**: Require large amounts of hand-labelled training data
  - **RL** on the other hand, learns from a scalar reward signal that is frequently sparse, noisy, and delayed
- **Supervised**: Assume the data samples are independent
  - In **RL** one typically encounters sequences of highly correlated state
- **Supervised**: Assume a fixed underlying distribution
  - In **RL** the data distribution changes as the algorithm learns new behaviors
- DQN was first to demonstrate that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments

Deep Q learning [Mnih et al. 2015]

- Trains a generic neural network-based agent that successfully learns to operate in as many domains as possible
- The network is not provided with any domain-specific information or hand-designed features
- Must learn from nothing but the raw input (pixels), the reward, terminal signals, and the set of possible actions

Original Q-learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $\theta = \theta + \alpha \left(R + \gamma \max_a \hat{Q}(S', a; \theta) - \hat{Q}(S, A; \theta) \right) \nabla_\theta \hat{Q}(S, A; \theta)$
    $S \leftarrow S'$
  until S is terminal

Deep Q learning [Mnih et al. 2015]

- DQN addresses problems of correlated data and non-stationary distributions
  - Use an experience replay mechanism
  - Randomly samples and trains on previous transitions
  - Results in a smoother training distribution over many past behaviors

Deep Q learning [Mnih et al. 2015]

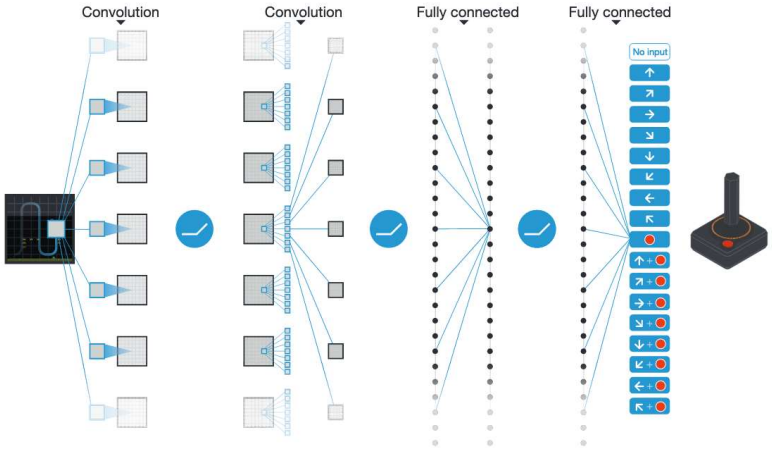
- Learn a function approximator (Q network), $\hat{Q}(s, a; \theta) \approx Q^*(s, a)$
- Value propagation: $Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a')\right]$
  - $\mathcal{E}$ represents the environment (underlying MDP)
- Update $\hat{Q}(s, a; \theta)$ at each step i using SGD with squared loss:
- $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[\left(y_i - \hat{Q}(s, a; \theta_i)\right)^2\right]$
  - $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1})\right]$
  - $\rho(s, a)$ is the behavior distribution, e.g., epsilon greedy
  - θ_{i-1} is considered fix when optimizing the loss function (helps when taking the derivative with respect to θ and with stability)

Deep Q learning [Mnih et al. 2015]

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

Independent of θ_i (because θ_{i-1} is considered fix)

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



Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

Initialize the replay memory and two identical Q approximators (DNN). $\hat{Q}$ is our target approximator.

For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a;\theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Play *m* episodes (full games)

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Start episode from x_1 (pixels at the starting screen).
Preprocess the state (include 4 last frames, RGB to grayscale conversion, downsampling, cropping)

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

For each time step during the episode

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

With small probability select a random action (explore), otherwise select the, currently known, best action (exploit).

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Execute the chosen action and store the (processed) observed transition in the replay memory

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Experience replay:
Sample a random minibatch of transitions from replay memory and perform gradient decent step on Q (not on $\hat{Q}$)

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Once every several steps set the target function, $\hat{Q}$, to equal Q

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Such delayed online learning helps in practice:
"This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy" [Human-level control through deep reinforcement learning. Nature 518.7540 (2015): 529.]

Deep Q learning

- *model-free*: solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly learning an estimate of $\mathcal{E}$
- *off-policy*: learns the optimal policy, $a = \underset{a}{arg\max}\, Q(s, a; \theta)$, while following a different behavior policy
  - One that ensures adequate exploration of the state space

Experience replay

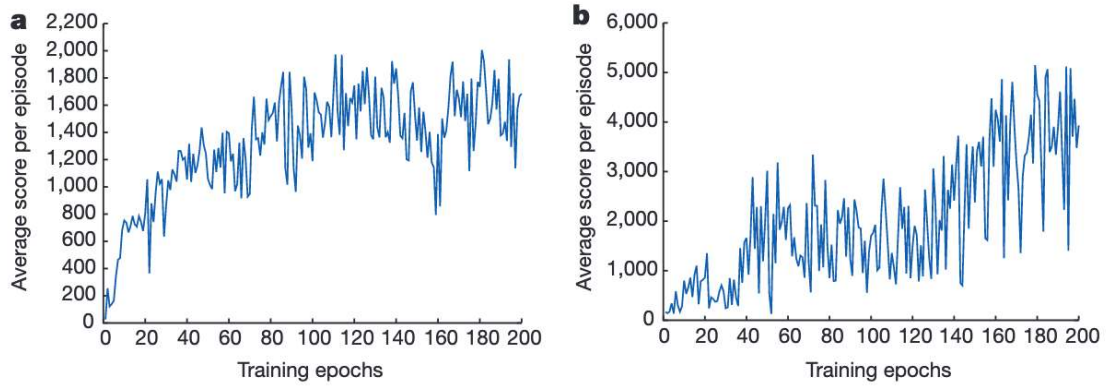
- Utilizing experience replay has several advantages
  - Each step of experience is potentially used in many weight updates, which allows for greater data efficiency
  - Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates
  - The behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters
- Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning

Experience replay

- DQN only stores the last N experience tuples in the replay memory
  - Old transitions are overwritten
- Samples uniformly at random from the buffer when performing updates
- Is there room for improvement?
  - Important transitions?
    - Prioritized sweeping
  - Prioritize deletions from the replay memory
  - see prioritized experience reply, https://arxiv.org/abs/1511.05952

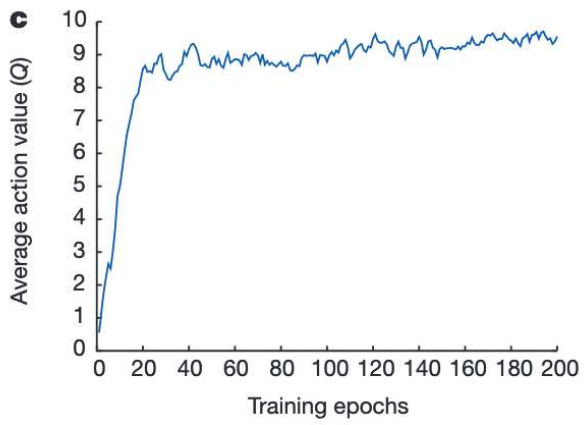
Results: DQN

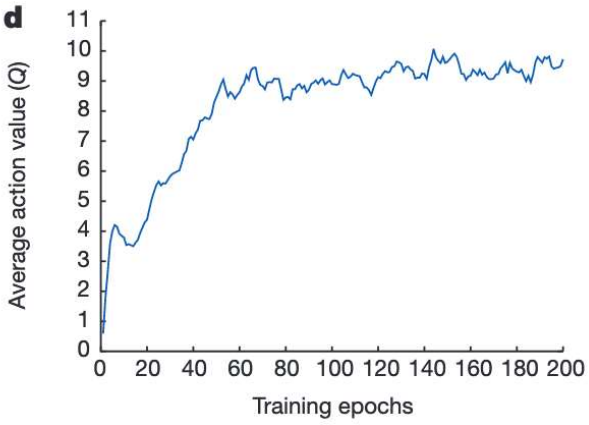
- Each point is the average score achieved per episode after the agent is run with an epsilon-greedy policy ($\varepsilon = 0.05$) for 520k frames on SpaceInvaders (a) and Seaquest (b)

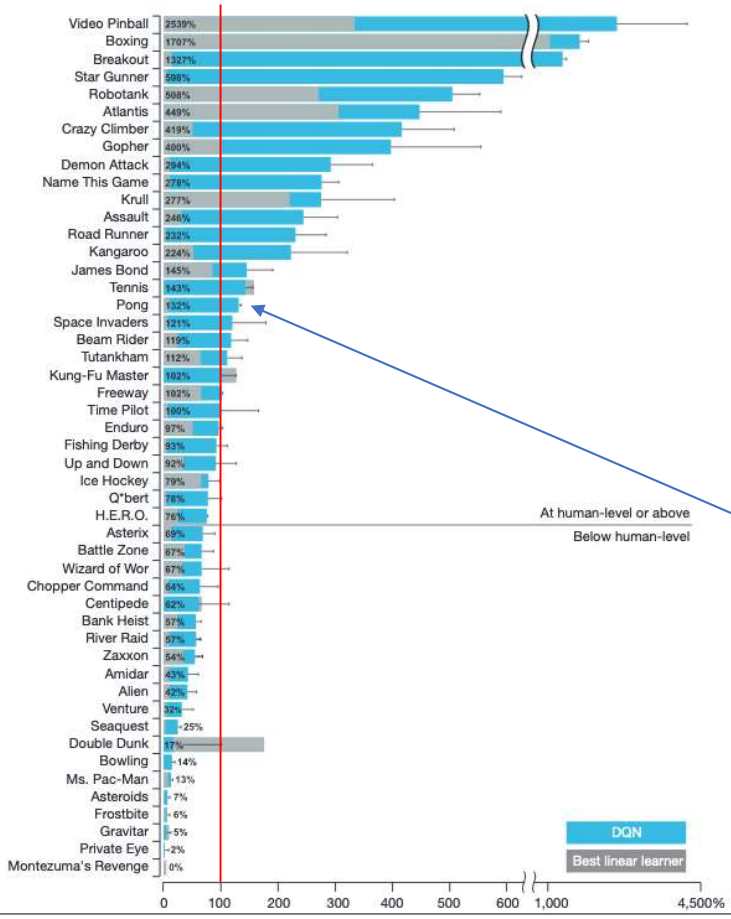


Results: DQN

- Average predicted action-value on a held-out set of states on Space Invaders (c) and Seaquest (d)







- Normalized between a professional human games tester (100%) and random play (0%)
- E.g., in Pong, DQN achieved a factor of 1.32 higher score on average when compared to a professional human player

Maximization bias

- See lecture 5TD_learning.pptx, slide 41
- The max operator in Q-learning uses the same values both to select and to evaluate an action
  - $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) \right]$
  - Makes it more likely to select overestimated values, resulting in overoptimistic value estimates
- We can use a technique similar to the previously discussed Double Q-learning (lecture 5TD_learning.pptx, slide 43)
  - Two value functions are trained. Update only one of the two value functions at each step while considering the $\max_a$ from the other

Double Deep Q networks (DDQN)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

We have two available Q networks. Let's use them for double learning

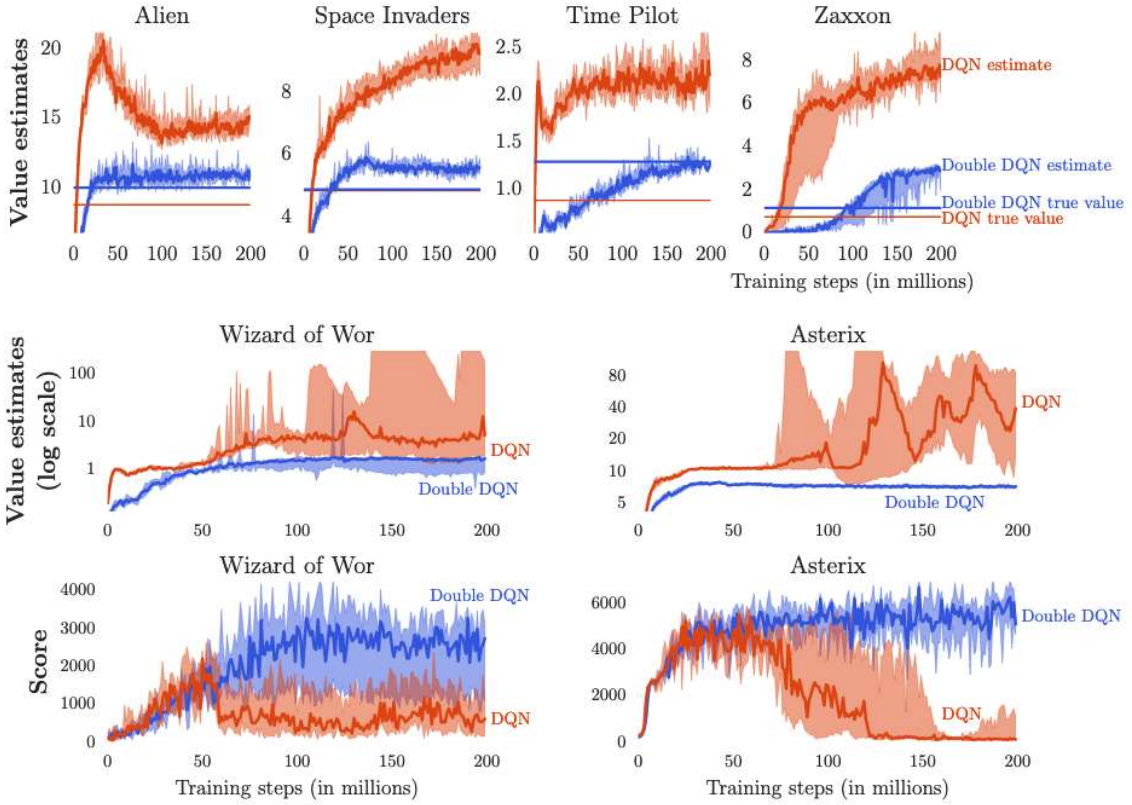
Double Deep Q networks (DDQN)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

We have two available Q networks. Let's use them for double learning

$$r_j + \gamma \hat{Q}\left(\phi_{j+1}, \arg\max_{a'} Q(\phi_{j+1}, a'; \theta), \theta^-\right)$$

- Hasselt et al. 2015
- The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias.



What did we learn?

- Using deep neural networks as function approximators in RL is tricky
  - Sparse samples
  - Correlated samples
  - Evolving policy (nonstationary sample distribution)
- DQN attempts to address these issues
  - Reuse previous transitions at each training (SGD) step
  - Randomly sample previous transitions to break correlation
  - Use off-policy, TD(0) learning to allow convergence to the true target values (Q*)
    - No guarantees for non-linear (DNN) approximators

Required Readings

1. Chapter-6 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Deep Reinforcement Learning
2022-23 Second Semester, M.Tech (AIML)

Session #13:
Policy Gradients - REINFORCE, Actor-Critic algorithms

Instructors :
1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Prof. Sangeetha Viswanathan (sangeetha.viswanathan@pilani.bits-pilani.ac.in)

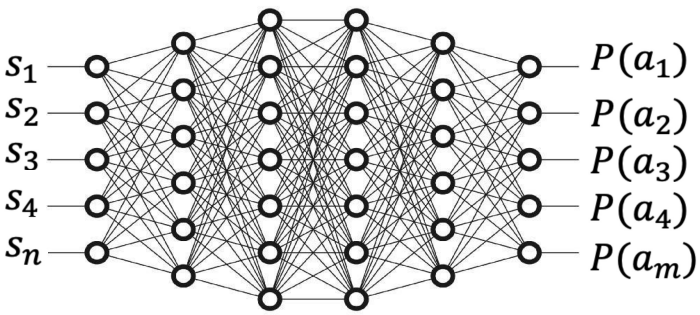
Agenda for the classes

- ➔ Introduction
- ➔ Policy gradients
- ➔ REINFORCE algorithm
- ➔ Actor-critic methods
- ➔ REINFORCE - example

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Notation

- The policy is a parametrized function: $\pi_\theta(a|s)$
  - For policy gradient we need a continues, differentiable policy… (Softmax activation can be useful for discreate action spaces)
  - $\pi(a|s)$ is assumed to be differentiable with respect to θ
  - E.g., a DNN were θ is the set of weights and biases
- $J(\theta)$ is a scalar policy performance measure (expected sum of discounted rewards) with respect to the policy params



Improving the policy

- SGD: $\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$
- Where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate, whose expectation approximates the gradient of the performance measure
- All methods that follow this general schema we call policy gradient methods
  - Might also learn an approximate value function for normalization (baseline)
- Methods that use approximations to both policy and value functions for computing the policy's gradient are called actor–critic methods (more on this later)

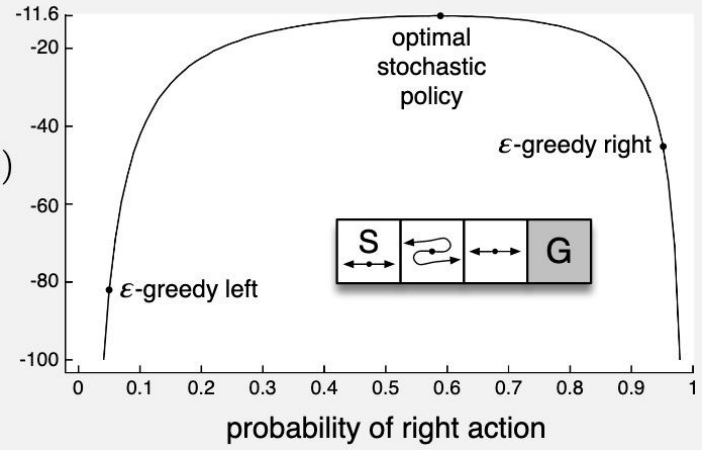
Advantages of PG

- The policy convergence over time as opposed to an epsilon-greedy value-based approach
- Naturally applies to continues action space as opposed to a Q learning approach
- In many domains the policy is a simpler function to approximate
  Though this is not always the case
- Choice of policy parameterization is sometimes a good way of injecting prior knowledge
  E.g., in phase assignment by a traffic controller
- Can converge on stochastic optimal policies, as opposed to value-based approaches
  Useful in games with imperfect information where the optimal play is often to do two different things with specific probabilities, e.g., bluffing in Poker

Stochastic policy

- Reward = -1 per step
- $\mathcal{A} = \{left,\ right\}$
- Left goes left and right goes right except in the middle state where they are reversed
- States are identical from the policy's perspective
- $\pi^* = [0.41 \quad 0.59]$

$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\mathrm{S})$



Evaluate the gradient in performance

- $\widehat{\nabla_\theta J(\theta)} = ?$
- J depends on both the action selections and the distribution of states in which those selections are made
  - Both of these are affected by the policy parameter θ
- Seems impossible to solve without knowing the transition function (or the distribution of visited states)
  - $p(s'|s,a)$ is unknown in model free RL
- The PG theorem allows us to evaluate $\widehat{\nabla_\theta J(\theta)}$ without the need for $p(s'|s,a)$

Monte-Carlo Policy Gradient

- Sample-based gradient estimation
- $\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s)$
- $\sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s) = \mathbb{E}_\pi[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t)]$
- We can now use this gradient estimation for PG steps
  - $\theta_{t+1} = \theta_t + \alpha \sum_a \widehat{q_\pi}(S_t,a) \nabla_\theta \pi(a|S_t;\theta)$
  - Requires an approximation to q_π, denoted $\hat{q}$, for every possible action
  - Can we avoid this approximation?

REINFORCE [Williams, 1992]

1. $\theta_{t+1} = \theta_t + \alpha \sum_a \widehat{q_\pi}(S_t,a) \nabla_\theta \pi(a|S_t)$
   - Can we avoid this approximation? YES!
2. $\nabla J(\theta) \propto \mathbb{E}_\pi[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t)]$
3. $= \mathbb{E}_\pi\left[\sum_a \pi(a|S_t;\theta) q_\pi(S_t,a) \frac{\nabla_\theta \pi(a|S_t)}{\pi(a|S_t)}\right]$ (multiplied and divided by the same number)
4. $= \mathbb{E}_\pi\left[q_\pi(S_t,A_t) \frac{\nabla_\theta \pi(A_t|S_t)}{\pi(A_t|S_t)}\right]$ ($\sum_x \Pr(x) f(x) = \mathbb{E}_{x\sim \Pr(x)}[f(x)]$)
5. $= \mathbb{E}_\pi\left[G_t \frac{\nabla_\theta \pi(A_t|S_t)}{\pi(A_t|S_t)}\right]$ ($\mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t)$)

- We can now use this gradient estimation for PG steps
  - $\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t)}{\pi(A_t|S_t)}$

REINFORCE [Williams, 1992]

- $\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$
- Each increment is proportional to the product of a return G_t and a vector
  - The gradient of the probability of taking the action actually taken over the (current) probability of taking that action
  - The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t
- The update increases the parameter vector (*) proportional to the return, and (**) inversely proportional to the action probability
  - (*) makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return
  - (**) makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return

REINFORCE [Williams, 1992]

- $\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$
- REINFORCE uses G_t: the complete return from time t until the end of the episode
- In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed

REINFORCE [Williams, 1992]

* In the literature you might encounter "grad log pi" instead of "grad ln pi". That's not a problem since: $\nabla \ln(f(x)) \propto \nabla \log(f(x))$

Specifically: $\nabla \ln(f(x)) = \nabla \log_a(f(x)) \ln(a)$

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

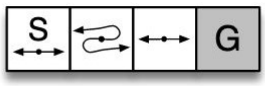
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

How did we get here from $\frac{\nabla_\theta \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$?

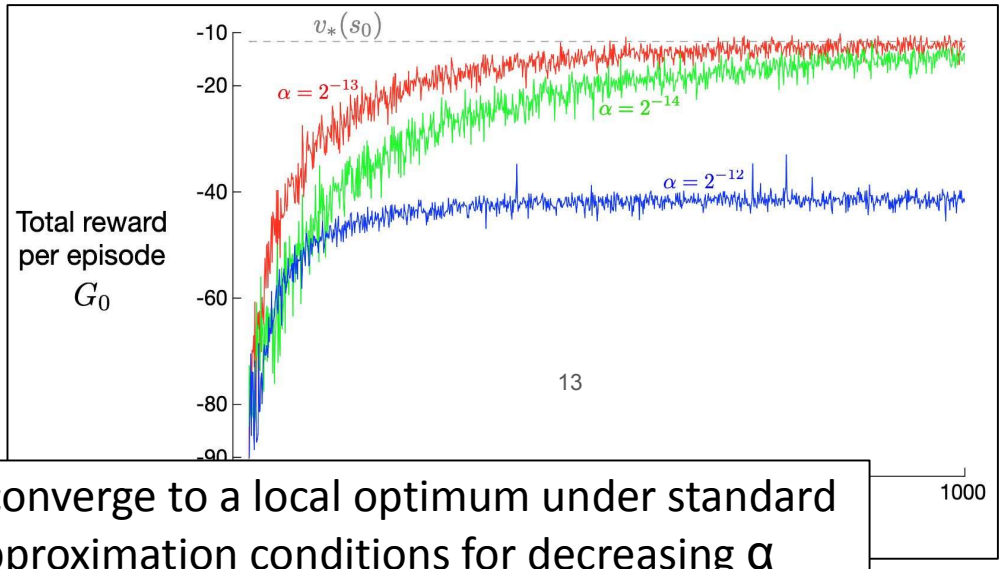
$$\nabla \ln(f(x)) = \frac{\nabla f(x)}{f(x)}$$

REINFORCE [Williams, 1992]

- The crazy corridor domain



- With the right step size, the total reward per episode approaches the optimal value of the start state



Guaranteed to converge to a local optimum under standard stochastic approximation conditions for decreasing α

Calibrating REINFORCE

- Imagine a domain with two possible episode trajectories (rollouts)
  - $\tau_1 = \{S_0, A_0, R_1, S_1, \dots, A_{T-1}, R_T, S_T\}$ with $G = 1001$
  - $\tau_2 = \{S'_0, A'_0, R'_1, S'_1, \dots, A'_{T-1}, R'_T, S'_T\}$ with $G' = 1000$
  - Further assume that $R_{i<T} = R'_i = 0$ and $R_T = G, R'_T = G'$
- Following REINFORCE: $\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \ln \pi(A_t|S_t;\theta)$
- How will the policy be updated after sampling each of the trajectories?
  - $\tau_1 ++, \tau_2 ++$
- How should the policy be updated after sampling each of the trajectories?
  - $\tau_1 +, \tau_2 -$
- How can we address this gap?

PG with baseline

- Normalize the returns with a baseline!
- $\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \big(q_\pi(s,a) - \mathrm{b(s)}\big) \nabla \pi(a|s)$
- Are we allowed to do that???
- Can we subtract $\sum_a b(s) \nabla \pi(a|s)$ from $\nabla J(\theta)$?
- Yes! As long as $b(s)$ isn't a function of a
- $\sum_a b(s) \nabla \pi(a|s) = b(s) \nabla \sum_a \pi(a|s) = b(s) \nabla 1 = 0$ (actions' probabilities sum to 1)
- REINFORCE with baseline: $\theta_{t+1} = \theta_t + \alpha \big(G_t - b(S_t)\big) \nabla_\theta \ln \pi_\theta(A_t|S_t)$

PG with baseline

- In the bandit algorithms the baseline was the average of the rewards seen so far
- For MDPs the baseline should be different for each states
  - In some states all actions have high (q) values, and we need a high baseline to differentiate the higher valued actions from the less highly valued ones
  - In other states all actions will have low values and a low baseline is appropriate
- What would be the equivalent of average reward (bandits) for a given state (MDP) ?
  - $v_\pi(s)$

PG with baseline

- $\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \big(q_\pi(s,a) - v_\pi(s)\big) \nabla \pi(a|s)$
- Actions that improve on the current policy are encouraged
  - $q_\pi(s,a) - v_\pi(s) > 0$
- Actions that damage the current policy are discouraged
  - $q_\pi(s,a) - v_\pi(s) < 0$
- Also known as the **advantage** of action a in state s
- How can we learn $v_\pi(s)$?
- Another function approximator $\hat{v}(s;w)$
  - On top of the policy

REINFORCE with baseline

REINFORCE with Baseline (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

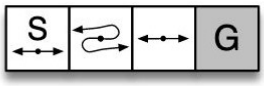
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\delta \leftarrow G_t - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla_{\mathbf{w}} \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$

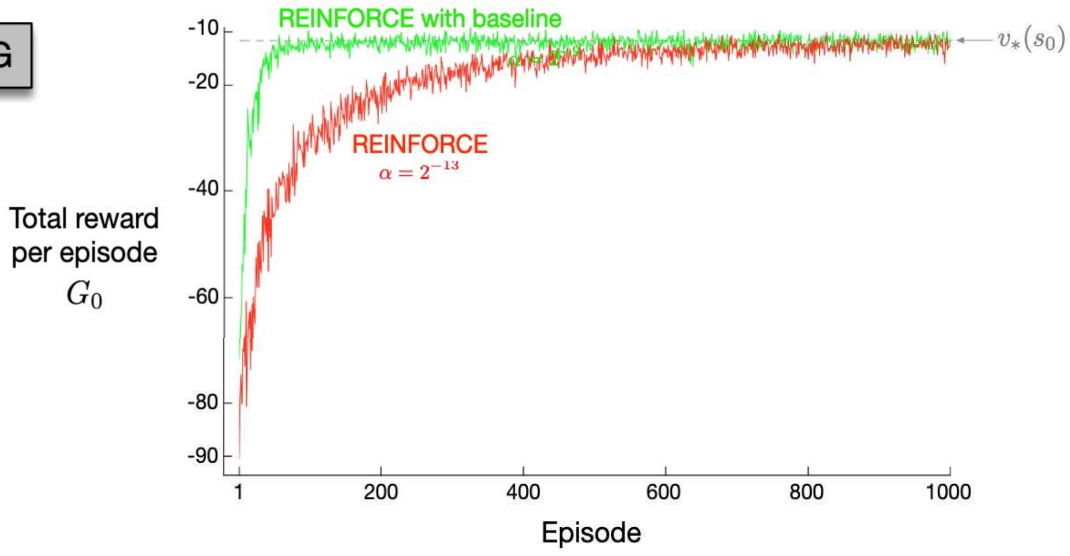
Each approximator has its unique learning rate

REINFORCE with baseline

- The crazy corridor domain

- $\alpha^{\theta} = 2^{-9}$
- $\alpha^{w} = 2^{-6}$





Policy Gradient

- $\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$
- PG theorem: $\widehat{\nabla J(\theta_t)} \propto \big(q_\pi(S_t, A_t) - b(S_t)\big)\nabla_\theta \log \pi(A_t|S_t;\theta)$
  - REINFORCE+baseline: $\theta_{t+1} = \theta_t + \alpha\big(G_t - b(S_t)\big)\nabla_\theta \log \pi(A_t|S_t;\theta)$
- **Pros**: approximating the optimal policy directly is more accurate and faster to converge in many domains when compared to value-based approaches
- **Cons**: using G_t as an estimator for $q_\pi(S_t, A_t)$ is noisy (high variance) though unbiased
  - Unstable learning

Add a critic

- $\widehat{\nabla J(\theta_t)} \propto \big(q_\pi(S_t, A_t) - b(S_t)\big)\frac{\nabla_\theta \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$
- Define a new estimator: $\hat{q}_\pi(s, a; \theta) \approx q_\pi(s, a)$
  - To be used instead of G_t in REINFORCE
- How should we train $\hat{q}_\pi(s, a; \theta)$?
  - Monte-Carlo updates
    - High variance samples
    - Requires a full episode
  - Bootstrapping e.g., Q-learning
    - Lower variance (though introduces bias)

Critic's duties

1. Approximate state or action or advantage ($q(s,a) - v(s)$) values
2. Trained via bootstrapping, criticizes the action chosen by the actor

   adjusting the actor's (policy) gradient

- Is REINFORCE (+ state value baseline) an actor-critic framework?
  - $\theta_{t+1} = \theta_t + \alpha\big(G_t - \hat{v}_\pi(S_t)\big)\nabla_\theta \ln \pi(A_t|S_t;\theta)$
  - NO! the state-value function is used only as a baseline, not as a critic. Moreover, it doesn't utilize bootstrapping (uses high-variance MC updates)
- The bias introduced through bootstrapping is often worthwhile because it reduces variance and accelerates learning

Benefits from a critic

- REINFORCE with baseline is unbiased* and will converge asymptotically to a local optimum
  - * With a linear state value approximator, and when *b* is not a function of *a*
  - Like all Monte-Carlo methods it tends to learn slowly (produce estimates of high variance)
  - Not suitable for online or for continuing problems
- Temporal-difference methods can eliminate these inconveniences
- In order to gain the TD advantages in the case of policy gradient methods we use actor–critic methods

Actor+critic

- Actor-critic algorithms are a derivative of policy iteration, which alternates between policy evaluation—computing the value function for a policy—and policy improvement—using the value function to obtain a better policy
- In large-scale reinforcement learning problems, it is typically impractical to run either of these steps to convergence, and instead the value function and policy are optimized jointly
- The policy is referred to as the actor, and the value function as the critic

Advantage function

- Eventually we would like to shift the policy towards actions that result in higher return
- what is the benefit from taking action a at state s while following policy π?
  - $A_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$
- This resembles PG with baseline but not the same as q_π is approximated:
  - $\widehat{\nabla J(\theta_t)} = A_\pi(s_t,a_t)\nabla_\theta \ln \pi(a_t|s_t;\theta)$
- Actions that improve on the current policy are encouraged
  - $A_\pi(s,a) > 0$
- Actions that damage the current policy are discouraged
  - $A_\pi(s,a) < 0$

One-step actor-critic

- $A_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$
  - Does that mean that approximating the advantage function requires two function approximators ($\hat{q}$ and $\hat{v}$) ?
  - No since q values can be derived from state values + one step transition
  - $\hat{q}_\pi(s_t,a_t) - \hat{v}_\pi(s_t;w) = \widehat{\mathbb{E}}[r_{t+1} + \gamma\hat{v}(s_{t+1};w)] - \hat{v}(s_t;w)$
- $\theta_{t+1} = \theta_t + \alpha\big(r_{t+1} + \gamma\hat{v}(s_{t+1};w) - \hat{v}(s_t;w)\big)\nabla_\theta \ln \pi(a_t|s_t;\theta)$

δ – TD error

- One-step Actor-Critic is a fully online, incremental algorithm, with states, actions, and rewards processed as they occur and then never revisited

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\ \delta\nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Keep track of accumulated discount

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Follow the current policy

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Compute the TD error

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Update the critic without the accumulated discount. (The discount factor is included in the TD error)

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Update the actor with discounting. Early actions matter more.

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Update accumulated discount and progress to the next state

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \; \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

In practice: training the network at every step on a single observation is inefficient (slow and correlated)

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} v(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Instead: store all state approx values, log probabilities, and rewards along the episode. Train once at the end of the episode.

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

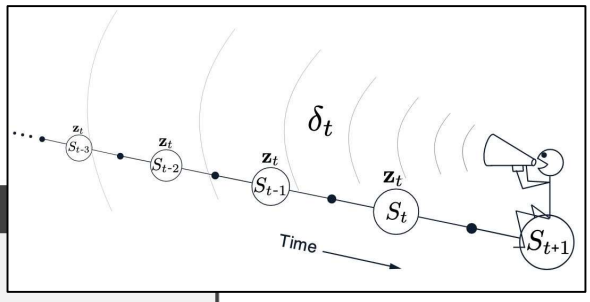
Instead: store all state approx. values, log probabilities, and rewards along the episode. Train once at the end of the episode.

```
values = torch.FloatTensor(values)
Qvals = torch.FloatTensor(Qvals)
log_probs = torch.stack(log_probs)
advantage = Qvals - values
```

Advantage Actor-Critic (A2C)

- Initialize the actor π_θ and the critic $\hat{V}_w$
- For each episode:
  - Init empty episode memory
  - $s = init\ env$
  - For each time step:
    - $a \sim \pi(s)$
    - $s', r_t \sim env(s, a)$
    - Store (s, a, r) in episode memory
    - $s = s'$
  - Reset $d\theta = 0, dw = 0$
  - Backwards iteration (i from length to 0) over the episode
    - Compute advantage $\delta_t = r_i + \gamma \hat{V}_w(s_{i+1}) - \hat{V}_w(s_i)$
    - Accumulate the policy gradient using the critic: $d\theta \leftarrow d\theta + \delta \nabla_\theta \log \pi_\theta(s_i, a_i)$
    - Accumulate the critic gradient: $dw \leftarrow dw + \delta \nabla_w \hat{V}_w(s_i)$
    - Update the actor and the critic with the accumulated gradients using gradient descent

Add eligibility traces



Actor–Critic with Eligibility Traces (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s, \mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0, 1]$, $\lambda^{\mathbf{w}} \in [0, 1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$
Repeat forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  While S is not terminal (for each time step):
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla_{\boldsymbol{\theta}} \ln \pi(A|S, \boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Gradient eligibility per tunable parameter for both the actor and the critic approximators

What did we learn?

- In many domains it's more efficient to learn the policy directly
  - Instead of deriving one from state or action values
- Applicable for continuous action spaces and stochastic policies
- Approximate the change to the policy that results in the highest increase in the expected return: $\widehat{\nabla_\theta J(\theta)}$
- Such approximation is made possible when by the policy gradient theorem
- Should we reinforce polices that result in high return?
  - Not always, a different policy might yield higher return
  - We need to use a baseline to determine if a policy is **relatively** good

What did we learn?

- REINFORCE:
  - $\widehat{\nabla J(\theta_t)} = G_t \nabla_\theta \ln \pi(A_t|S_t;\theta)$
- Q Actor-Critic:
  - $\widehat{\nabla J(\theta_t)} = \hat{q}(S_t, A_t; w) \nabla_\theta \ln \pi(A_t|S_t;\theta)$
- REINFORCE + baseline:
  - $\widehat{\nabla J(\theta_t)} = (G_t - \hat{v}(S_t; w)) \nabla_\theta \ln \pi(A_t|S_t;\theta)$
- Advantage Actor-Critic:
  - $\widehat{\nabla J(\theta_t)} = (R_{t+1} + \gamma \hat{v}(S_{t+1}; w) - \hat{v}(S_t; w)) \nabla_\theta \ln \pi(A_t|S_t;\theta)$

Required Readings

1. Chapter-6 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

AIMLC ZG512 -
Deep Reinforcement Learning

Session #14:
Model Based Algorithms

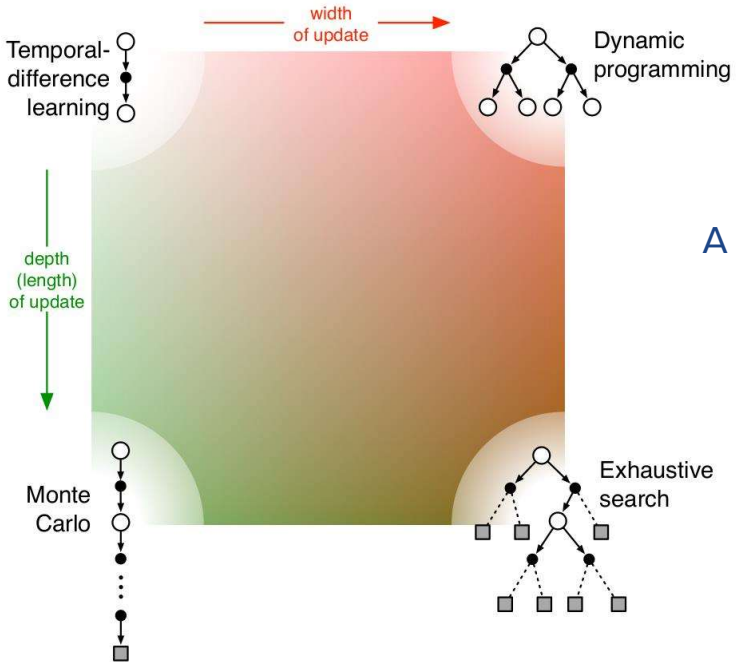
S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Agenda for the class

- ➔ Introduction
- ➔ Upper-Confidence-Bound [UCB] Action Selection
- ➔ Monte-Carlo Tree Search [MCTS]
- ➔ AlphaGo & AlphaGo Zero [Next Class]
- ➔ MuZero, PlaNet [Next Class]

S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Monte-Carlo Tree Search (MCTS)



A summary of pre-mid sem coverage !!!

Monte-Carlo Tree Search (MCTS)

Rollout Algorithms:

- Decision-time planning algorithms
- Produce Monte-Carlo estimates of action values only for each current state and for a given policy (Rollout policy)
- Simple, as there is no need to approximate a function over either the
  - entire state space (or)
  - state-action space
- How & Why?
  - Averaging the returns of the simulated trajectories produces estimates of $q\pi(s, a')$ for each action $a' \in A(s)$.
  - The policy selects an action in s that maximizes these estimates & then follows $\boldsymbol{\pi}$
- Aim of a rollout algorithm is to improve upon the rollout policy
  - Rollout policy could be completely random !!!

Monte-Carlo Tree Search (MCTS)

Rollout Algorithms:

- Decision-time planning algorithms
- Produce Monte-Carlo estimates of action values only for each current state and for a given policy (Rollout policy)
- Simple, as there is no need to approximate a function over either the
  - entire state space (or)
  - state-action space
- How & Why?
  - Averaging the returns of the simulated trajectories produces estimates of $q\pi(s, a')$ for each action $a' \in A(s)$.
  - The policy selects an action in s that maximizes these estimates & then follows $\boldsymbol{\pi}$
- Aim of a rollout algorithm is to improve upon the rollout policy
  - Rollout policy could be completely random !!!

- MCTS is a recent and strikingly successful example of decision-time planning
- An enhanced rollout algorithm
  - Accumulates value estimates obtained from the simulations to successively direct simulations toward more highly-rewarding trajectories

Monte-Carlo Tree Search (MCTS)

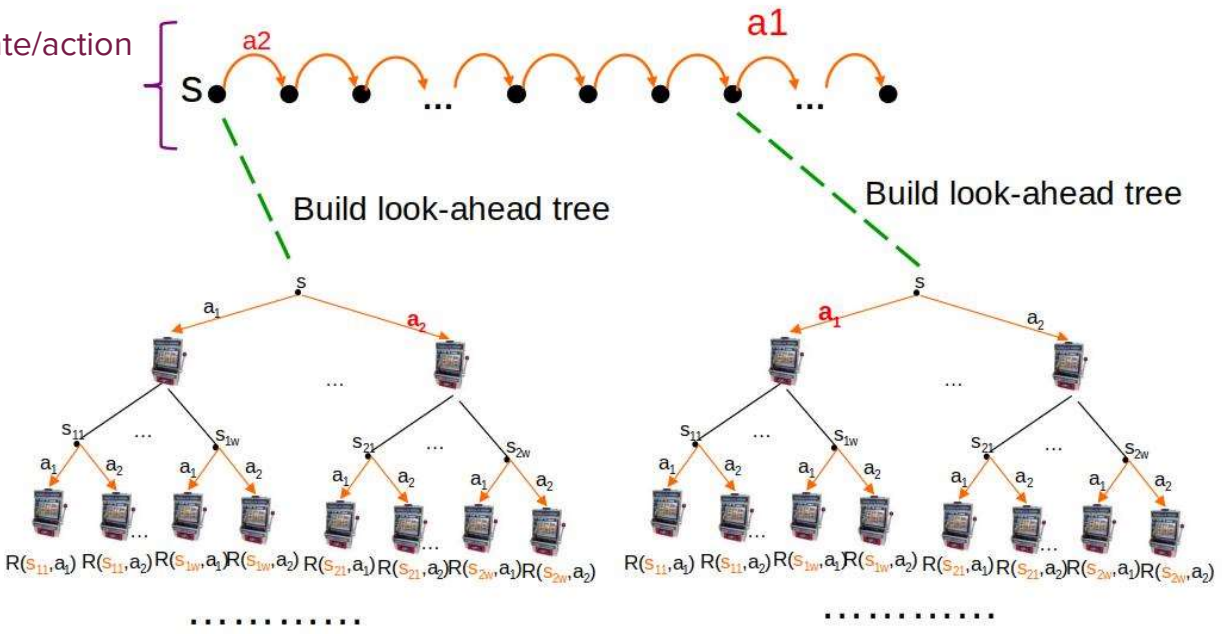
How MCTS works?

- MCTS is *executed* after encountering each new state (s)
  - [?] to select the agent's action for s
- *Each execution is an iterative process* that simulates many trajectories starting from s and
  - running to a terminal state (or)
  - until discounting makes any further reward negligible to the return
- Focus on multiple simulations starting at s by extending the initial portions of trajectories that have received high evaluations from earlier simulations.

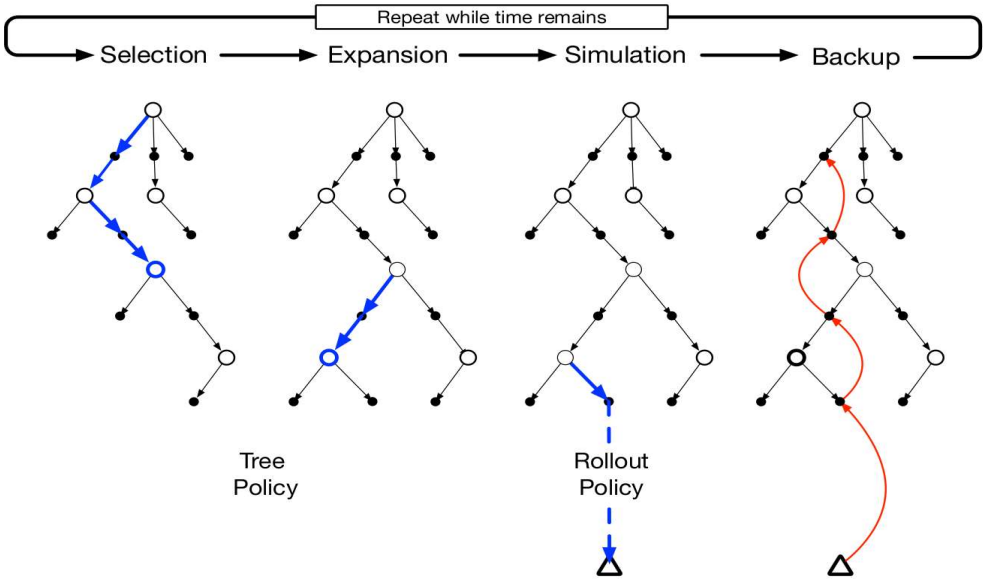
Slide from Prof.Alan Fern, Oregon State University

Monte-Carlo Tree Search (MCTS)

Real world state/action sequence



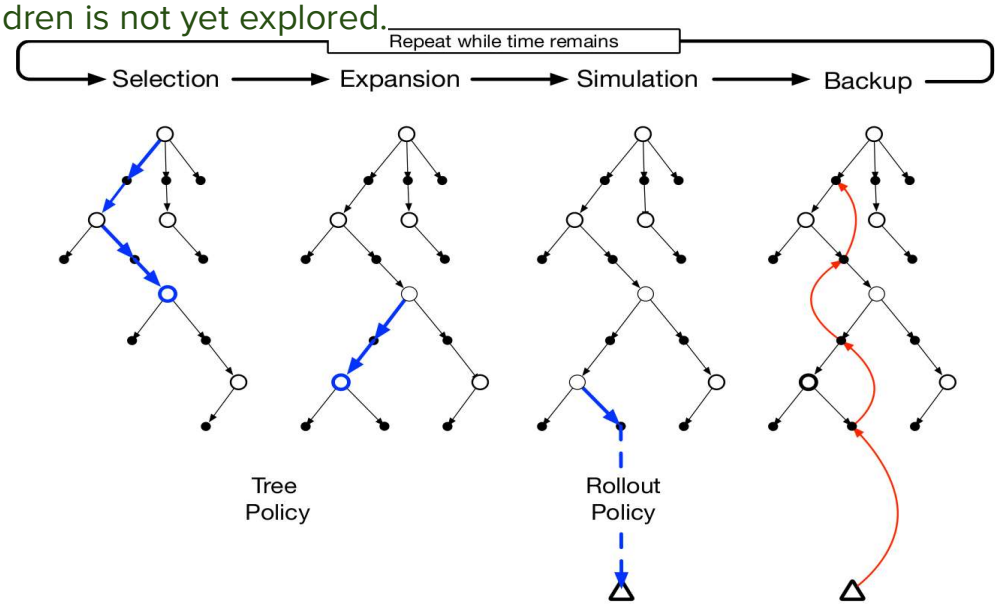
Monte-Carlo Tree Search (MCTS)



$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

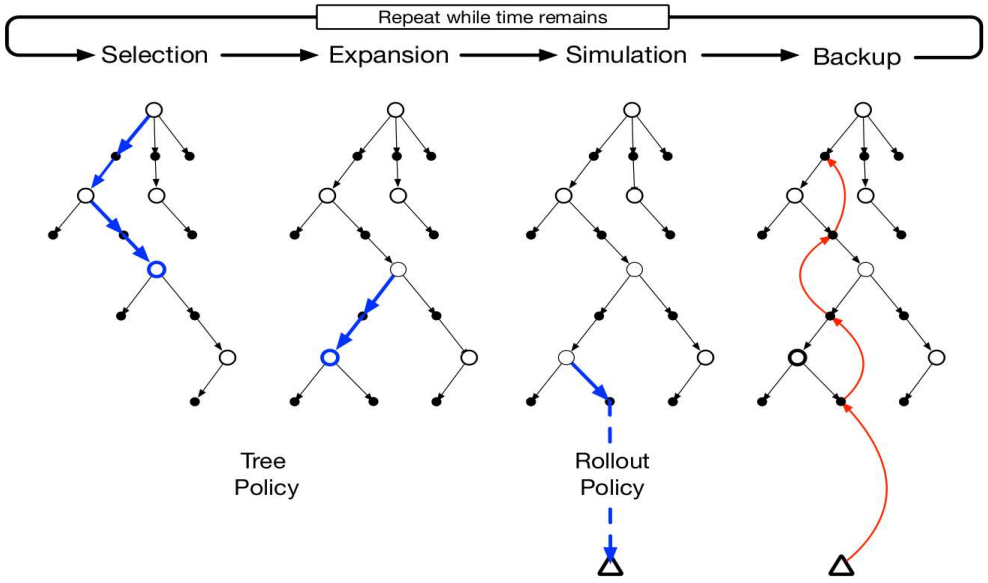
Monte-Carlo Tree Search (MCTS) -- Selection

Select: Select a single node in the tree that is *not fully expanded*. By this, we mean at least one of its children is not yet explored.



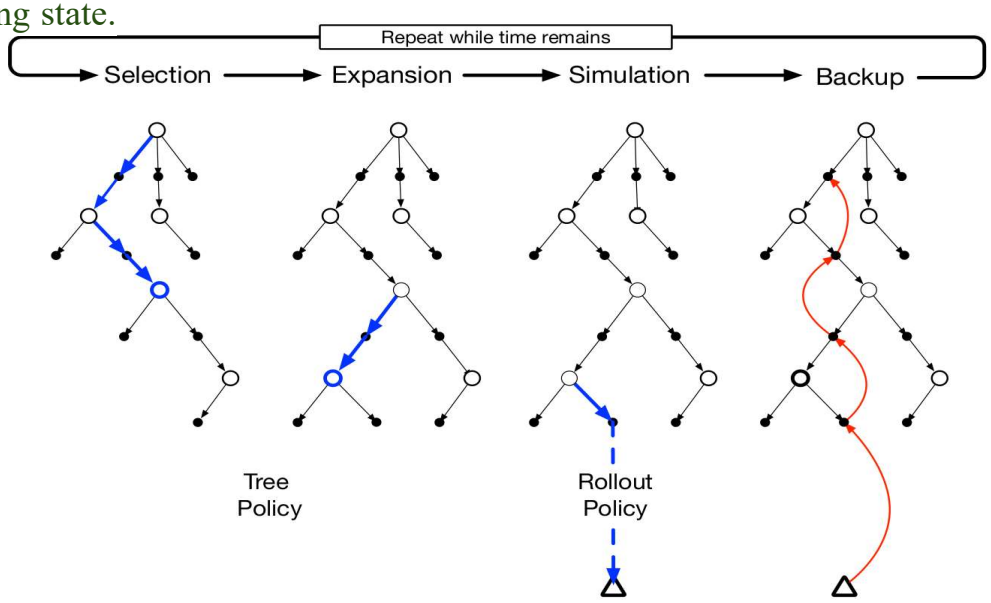
Monte-Carlo Tree Search (MCTS) -- Expansion

Expand: Expand this node by applying one available action (as defined by the MDP) from the node.



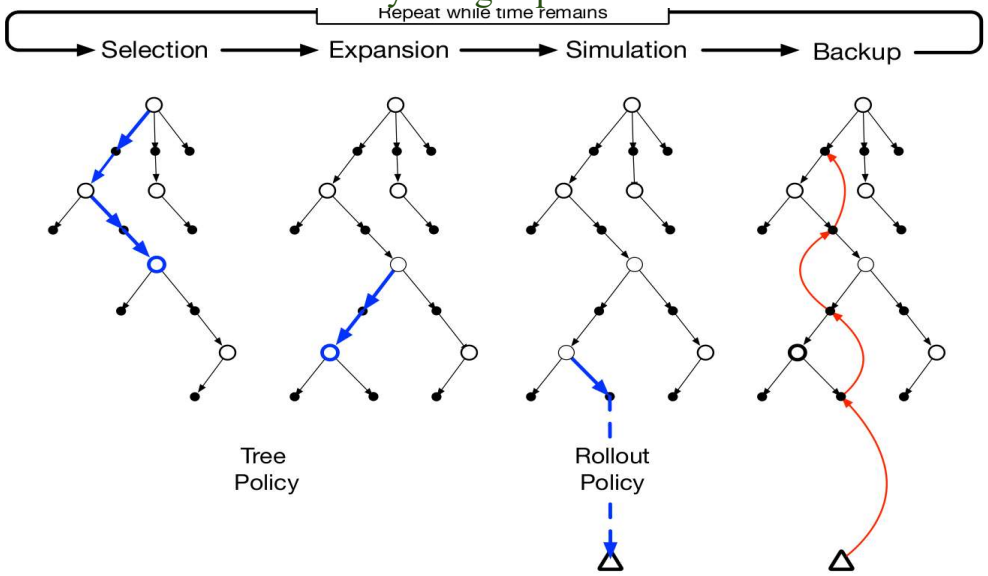
Monte-Carlo Tree Search (MCTS) -- Simulation

Simulation: From one of the outcomes of the expanded, perform a complete random simulation oto a terminating state.



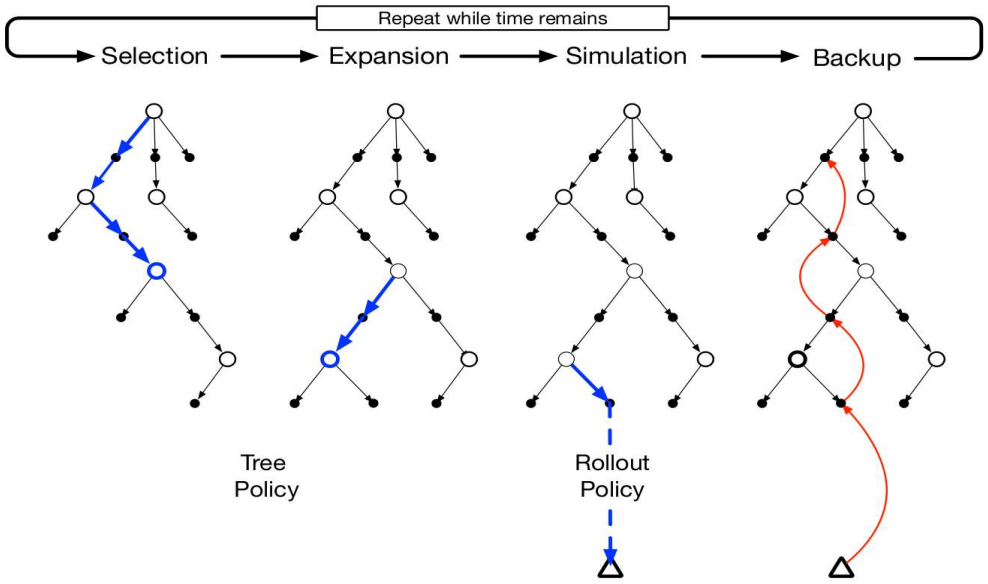
Monte-Carlo Tree Search (MCTS) -- Backup

Backup/ Backpropagate: The value of the node is *back propagated* to the root node, updating the value of each ancestor node on the way using expected value



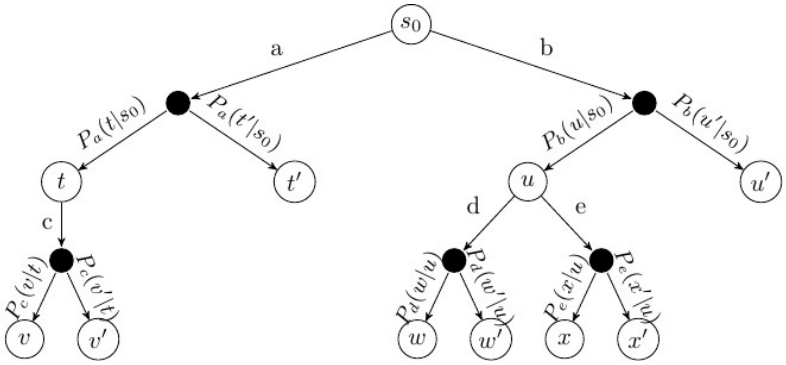
Monte-Carlo Tree Search (MCTS) -- Summarizing

Comments on the overall approach,,,,

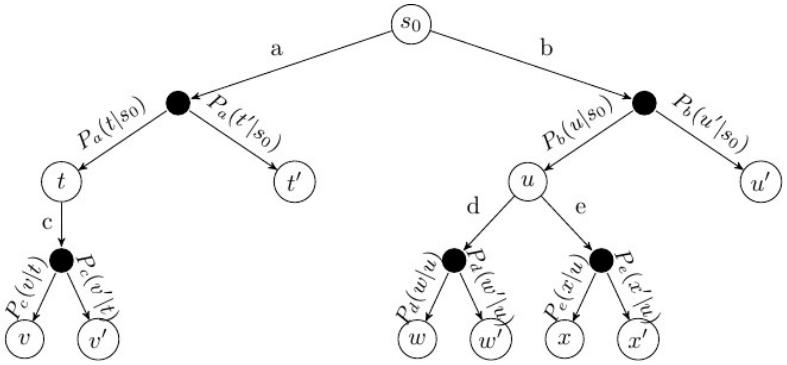


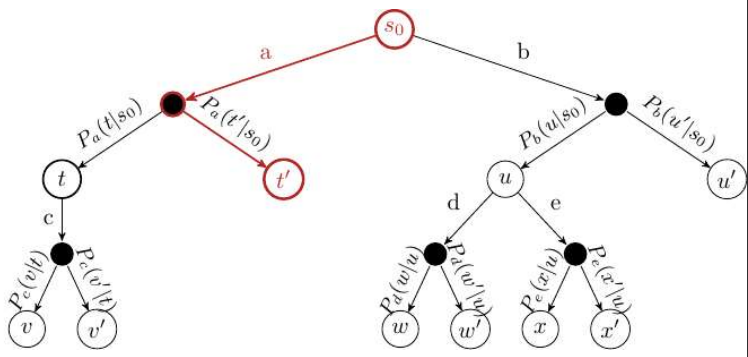
COMP90054: AI Planning for Autonomy ; https://gibberblot.github.io/rl-notes/single-agent/mcts.html

Monte-Carlo Tree Search (MCTS) -- Selection

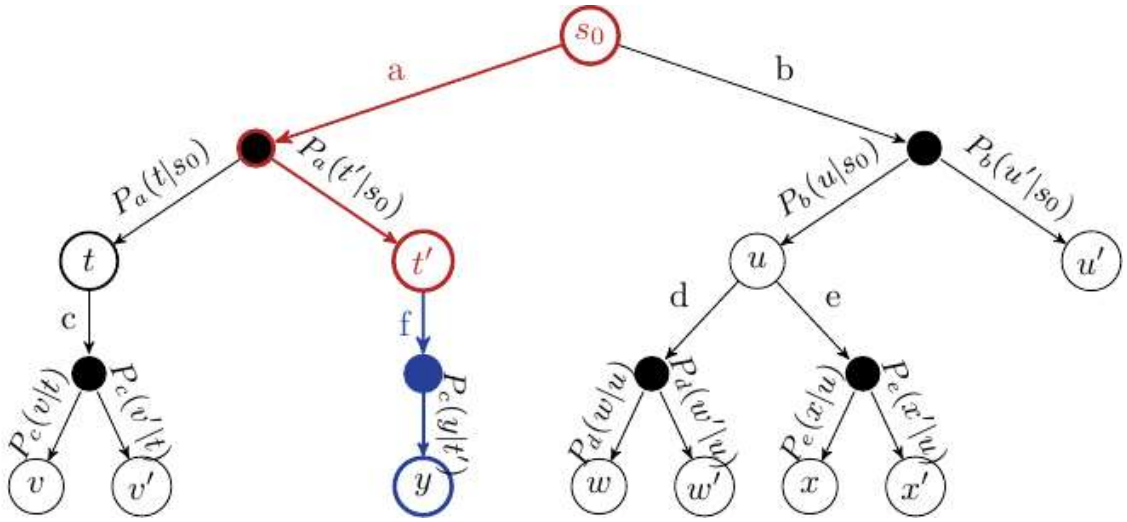


Monte-Carlo Tree Search (MCTS) -- Selection

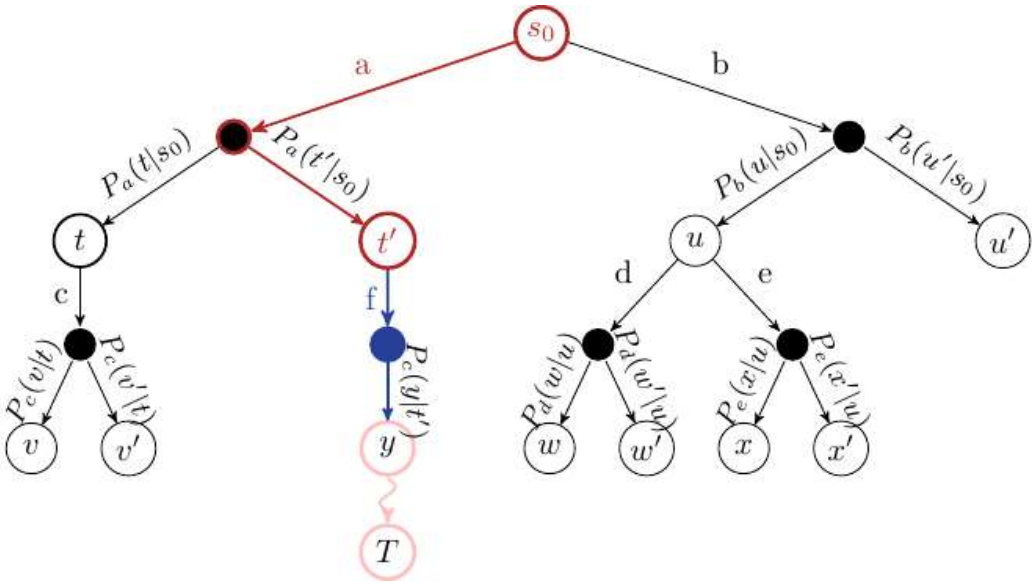




Monte-Carlo Tree Search (MCTS) -- Expansion

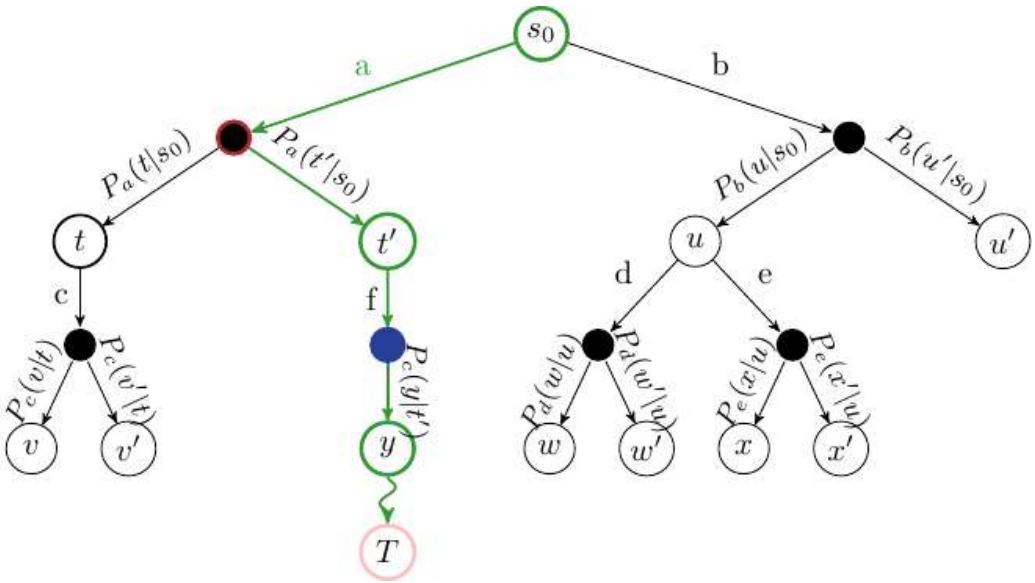


Monte-Carlo Tree Search (MCTS) -- Simulation



COMP90054: AI Planning for Autonomy ;
https://gibberblot.github.io/rl-notes/single-agent/mcts.html

Monte-Carlo Tree Search (MCTS) -- Backup



Monte-Carlo Tree Search (MCTS)

Algorithm – Monte-Carlo Tree Search

Input: MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s, a, s') \rangle$, base value function Q, time limit T.
Output: updated Q-function Q

while $currentTime < T$
    $selected\_node \leftarrow \text{Select}(s_0)$
    $child \leftarrow \text{Expand}(selected\_node)$ – expand and choose a child to simulate
    $G \leftarrow \text{Simulate}(child)$ – simulate from $child$
    $\text{Backpropagate}(selected\_node, child, G)$
return Q

COMP90054: AI Planning for Autonomy ;
https://gibberblot.github.io/rl-notes/single-agent/mcts.html

Monte-Carlo Tree Search (MCTS)

Function – $\text{Select}(s : S)$

Input: state s
Output: unexpanded state

while s is fully expanded
    Select action a to apply in s using a multi-armed bandit algorithm
    Choose one outcome s' according to $P_a(s' \mid s)$
    $s \leftarrow s'$
return s

S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Monte-Carlo Tree Search (MCTS)

Function – $\text{Expand}(s : S)$

Input: state s
Output: expanded state s'

Select an action a from s to apply
Expand one outcome s' according to the distribution $P_a(s' \mid s)$ and observe reward r
return s'

COMP90054: AI Planning for Autonomy ;
https://gibberblot.github.io/rl-notes/single-agent/mcts.html

Monte-Carlo Tree Search (MCTS)

Procedure – $\text{Backpropagation}(s : S; a : A)$

Input: state-action pair (s, a)
Output: none

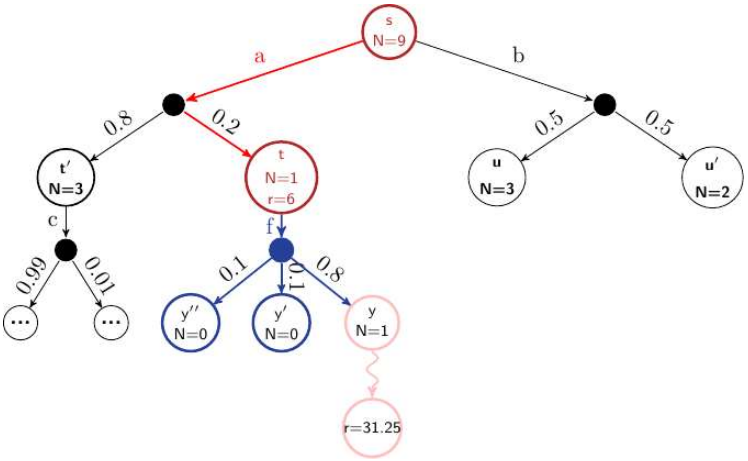
do

$N(s, a) \leftarrow N(s, a) + 1$
$G \leftarrow r + \gamma G$
$Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}[G - Q(s, a)]$
$s \leftarrow$ parent of s
$a \leftarrow$ parent action of s

while $s \neq s_0$

S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Monte-Carlo Tree Search (MCTS)

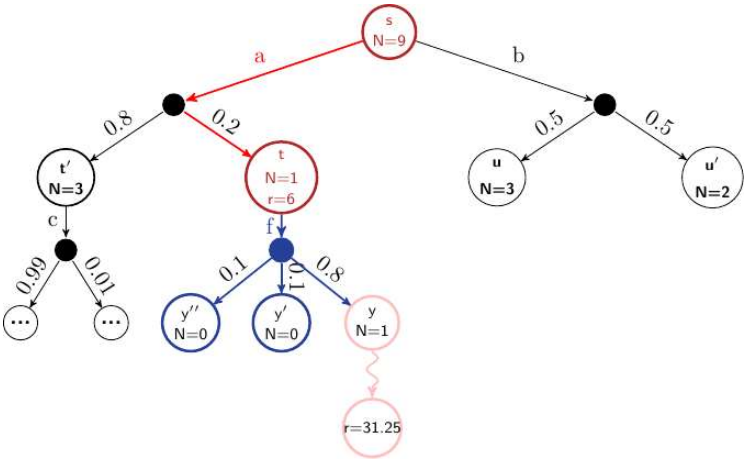


Before backpropagation

$$Q(s,a) = 18$$
$$Q(t,f) = 0$$

COMP90054: AI Planning for Autonomy ;
https://gibberblot.github.io/rl-notes/single-agent/mcts.html

Monte-Carlo Tree Search (MCTS)



Before backpropagation

$$Q(s,a) = 18$$
$$Q(t,f) = 0$$

The backpropagation step is then calculated for the nodes y, t, and s as follows:

$$\begin{aligned} Q(y,g) &= \gamma^2 \times 31.25 \text{ (simulation is 3 steps long and receives reward of 31.25)} \\ &= 20 \end{aligned}$$

$$\begin{aligned} N(t,f) &\leftarrow N(t,f)+1 = N(y)+N(y')+N(y'')+1 = 2 \\ Q(t,f) &= Q(t,f) + \tfrac{1}{N(t,f)}[r+\gamma G - Q(t,f)] \\ &= 0 + \tfrac{1}{2}[0 + 0.8 \cdot 20 - 0] \\ &= 8 \end{aligned}$$

$$\begin{aligned} N(s,a) &\leftarrow N(s,a)+1 = N(t)+N(t')+1 = 5 \\ Q(s,a) &= Q(s,a) + \tfrac{1}{N(s,a)}[r+\gamma G - Q(s,a)] \\ &= 18 + \tfrac{1}{5}[6 + 0.8 \cdot (0.8 \cdot 20) - 18] \\ &= 18 + \tfrac{1}{5}[6 + 12.8 - 18] \\ &= 18.16 \end{aligned}$$

S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Upper-Confidence-Bound Action Selection

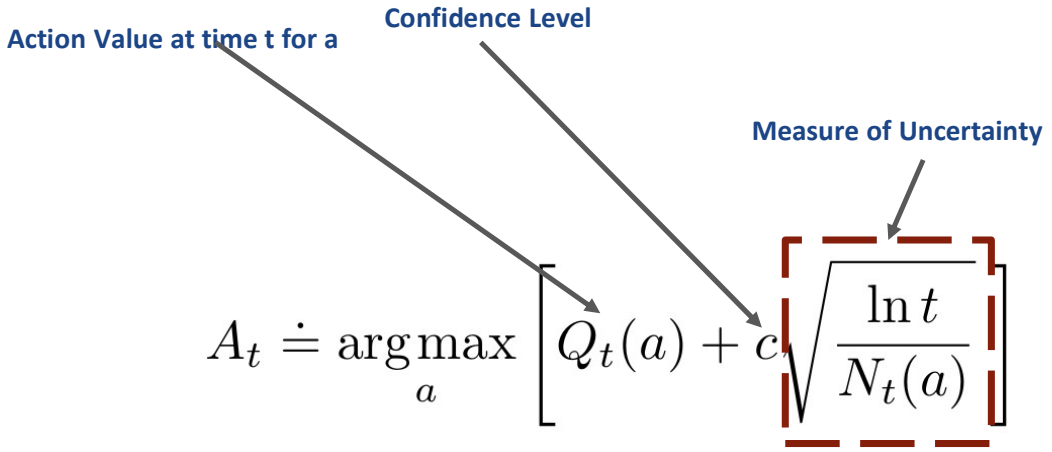
- ε-greedy action selection forces the non-greedy actions to be tried,
  - Indiscriminately, with no preference for those that are nearly greedy or particularly uncertain
- It would be better to select among the non-greedy actions according to their potential for actually being optimal
  - Take into account both how close their estimates are to being maximal and the uncertainties in those estimates.

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Upper-Confidence-Bound Action Selection

- Each time a is selected the uncertainty is presumably reduced
- Each time an action other than a is selected, t increases but $N_t(a)$ does not; because t appears in the numerator, the uncertainty estimate increases.
- Actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time



S. P. Vimal, Department of CSIS, WILP Division (vimalsp@wilp.bits-pilani.ac.in)

Deep Reinforcement Learning

2022-23 Second Semester, M.Tech (AIML)

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Session #15:
Imitation Learning

Instructors :
1. Prof. S. P. Vimal (vimalsp@wilp.bits-pilani.ac.in),
2. Prof. Sangeetha Viswanathan (sangeetha.viswanathan@pilani.bits-pilani.ac.in)

Agenda for the classes

- Imitation Learning
  - Behaviour Cloning
  - Inverse RL

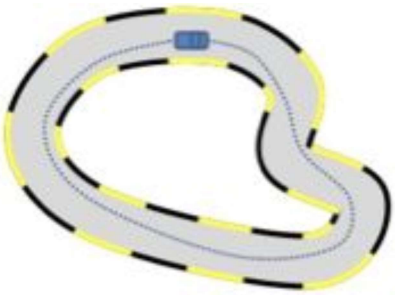
Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

Imitation Learning in a Nutshell

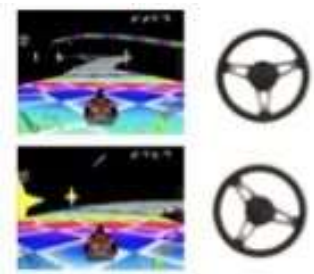
Given: demonstrations or demonstrator

Goal: train a policy to mimic demonstrations

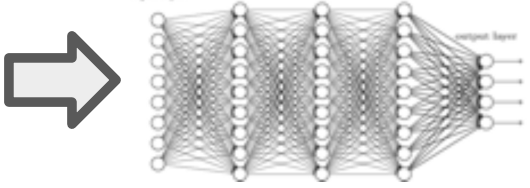
Expert Demonstrations



State/Action Pairs

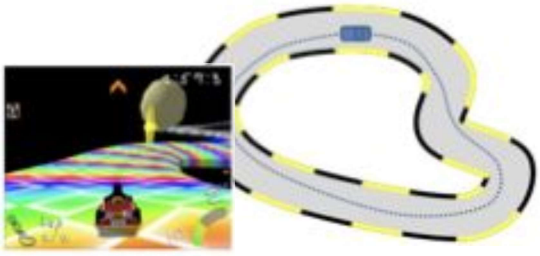


Learning



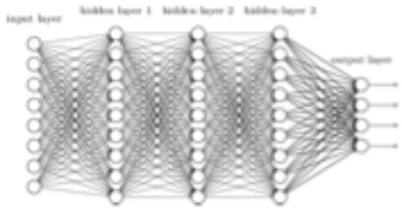
Images from Stephane Ross

Ingredients of Imitation Learning

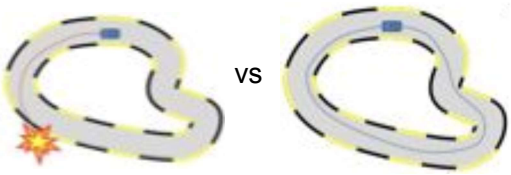


Demonstrations or Demonstrator

Environment / Simulator



Policy Class



Loss Function

Learning Algorithm

Some Interesting Examples

- ALVINN
  https://www.ri.cmu.edu/publications/alvinn-an-auton`omous-land-vehicle-in-a-neural-network/
  Dean Pomerleau et al., 1989-1999 https://www.youtube.com/watch?v=ilP4aPDTBPE
- Helicopter Acrobatics

Learning for Control from Multiple Demonstrations - Adam Coates, Pieter Abbeel, Andrew Ng, ICML 2008
An Application of Reinforcement Learning to Aerobatic Helicopter Flight - Pieter Abbeel, Adam Coates, Morgan Quigley, Andrew Y. Ng, NIPS 2006

https://www.youtube.com/watch?v=0JL04JJjocc

- Ghosting (Sports Analytics) - Next Slide.

Some Interesting Examples

Some Interesting Examples

What's Hidden in the Hidden Layers?

The contents can be easy to find with a geometrical problem, but the hidden layers have yet to give up all their secrets

David S. Touretzky and Dean A. Pomerleau

tions, we fed the network road images taken under a wide variety of viewing angles and lighting conditions. It would be impractical to try to collect thousands of real road images for such a data set. Instead, we developed a synthetic road-image generator that can create as many training examples as we need.

To train the network, 1200 simulated road images are presented 40 times each, while the weights are adjusted using the back-propagation learning algorithm. This takes about 30 minutes on Carnegie Mellon's Warp systolic-array supercomputer. (This machine was designed at Carnegie Mellon and is built by General Electric. It has a peak rate of 100 million floating-point operations per second and can compute weight adjustments for back-propagation networks at a rate of 20 million connections per second.)

Once it is trained, ALVINN can accurately drive the NAVLAB vehicle at about 3½ miles per hour along a path through a wooded area adjoining the Carnegie Mellon campus, under a variety of weather and lighting conditions. This speed is nearly twice as fast as that achieved by non-neural-network algorithms running on the same vehicle. Part of the reason for this is that the forward pass of a back-propagation network can be computed quickly. It takes about 200 milliseconds on the Sun-3/160 workstation installed on the NAVLAB.

The hidden-layer representations ALVINN develops are interesting. When trained on roads of a fixed width, the network chooses a representation in which hidden units act as detectors for complete roads at various positions and orientations. When trained on roads of variable

continued

Photo 1: *The NAVLAB autonomous navigation test-bed vehicle and the road used for trial runs.*

Some Interesting Examples

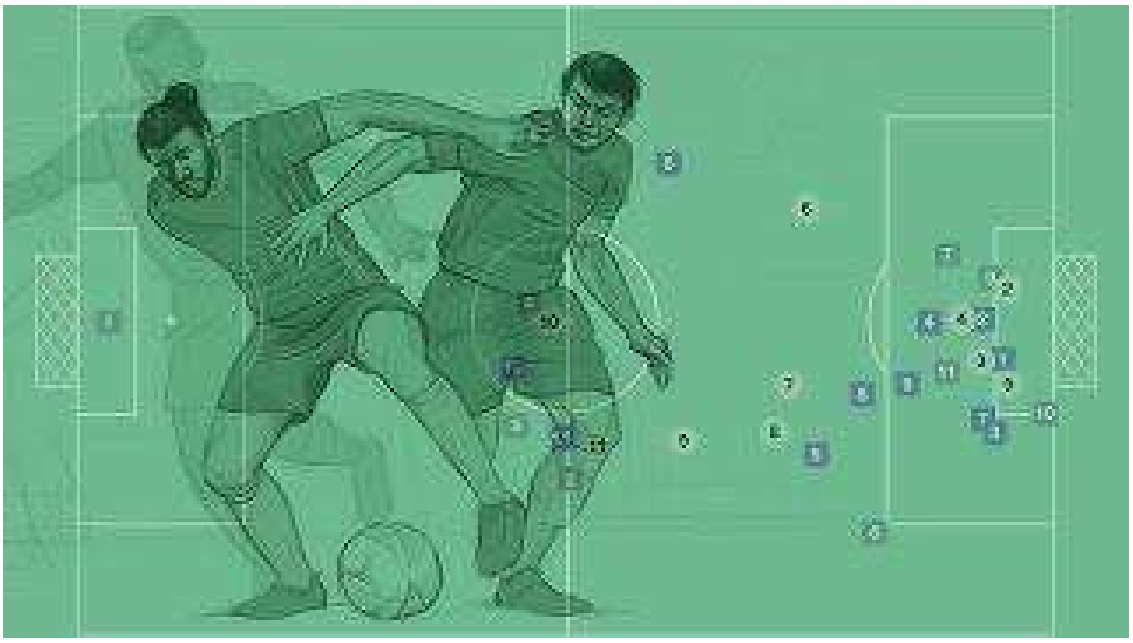
Learning for Control from Multiple Demonstrations - Adam Coates, Pieter Abbeel, Andrew Ng, ICML 2008
An Application of Reinforcement Learning to Aerobatic Helicopter Flight - Pieter Abbeel, Adam Coates, Morgan Quigley, Andrew Y. Ng, NIPS 2006

Ghosting

Data Driven Ghosting using Deep Imitation Learning
Hoang M. Le et al., SSAC 2017

https://www.youtube.com/watch?v=WI-WL2cj0CA

Some Interesting Examples



Notation & Set-up

State: s (sometimes x) (**state may only be partially observed)

Action: a (sometimes y)

Policy: π_θ (sometimes h)

- Policy maps states to actions: $\pi_\theta(s) \rightarrow a$
- ...or distributions over actions: $\pi_\theta(s) \rightarrow P(a)$

State Dynamics: P(s'|s,a)

- Typically not known to policy.
- Essentially the simulator/environment

Notation & Set-up

Rollout: sequentially execute $\pi(s_0)$ on an initial state

- Produce trajectory $\boldsymbol{\tau}=(s_0,a_0,s_1,a_1,\ldots)$

$P(\boldsymbol{\tau}|\pi)$: distribution of trajectories induced by a policy

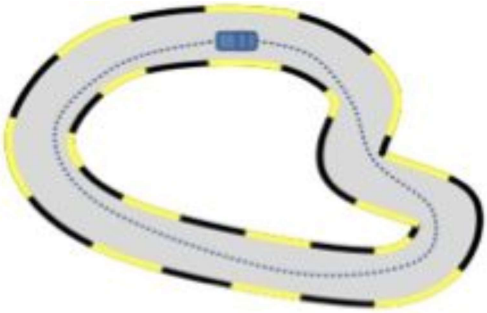
1. Sample s_0 from P_0 (distribution over initial states), initialize t = 1.
2. Sample action a_i from $\pi(s_{t-1})$
3. Sample next state s_t from applying a_t to s_{t-1} (requires access to environment)
4. Repeat from Step 2 with t=t+1

$P(s|\pi)$: distribution of states induced by a policy

- Let $P_t(s|\pi)$ denote distribution over t-th state
- $P(s|\pi) = (1/T)\sum_t P_t(s|\pi)$

Example #1: Racing Game

(Super Tux Kart)



s = game screen

a = turning angle

Training set: D={r:=(**s**,**a**)} from π^*

- **s** = sequence of s
- **a** = sequence of a

Goal: learn $\pi_\theta(s)\rightarrow a$

Images from Stephane Ross

Example #2: Basketball Trajectories

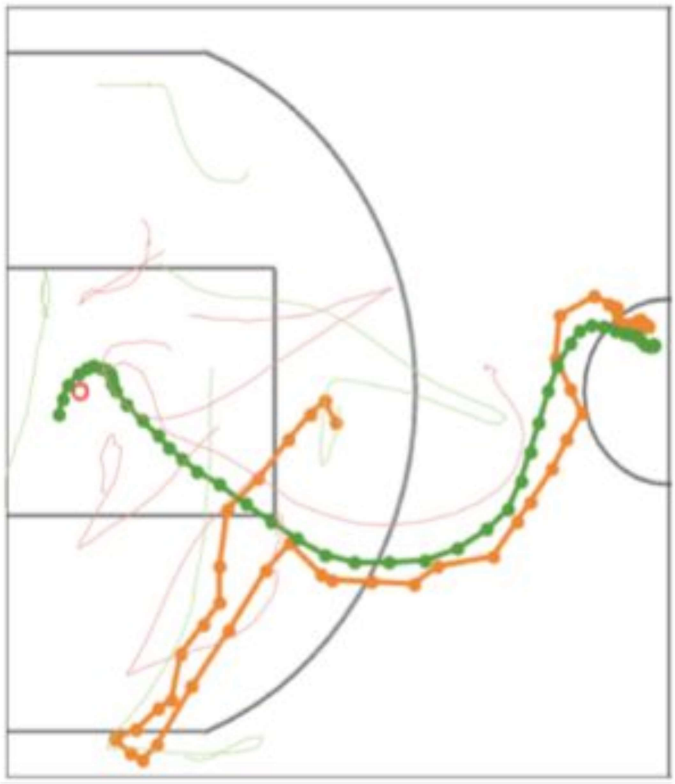
s = location of players & ball

a = next location of player

Training set: D={r:=(**s**,**a**)} from π^*

- **s** = sequence of s
- **a** = sequence of a

Goal: learn π_θ(s)→a

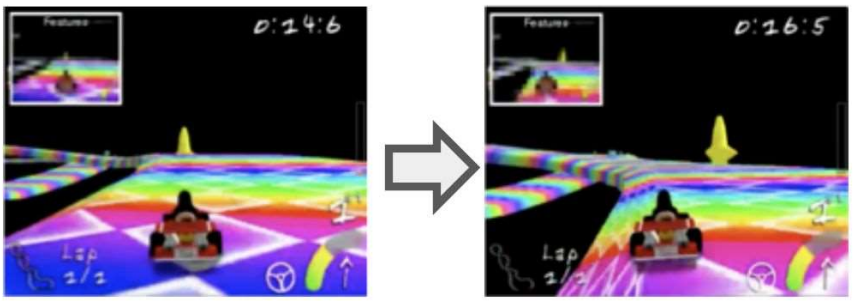


Behavioral Cloning = Reduction to Supervised Learning (Ignoring regularization for brevity.)

Write:

Behavioral Cloning vs. Imitation Learning

Limitations of Behavioral Cloning

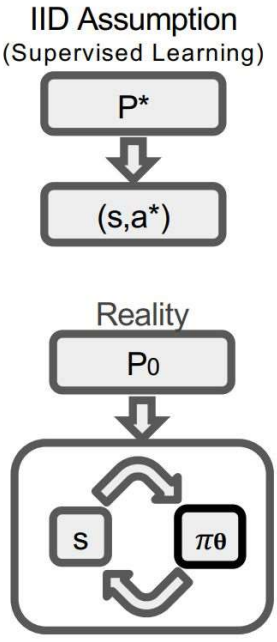


π_θ makes a mistake

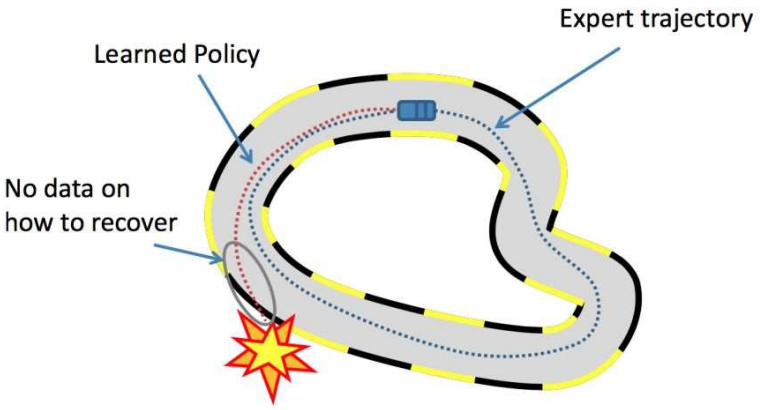
New state sampled not from P*!

Worst case is catastrophic!

Images from Stephane Ross



Limitations of Behavioral Cloning



Compounding Errors

Data distribution mismatch!

In supervised learning, $(x, y) \sim D$ during train **and** test. In MDPs:

- Train: $s_t \sim D_{\pi^*}$
- Test: $s_t \sim D_{\pi_\theta}$

When to use Behavioral Cloning?

Advantages

- Simple
- Simple
- Efficient

Use When:

- 1-step deviations not too bad
- Learning reactive behaviors
- Expert trajectories "cover" state space

Disadvantages

- Distribution mismatch between training and testing
- No long term planning

Don't Use When:

- 1-step deviations can lead to catastrophic error
- Optimizing long-term objective (at least not without a stronger model)

Types of Imitation Learning

Behavioral Cloning

$\text{argmin}_\theta E_{(s,a^*)\sim P^*} L(a^*, \pi_\theta(s))$

Works well when P* close to Pθ

Inverse RL

Learn r such that:

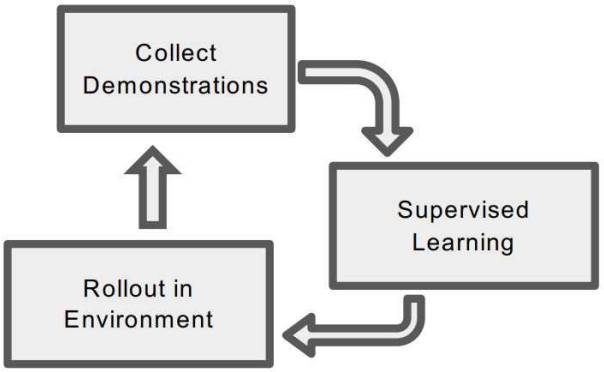
$\pi^* = \text{argmax}_\theta E_{s\sim P(s|\theta)} r(s, \pi_\theta(s))$

RL problem

Assumes learning r is statistically easier than directly learning π^*

Direct Policy Learning

via Interactive Demonstrator



Requires Interactive Demonstrator (BC is 1-step special case)

Interactive Expert

Can query expert at any state

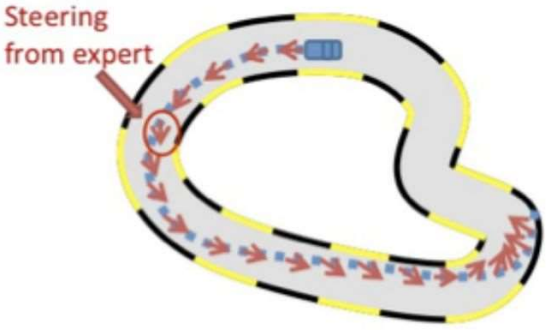
Construct loss function

- $L(\pi^*(s), \pi(s))$

Typically applied to rollout trajectories

- $s \sim P(s|\pi)$

Driving example: $L(\pi^*(s), \pi(s)) = (\pi^*(s) - \pi(s))^2$



Example from Super Tux Kart
(Image courtesy of Stephane Ross)

Expert provides feedback on state visited by policy

Images from Stephane Ross

DAGGER: Dataset Aggregation

Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in Π.
for $i = 1$ **to** N **do**
  Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
  Sample T-step trajectories using π_i.
  Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert.
  Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
  Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
end for
Return best $\hat{\pi}_i$ on validation.

β_i: a decreasing coefficient s.t.
$\frac{1}{N}\sum_{i=1}^{N} \beta_i \rightarrow 0$ as $N \rightarrow \infty$

- Idea: Get more labels of the expert action along the path taken by the policy computed by behavior cloning
- Obtains a stationary deterministic policy with good performance under its induced state distribution

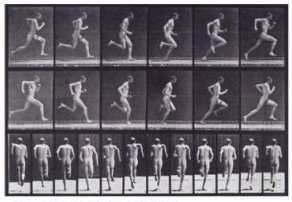
Inverse RL

- What if we don't have an online demonstrator?
  - We only have access to an offline set of demonstrated trajectories
- Behavioral cloning is not robust
  - Suffers from overfitting
  - We know what to do in observed states but can't generalize well to other states
- How can we learn to mimic the demonstrator in a general why?
  - Learn the demonstrator's objective (reward) function
  - Apply RL

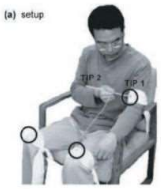
Inverse RL

- What if we don't have an online demonstrator?
  - We only have access to an offline set of demonstrated trajectories
- Behavioral cloning is not robust
  - Suffers from overfitting
  - We know what to do in observed states but can't generalize well to other states
- How can we learn to mimic the demonstrator in a general why?
  - Learn the demonstrator's objective (reward) function
  - Apply RL

Inverse RL


Muybridge (c. 1870)


Mombaur et al. '09



Li & Todorov '06


Ziebart '08

$$\mathbf{a}_1, \ldots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\ldots,\mathbf{a}_T} \sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)$$

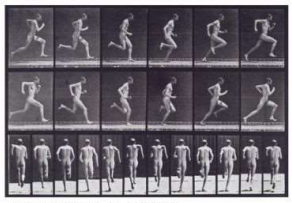
$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$$

optimize this to explain the data

$$\pi = \arg\max_{\pi} E_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t),\mathbf{a}_t\sim\pi(\mathbf{a}_t|\mathbf{s}_t)}[r(\mathbf{s}_t, \mathbf{a}_t)]$$

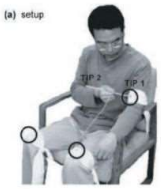
$$\mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{s}_t)$$

Inverse RL



Muybridge (c. 1870)

Mombaur et al. '09



Li & Todorov '06

Ziebart '08

$$\mathbf{a}_1, \ldots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\ldots,\mathbf{a}_T} \sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)$$

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$$

optimize this to explain the data

$$\pi = \arg\max_{\pi} E_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t),\mathbf{a}_t\sim\pi(\mathbf{a}_t|\mathbf{s}_t)}[r(\mathbf{s}_t, \mathbf{a}_t)]$$

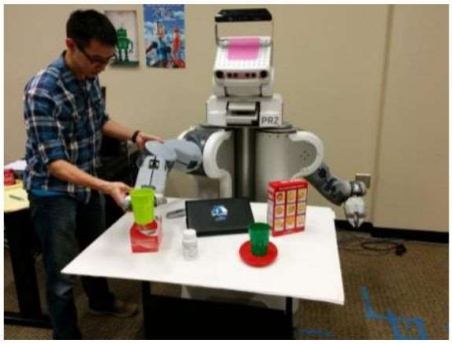
$$\mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{s}_t)$$

Inverse RL

The imitation learning perspective

Standard imitation learning:

- copy the *actions* performed by the expert
- no reasoning about outcomes of actions

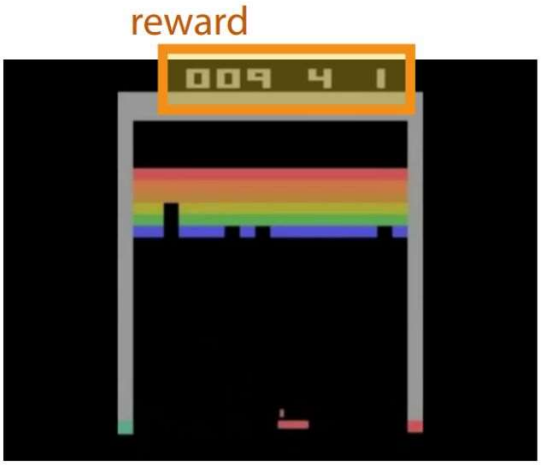


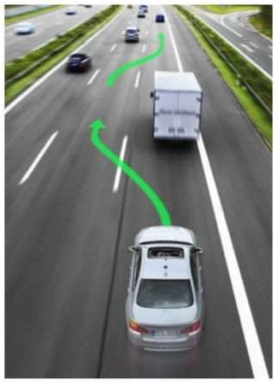
Human imitation learning:

- copy the *intent* of the expert
- might take very different actions!

Inverse RL

The reinforcement learning perspective

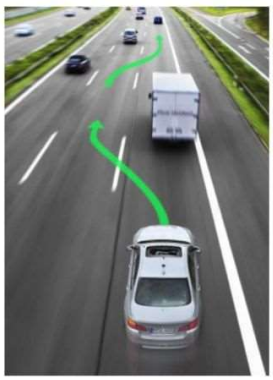




what is the reward?

Inverse RL

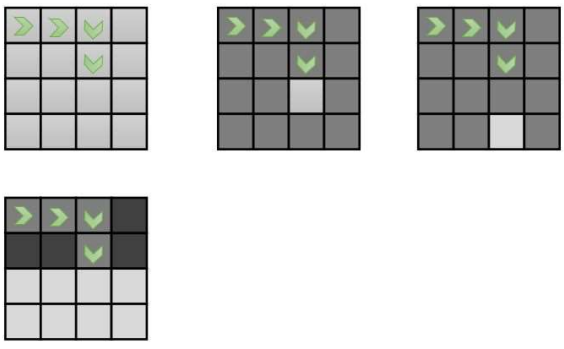
Infer **reward functions** from **demonstrations**



$r(\mathbf{s}, \mathbf{a})$

by itself, this is an **underspecified** problem

many reward functions can explain the **same** behavior



Inverse RL

"forward" reinforcement learning

given:

states $\mathbf{s} \in \mathcal{S}$, actions $\mathbf{a} \in \mathcal{A}$

(sometimes) transitions $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$

reward function $r(\mathbf{s}, \mathbf{a})$

learn $\pi^\star(\mathbf{a}|\mathbf{s})$

inverse reinforcement learning

given:

states $\mathbf{s} \in \mathcal{S}$, actions $\mathbf{a} \in \mathcal{A}$

(sometimes) transitions $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$

samples $\{\tau_i\}$ sampled from $\pi^\star(\tau)$

learn $r_\psi(\mathbf{s}, \mathbf{a})$

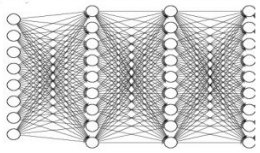
reward parameters

...and then use it to learn $\pi^\star(\mathbf{a}|\mathbf{s})$

linear reward function:

$r_\psi(\mathbf{s}, \mathbf{a}) = \sum_i \psi_i f_i(\mathbf{s}, \mathbf{a}) = \psi^T \mathbf{f}(\mathbf{s}, \mathbf{a})$

neural net reward function:



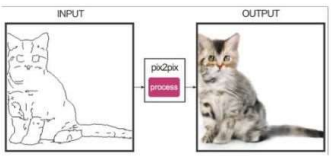
$r_\psi(\mathbf{s}, \mathbf{a})$

parameters ψ

GAN

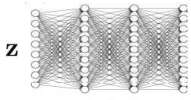
Zhu et al. '17

Arjovsky et al. '17



Isola et al. '17

"generator"



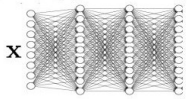
$p_\theta(\mathbf{x}|\mathbf{z})$

samples from $p_\theta(\mathbf{x})$

data ("demonstrations")

samples from $p^\star(\mathbf{x})$

$D(\mathbf{x}) = p_\psi(\text{real image}|\mathbf{x})$



$$\psi = \arg\max_\psi \frac{1}{N} \sum_{\mathbf{x} \sim p^\star} \log D_\psi(\mathbf{x}) + \frac{1}{M} \sum_{\mathbf{x} \sim p_\theta} \log(1 - D_\psi(\mathbf{x}))$$

$$\theta \leftarrow \arg\max_\theta E_{\mathbf{x} \sim p_\theta} \log D_\psi(\mathbf{x})$$

Goodfellow et al. '14

Inverse RL as GAN

generator/policy

$\pi_\theta(\tau)$

samples from $\pi_\theta(\tau)$

data/demonstrations

samples from $p^\star(\tau)$

$$\psi \leftarrow \arg\max_\psi E_{\tau\sim p^\star}[\log D_\psi(\tau)] + E_{\tau\sim\pi_\theta}[\log(1 - D_\psi(\tau))]$$

$$D_\psi(\tau) = \frac{\frac{1}{Z}\exp(r(\tau))}{\prod_t \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \frac{1}{Z}\exp(r(\tau))}$$

$$\nabla_\theta \mathcal{L} \approx \frac{1}{M}\sum_{j=1}^{M} \nabla_\theta \log \pi_\theta(\tau_j) r_\psi(\tau_j)$$

policy changed to make it *harder* to distinguish from demos

Finn*, Christiano* et al. "A Connection Between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models."

Inverse RL as GAN

initial policy π

samples from $\pi_\theta(\tau)$

human demonstrations

samples from $\pi^\star(\tau)$

$$\nabla_\psi \mathcal{L} \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j}\sum_{j=1}^{M} w_j \nabla_\psi r_\psi(\tau_j)$$

demos are made more likely, samples less likely

$$\nabla_\theta \mathcal{L} \approx \frac{1}{M}\sum_{j=1}^{M} \nabla_\theta \log \pi_\theta(\tau_j) r_\psi(\tau_j)$$

policy changed to make it *harder* to distinguish from demos

Required Readings and references

1. Human-in-the-Loop Deep Reinforcement Learning with Application to Autonomous Driving, Jingda Wu, Zhiyu Huang, Chao Huang, Zhongxu Hu, Peng Hang, Yang Xing, Chen Lv*

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Thank you

Multi Agent Reinforcement Learning

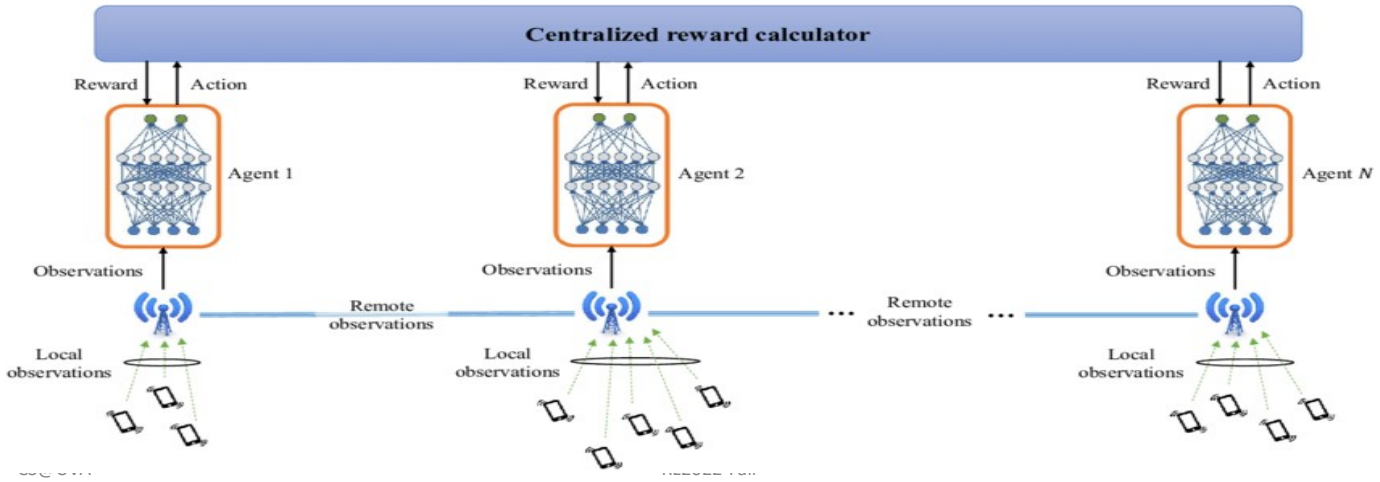
Dr.Chandra Sekhar Vorugunti

Multi Agent Reinforcement Learning (MARL)

What is Multi-Agent Reinforcement Learning?

- Vanilla reinforcement learning is concerned with a single agent, in an environment, seeking to maximize the total reward in that environment.
- It receives rewards for taking steps without falling over, and through trial and error, and maximizing these rewards, the robot eventually learns to walk.

- In this context, we have a single agent seeking to accomplish a goal through maximizing total rewards.



Multi Agent Reinforcement Learning (MARL)

Multi-agent reinforcement learning studies how multiple agents interact in a common environment. That is, when these agents interact with the environment and one another, can we observe them collaborate, coordinate, compete, or collectively learn to accomplish a particular task. It can be further broken down into three broad categories:

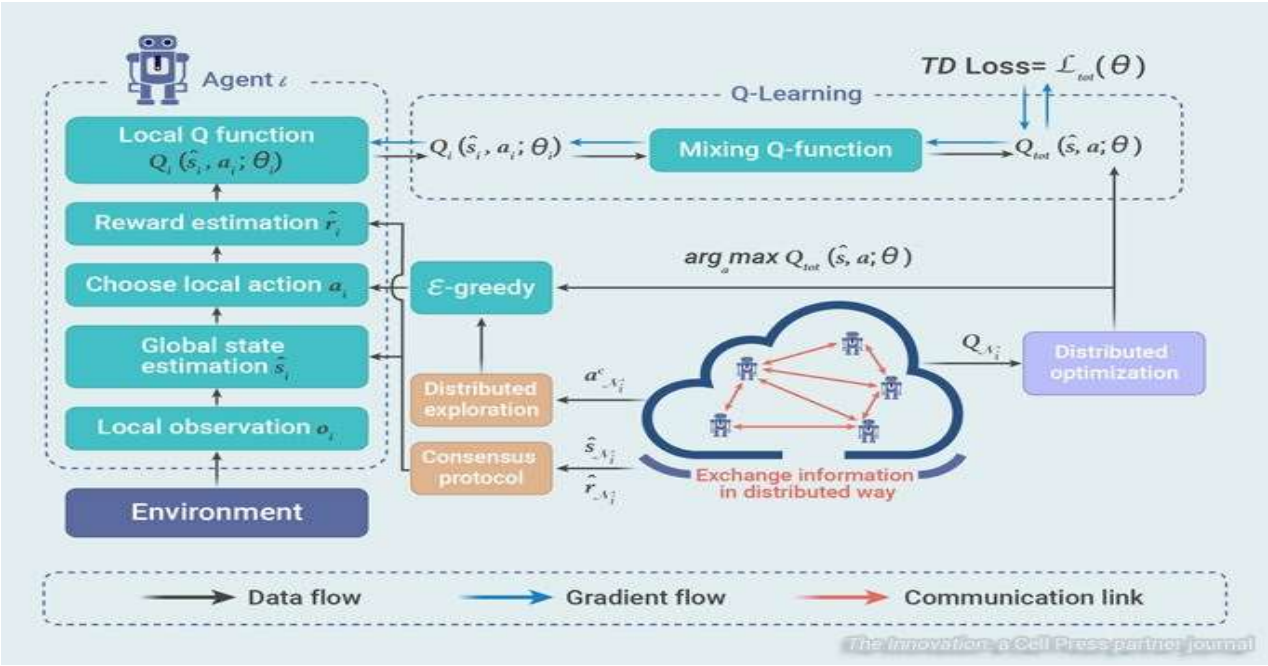
Cooperative: All agents working towards a common goal

Competitive: Agents competing with one another to accomplish a goal

Some mix of the two: Think a 5v5 basketball game, where individuals on the same team are coordinating with one another, but the two teams are competing against one another.

A canonical example of MARL is a swarm of robots seeking to rescue an individual. Each robot has only partial observability of its environment (they can only see a small patch of land below them) therefore the robots need to coordinate with one another to rescue the individual.

Multi Agent Reinforcement Learning (MARL)



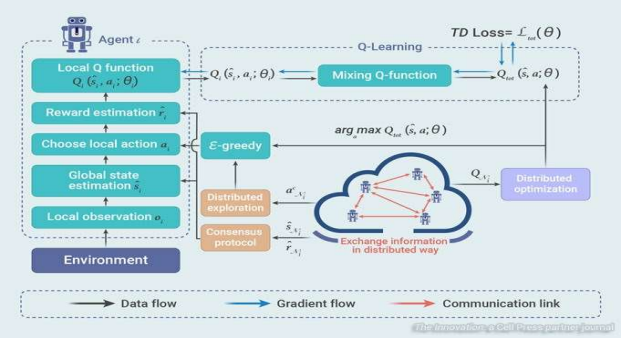
Multi Agent Reinforcement Learning (MARL)

Local Observation O_i: Each agent *i* receives an observation from the environment, which may represent a partial view of the entire state space.

Global State Estimation S_i : The agent uses its observation to estimate the global state of the environment, which is necessary for making informed decisions. In some systems, the global state may be directly observable, or agents may need to communicate to estimate it.

Choose Local Action a_i : Based on its state estimation, the agent selects an action to take. This decision is often made using an ϵ-greedy strategy, where the agent usually takes the best known action but occasionally explores by choosing a random action.

Reward Estimation : After taking an action, the agent receives a reward signal that estimates the immediate benefit of the action taken.



Multi Agent Reinforcement Learning (MARL)

Learning Process

- **Local Q function** $Q_i(s, a; \theta_i)$: Each agent maintains a Q-value function that estimates the expected return of taking action a in state s, parameterized by θ_i.
- **Q-Learning**: Agents use Q-learning to update their Q-value functions. This involves using the reward signal and the maximum estimated future rewards.
- **Mixing Q-function**: In some MARL systems, individual Q-value functions from different agents are combined or 'mixed' to form a global Q-value function that represents the value of joint actions in the global state space.
- **TD Loss** $\mathcal{L}(\theta)$: The Temporal Difference (TD) Loss is used to update the parameters θ of the Q-value functions. It measures the difference between the estimated Q-values and the observed Q-values following an action.

Multi Agent Reinforcement Learning (MARL)

Data and Gradient Flow:

- Solid blue lines indicate the data flow from the environment to the agents and among the agents themselves.
- Dashed blue lines represent the gradient flow, which is the direction in which the parameters of the Q-value functions are updated.
- Red dashed lines symbolize communication links between agents, which are used to exchange information necessary for learning and coordination.

Multi Agent Reinforcement Learning (MARL)

The overall system demonstrates how agents in a multi-agent reinforcement learning setting can independently gather information, learn from it, and collaborate to make decisions that are informed by both their own experiences and the shared knowledge from other agents.

Multi Agent Reinforcement Learning (MARL)

Local Observation o_i

Each agent i observes its local environment, which is typically modeled as a Partially Observable Markov Decision Process (POMDP). The observation o_i may not fully represent the true state s due to partial observability.

Global State Estimation s_i

The agent uses its observation to estimate the global state s. The estimated state s_i could be an approximation of the true state s based on the agent's local information and possibly information shared by other agents. Mathematically, it can be represented as a function of the local observation and potentially others' observations if communication is involved:

$s_i = f(o_i, o_{-i})$

where o_{-i} represents the observations of other agents.

Multi Agent Reinforcement Learning (MARL)

Choose Local Action a_i

The agent selects an action based on its policy $\pi(a_i|s_i)$. An ϵ-greedy policy is often used to balance exploration and exploitation:

$$\pi(a_i|s_i) = \begin{cases} \arg\max_{a_i} Q_i(s_i, a_i; \theta_i) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

where $Q_i(s_i, a_i; \theta_i)$ is the action-value function for agent i.

Reward Estimation $\hat{r}_i$

The reward $\hat{r}_i$ is an immediate signal provided by the environment after the agent takes action a_i, which may also depend on the actions of other agents:

$\hat{r}_i = R_i(s, a_i, a_{-i})$

where R_i is the reward function for agent i, and a_{-i} are the actions of other agents.

Multi Agent Reinforcement Learning (MARL)

Local Q function $Q_i(s, a; \theta_i)$

The Q-value function for each agent i, parameterized by θ_i, estimates the expected cumulative reward of taking action a_i in state s_i and following policy π thereafter:

$Q_i(s_i, a_i; \theta_i) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{i,t+k} \mid s_i, a_i\right]$

Q-Learning

Agents use the Q-learning algorithm to update their value function, where the update rule at time step t can be written as:

$$Q_i^{new}(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha \left[\hat{r}_i + \gamma \max_{a_i'} Q_i(s_i', a_i') - Q_i(s_i, a_i)\right]$$

Here, s_i' is the next state, and α is the learning rate.

Multi Agent Reinforcement Learning (MARL)

Mixing Q-function

In some multi-agent systems, a global Q-value function Q_{tot} is derived by combining the individual Q-functions of the agents through a mixing function f:

$Q_{tot}(s, \mathbf{a}) = f(Q_1(s, a_1), \ldots, Q_N(s, a_N))$

where $\mathbf{a}$ is the joint action of all agents.

TD Loss $\mathcal{L}(\theta)$

The Temporal Difference (TD) Loss for agent i with parameters θ_i is calculated as the squared difference between the predicted Q-values and the target Q-values:

$\mathcal{L}(\theta_i) = \left(\hat{r}_i + \gamma \max_{a_i'} Q_i(s_i', a_i'; \theta_i^-) - Q_i(s_i, a_i; \theta_i)\right)^2$

Here, θ_i^- represents the parameters of the target network, which are periodically updated with the parameters of the current Q-network to stabilize training.

Multi Agent Reinforcement Learning (MARL)

Distributed Exploration and Consensus Protocol

Distributed exploration ensures that agents explore the action space sufficiently. The consensus protocol ensures all agents agree on certain shared values, which might involve averaging gradients or parameters:

$\theta_i = \frac{1}{N} \sum_{j=1}^{N} \theta_j$

Multi Agent Reinforcement Learning (MARL)

1. Local Observation o_i

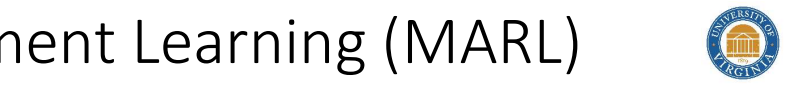
Agents operate based on their own local observations o_i of the environment. This observation may only provide partial information about the full state due to occlusions or distance limitations in the environment. The relationship between observation and the actual state can be probabilistic and is often defined by an observation function:

$P(o_i|s, a_i)$

This probability function models the likelihood of agent i receiving observation o_i after taking action a_i in state s.

Imagine a robot trying to locate a box in a warehouse. It might see the box with a high probability if it's close and there's good lighting (say, 90% chance, or = 0.9, P=0.9). If it's far away or behind obstacles, the chance it sees the box drops (say, 30% chance, or = 0.3 P=0.3). This probability changes based on the robot's action and location.

Multi Agent Reinforcement Learning (MARL)



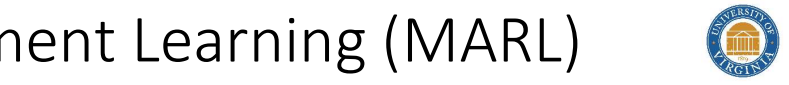
2. Global State Estimation s_i

Each agent uses its local observations to estimate the global state s_i. This estimation can be a simple function of the local observation or a more complex function that incorporates historical data, predictions, and possibly communicated information from other agents:

$s_i = \mathcal{F}(o_i, \mathcal{H}_i, \mathcal{C}_i)$

where $\mathcal{F}$ is the estimation function, $\mathcal{H}_i$ is the agent's historical observations and actions, and $\mathcal{C}_i$ is the communicated information from other agents.

Multi Agent Reinforcement Learning (MARL)



3. Choose Local Action a_i

The action selection process is based on a policy $\pi(a_i|s_i)$, typically derived from a Q-value function that predicts the expected rewards for different actions:

$a_i = \pi(s_i) = \arg\max_{a_i} Q_i(s_i, a_i; \theta_i)$

with probability $1 - \epsilon$, or a random action is selected with probability ϵ to allow for exploration.

4. Reward Estimation $\hat{r}_i$

After actions are taken, agents receive rewards, which may be individual or shared. The reward function $R_i(s, a_i, a_{-i})$ maps the current state and actions of all agents to a reward value. For cooperative agents, this reward might be the same for all agents and can foster collaboration:

$\hat{r}_i = R(s, \mathbf{a})$

where $\mathbf{a}$ is the joint action of all agents and R is the common reward function.

$$\hat{r}_i = R(s, \mathbf{a})$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Multi Agent Reinforcement Learning (MARL)

5. Local Q function $Q_i(s, a; \theta_i)$

Each agent has a Q-value function parameterized by θ_i that estimates the expected cumulative reward for taking an action a_i in state s_i. It is updated by learning from the discrepancy between predicted and actual outcomes (TD error):

$Q_i(s_i, a_i; \theta_i) = \mathbb{E}[r_i + \gamma \max_{a_i'} Q_i(s_i', a_i'; \theta_i)]$

6. Q-Learning

In Q-learning, agents update their Q-value functions iteratively based on the rewards received and the estimated future rewards:

$Q_i^{new}(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha \left[\hat{r}_i + \gamma \max_{a_i'} Q_i(s_i', a_i') - Q_i(s_i, a_i)\right]$

This equation indicates that the Q-value is adjusted towards the reward plus the discounted maximum future reward, modulated by a learning rate α.

Multi Agent Reinforcement Learning (MARL)

The expected return when taking action a in state s is calculated as:

$$Q(s, a) = \mathbb{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots | S_0 = s, A_0 = a]$$

where:

- $\mathbb{E}$ is the expectation operator.
- R_t is the reward received after t time steps.
- γ is the discount factor (between 0 and 1), which determines the importance of future rewards.
- S_0 and A_0 represent the initial state and action.
- The policy π^* guides the selection of future actions.

Q-function Update:

- The Q-function is updated using the Temporal Difference (TD) learning rule:
  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
  where α is the learning rate, and $\max_{a'} Q(s', a')$ is the maximum predicted Q-value for the next state s' over all possible actions.

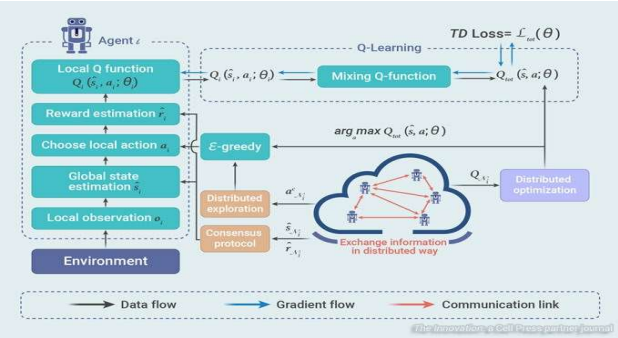
Multi Agent Reinforcement Learning (MARL)

7. Mixing Q-function

In a collaborative MARL setting, there might be a global valu
joint actions. It's a function of individual Q-values and possib

$$Q_{tot}(s, \mathbf{a}; \Theta) = f(Q_1(s, a_1; \theta_1), ..., Q_N(s, a_N; \theta_N), s)$$

where Θ denotes the collective parameters of all agents, and
individual Q-values in a manner that accounts for the joint ac



8. TD Loss $\mathcal{L}(\theta)$

The TD Loss is a measure of how well the Q-value function predicts the observed rewards, plus the future rewards. It's calculated as the squared difference between the predicted Q-values and the target Q-values: